Symfony Docs pt-BR Documentation

Versão 2.4

symfony

27/12/2016

Sumário

CAPÍTULO 1

Guia de Início Rápido

Inicie rapidamente no Symfony2 com o Guia de Início Rápido:

1.1 Guia de Início Rápido

1.1.1 Panorama Geral

Comece a usar o Symfony2 em 10 minutos! Este capítulo irá orientá-lo através de alguns dos conceitos mais importantes por trás do Symfony2 e explicar como você poderá iniciar rapidamente, mostrando-lhe um projeto simples em ação.

Se você já usou um framework web antes, você deve se sentir em casa com o Symfony2. Se não, bem-vindo à uma nova forma de desenvolver aplicações web!

Dica: Quer saber por que e quando você precisa usar um framework? Leia o documento "Symfony em 5 minutos" .

Baixando o Symfony2

Primeiro, verifique se você tem um servidor web instalado e configurado (como o Apache) com a versão mais recente possível do PHP (é recomendado o PHP 5.3.8 ou mais recente).

Pronto? Comece fazendo o download da "Edição Standard do Symfony2", uma distribuição do Symfony que é pré-configurada para os casos de uso mais comuns e contém também um código que demonstra como usar Symfony2 (obtenha o arquivo com os *vendors* incluídos para começar ainda mais rápido).

Após descompactar o arquivo no seu diretório raiz do servidor web, você deve ter um diretório `Symfony/` parecido com o seguinte:

```
www/ <- your web root directory
    Symfony/ <- the unpacked archive
        app/
            cache/
            config/
            logs/
            Resources/
        bin/
        src/
            Acme/
                DemoBundle/
```

```
                    Controller/
                    Resources/
                    ...
        vendor/
            symfony/
            doctrine/
            ...
        web/
            app.php
            ...
```

Nota: Se você está familiarizado com o Composer, poderá executar o seguinte comando em vez de baixar o arquivo:

```
$ composer.phar create-project symfony/framework-standard-edition path/to/install 2.1.x-dev

# remove the Git history
$ rm -rf .git

Para uma versão exata, substitua `2.1.x-dev` com a versão mais recente do Symfony
(Ex. 2.1.1). Para detalhes, veja a `Página de Instalação do Symfony`_
```

Dica: Se você tem o PHP 5.4, é possível usar o servidor web integrado:

```
# check your PHP CLI configuration
$ php ./app/check.php

# run the built-in web server
$ php ./app/console server:run
```

Então, a URL para a sua aplicação será "http://localhost:8000/app_dev.php"

O servidor integrado deve ser usado apenas para fins de desenvolvimento, mas pode ajudá-lo a iniciar o seu projeto de forma rápida e fácil.

Verificando a configuração

O Symfony2 vem com uma interface visual para teste da configuração do servidor que ajuda a evitar algumas dores de cabeça que originam da má configuração do servidor Web ou do PHP. Use a seguinte URL para ver o diagnóstico para a sua máquina:

```
http://localhost/config.php
```

Nota: Todos as URLs de exemplo assumem que você baixou e descompactou o Symfony diretamente no diretório raiz web do seu servidor web. Se você seguiu as instruções acima e descompactou o diretório *Symfony* em seu raiz web, então, adicione */Symfony/web* após *localhost* em todas as URLs:

```
http://localhost/Symfony/web/config.php
```

Se houver quaisquer questões pendentes informadas, corrija-as. Você também pode ajustar a sua configuração, seguindo todas as recomendações. Quando tudo estiver certo, clique em "*Bypass configuration and go to the Welcome page*" para solicitar a sua primeira página "real" do Symfony2:

```
http://localhost/app_dev.php/
```

O Symfony2 lhe dá as boas vindas e parabeniza-o por seu trabalho até agora!

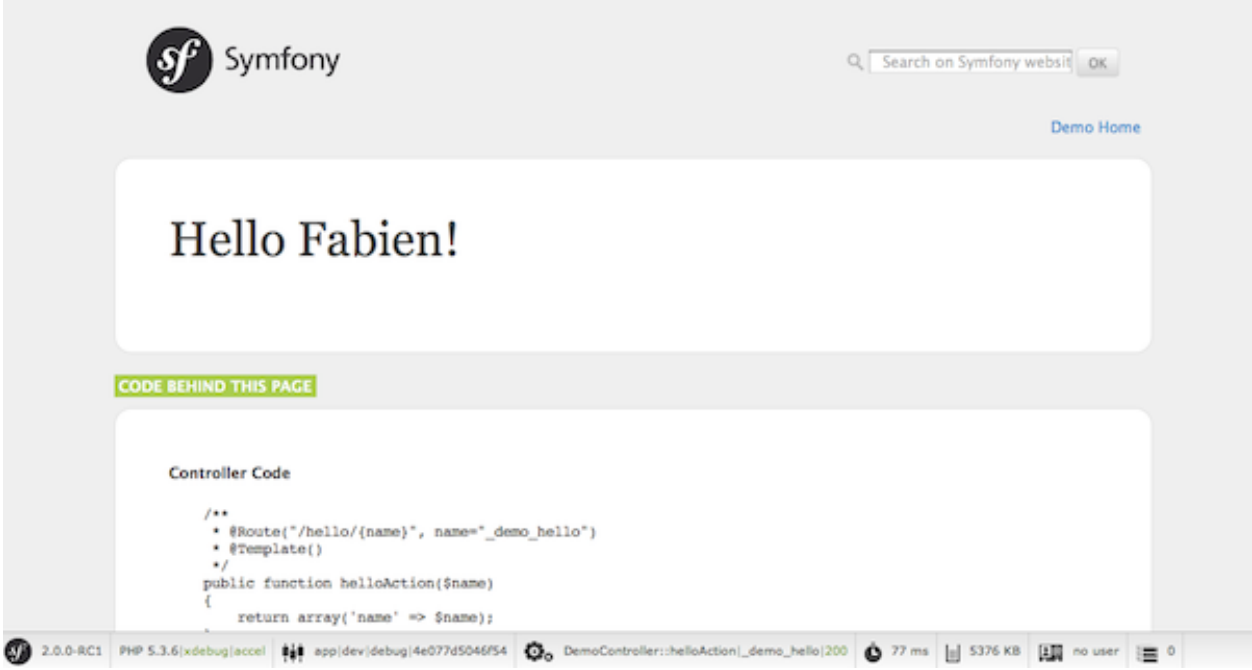
Compreendendo os Fundamentos

Um dos objetivos principais de um framework é garantir a Separação de Responsabilidades. Isso mantém o seu código organizado e permite que a sua aplicação evolua facilmente ao longo do tempo, evitando a mistura de chamadas ao banco de dados, de tags HTML e de lógica de negócios no mesmo script. Para atingir este objetivo com o Symfony, primeiro você precisa aprender alguns conceitos e termos fundamentais.

Dica: Quer uma prova de que o uso de um framework é melhor do que misturar tudo no mesmo script? Leia o capítulo "Symfony2 versus o PHP puro" do livro.

A distribuição vem com um código de exemplo que você pode usar para aprender mais sobre os principais conceitos do Symfony2. Vá para a seguinte URL para ser cumprimentado pelo Symfony2 (substitua *Fabien* pelo seu primeiro nome):

```
http://localhost/app_dev.php/demo/hello/Fabien
```



O que está acontecendo aqui? Vamos dissecar a URL:

- `app_dev.php`: Este é o front controller. É o único ponto de entrada da aplicação e responde à todas as solicitações dos usuários;
- `/demo/hello/Fabien`: Este é o *caminho virtual* para o recurso que o usuário quer acessar.

Sua responsabilidade como desenvolvedor é escrever o código que mapeia a *solicitação* do usuário (`/demo/hello/Fabien`) para o *recurso* associado à ela (a página HTML `Hello Fabien!`).

Roteamento

O Symfony2 encaminha a solicitação para o código que lida com ela, tentando corresponder a URL solicitada contra alguns padrões configurados. Por predefinição, esses padrões (chamados de rotas) são definidos no arquivo de configuração `app/config/routing.yml`. Quando você está no *ambiente* `dev` - indicado pelo front controller app_**dev**.php - o arquivo de configuração `app/config/routing_dev.yml` também é carregado. Na Edição Standard, as rotas para estas páginas "demo" são colocadas no arquivo:

```
# app/config/routing_dev.yml
_welcome:
    pattern:  /
    defaults: { _controller: AcmeDemoBundle:Welcome:index }

_demo:
    resource: "@AcmeDemoBundle/Controller/DemoController.php"
    type:     annotation
    prefix:   /demo

# ...
```

As três primeiras linhas (após o comentário) definem o código que é executado quando o usuário solicita o recurso "/" (ou seja, a página de boas-vindas que você viu anteriormente). Quando solicitado, o controlador `AcmeDemoBundle:Welcome:index` será executado. Na próxima seção, você vai aprender exatamente o que isso significa.

Dica: A Edição Standard do Symfony2 usa YAML para seus arquivos de configuração, mas o Symfony2 também suporta XML, PHP e anotações nativamente. Os diferentes formatos são compatíveis e podem ser utilizados alternadamente dentro de uma aplicação. Além disso, o desempenho de sua aplicação não depende do formato de configuração que você escolher, pois tudo é armazenado em cache na primeira solicitação.

Controladores

Controlador é um nome fantasia para uma função ou método PHP que manipula as *solicitações* de entrada e retorna *respostas* (código HTML, na maioria das vezes). Em vez de usar as variáveis globais e funções do PHP (como `$_GET` ou `header()`) para gerenciar essas mensagens HTTP, o Symfony usa objetos: `Request` e `Response`. O controlador mais simples possível pode criar a resposta manualmente, com base na solicitação:

```
use Symfony\Component\HttpFoundation\Response;

$name = $request->query->get('name');

return new Response('Hello '.$name, 200, array('Content-Type' => 'text/plain'));
```

Nota: O Symfony2 engloba a Especificação HTTP, que são as regras que regem toda a comunicação na Web. Leia o capítulo "Fundamentos de Symfony e HTTP" do livro para aprender mais sobre ela e o poder que isso acrescenta.

O Symfony2 escolhe o controlador com base no valor `_controller` da configuração de roteamento: `AcmeDemoBundle:Welcome:index`. Esta string é o *nome lógico* do controlador, e ela referencia o método `indexAction` da classe `Acme\DemoBundle\Controller\WelcomeController`:

```
// src/Acme/DemoBundle/Controller/WelcomeController.php
namespace Acme\DemoBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class WelcomeController extends Controller
{
    public function indexAction()
    {
        return $this->render('AcmeDemoBundle:Welcome:index.html.twig');
    }
}
```

Dica: Você poderia ter usado o nome completo da classe e do método - `Acme\DemoBundle\Controller\WelcomeController::indexAction` - para o valor `_controller`. Mas, se você seguir algumas convenções simples, o nome lógico é menor e permite mais flexibilidade.

A classe `WelcomeController` estende a classe nativa `Controller`, que fornece métodos de atalho úteis, tal como o método `render()` que carrega e renderiza um template (`AcmeDemoBundle:Welcome:index.html.twig`). O valor retornado é um objeto Response populado com o conteúdo processado. Assim, se as necessidades surgirem, o Response pode ser ajustado antes de ser enviado ao navegador:

```
public function indexAction()
{
    $response = $this->render('AcmeDemoBundle:Welcome:index.txt.twig');
```

```
    $response->headers->set('Content-Type', 'text/plain');

    return $response;
}
```

Não importa como você faz isso, o objetivo final do seu controlador sempre será retornar o objeto `Response` que deve ser devolvido ao usuário. Este objeto `Response` pode ser populado com código HTML, representar um redirecionamento do cliente, ou mesmo retornar o conteúdo de uma imagem JPG com um cabeçalho `Content-Type` de `image/jpg`.

Dica: Estender a classe base `Controller` é opcional. De fato um controlador pode ser uma função PHP simples ou até mesmo uma closure PHP. O capítulo "O Controlador" do livro lhe ensina tudo sobre os controladores do Symfony2.

O nome do template, `AcmeDemoBundle:Welcome:index.html.twig`, é o *nome lógico* do template e faz referência ao arquivo `Resources/views/Welcome/index.html.twig` dentro do `AcmeDemoBundle` (localizado em `src/Acme/DemoBundle`). A seção bundles abaixo irá explicar porque isso é útil.

Agora, dê uma olhada novamente na configuração de roteamento e encontre a chave `_demo`.

```
# app/config/routing_dev.yml
_demo:
    resource: "@AcmeDemoBundle/Controller/DemoController.php"
    type:     annotation
    prefix:   /demo
```

O Symfony2 pode ler/importar as informações de roteamento de diferentes arquivos escritos em YAML, XML, PHP ou até mesmo incorporado em anotações PHP. Aqui, o *nome lógico* do arquivo é `@AcmeDemoBundle/Controller/DemoController.php` e refere-se ao arquivo `src/Acme/DemoBundle/Controller/DemoController.php` . Neste arquivo, as rotas são definidas como anotações nos métodos da ação:

```
// src/Acme/DemoBundle/Controller/DemoController.php
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Template;

class DemoController extends Controller
{
    /**
     * @Route("/hello/{name}", name="_demo_hello")
     * @Template()
     */
    public function helloAction($name)
    {
        return array('name' => $name);
    }

    // ...
}
```

A anotação `@Route()` define uma nova rota com um padrão `/hello/{name}` que executa o método `helloAction` quando corresponder. A string entre chaves como `{name}` é chamada de placeholder. Como você pode ver, o seu valor pode ser obtido através do argumento do método `$name`.

Nota: Mesmo as anotações não sendo suportadas nativamente pelo PHP, você as usará extensivamente no Symfony2 como uma forma conveniente de configurar o comportamento do framework e manter a configuração próxima ao

código.

Se você verificar o código do controlador, poderá ver que em vez de renderizar um template e retornar um objeto `Response` como antes, ele apenas retorna um array de parâmetros. A anotação `@Template()` diz ao Symfony para renderizar o template para você, passando cada variável do array ao template. O nome do template que é renderizado segue o nome do controlador. Assim, neste exemplo, o template `AcmeDemoBundle:Demo:hello.html.twig` é renderizado (localizado em `src/Acme/DemoBundle/Resources/views/Demo/hello.html.twig`).

Dica: As anotações `@Route()` e `@Template()` são mais poderosas do que os exemplos simples mostrados neste tutorial. Saiba mais sobre "**'anotações em controladores'_**" na documentação oficial.

Templates

O controlador renderiza o template `src/Acme/DemoBundle/Resources/views/Demo/hello.html.twig` (ou `AcmeDemoBundle:Demo:hello.html.twig` se você usar o nome lógico):

```
{# src/Acme/DemoBundle/Resources/views/Demo/hello.html.twig #}
{% extends "AcmeDemoBundle::layout.html.twig" %}

{% block title "Hello " ~ name %}

{% block content %}
    <h1>Hello {{ name }}!</h1>
{% endblock %}
```

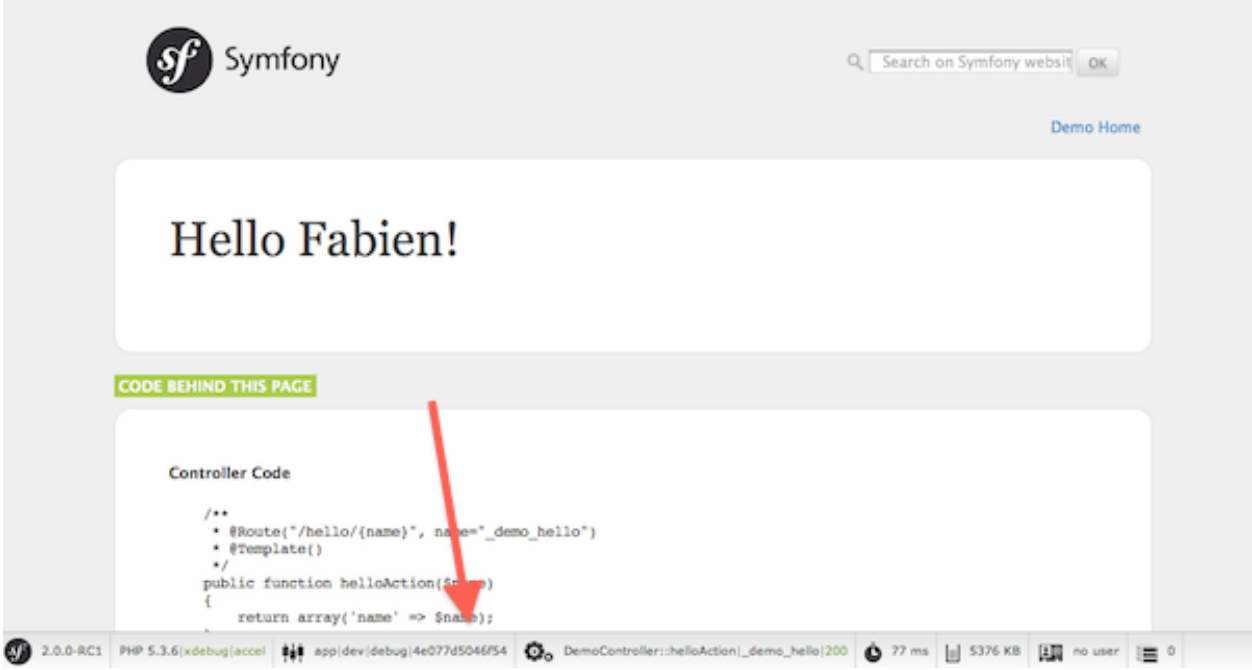
Por padrão, o Symfony2 usa o Twig como seu template engine, mas você também pode usar templates tradicionais PHP se você escolher. No próximo capítulo apresentaremos como os templates funcionam no Symfony2.

Bundles

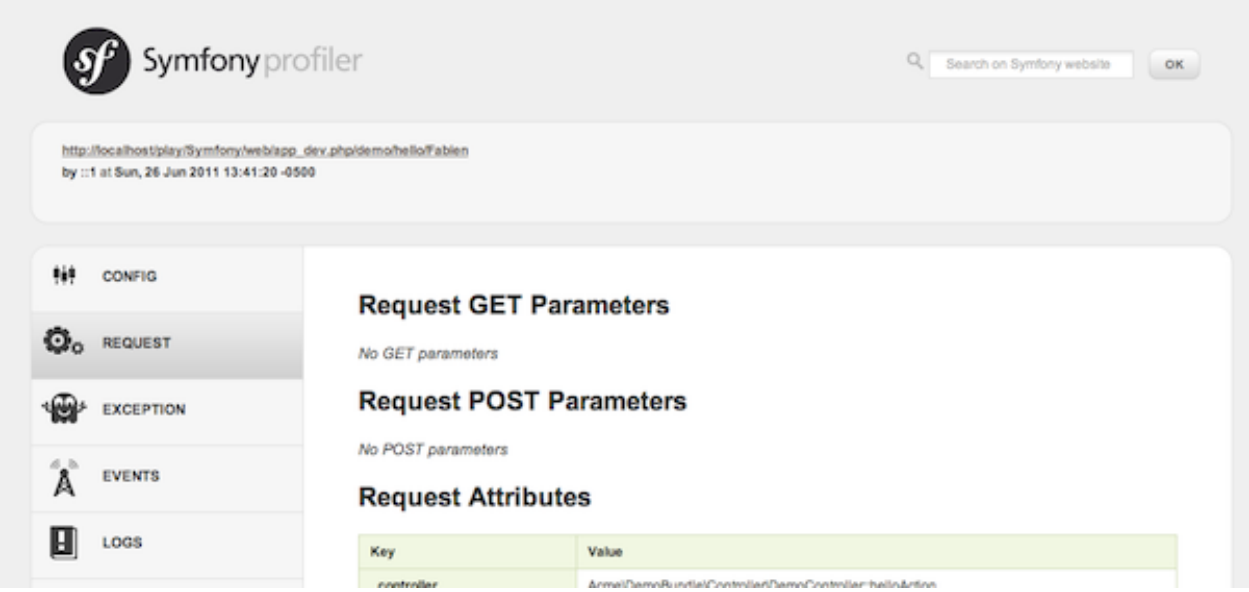
Você pode ter se perguntado por que a palavra bundle é usada em muitos nomes que vimos até agora. Todo o código que você escreve para a sua aplicação está organizado em bundles. Na forma de falar do Symfony2, um bundle é um conjunto estruturado de arquivos (arquivos PHP, folhas de estilo, JavaScripts, imagens, ...) que implementam uma funcionalidade única (um blog, um fórum, ...) e que podem ser facilmente compartilhados com outros desenvolvedores. Até agora, manipulamos um bundle, `AcmeDemoBundle`. Você vai aprender mais sobre bundles no último capítulo deste tutorial.

Trabalhando com Ambientes

Agora que você tem uma compreensão melhor de como funciona o Symfony2, verifique a parte inferior de qualquer página renderizada com o Symfony2. Você deve observar uma pequena barra com o logotipo do Symfony2. Isso é chamado de "Barra de ferramentas para Debug Web" e é a melhor amiga do desenvolvedor.



Mas, o que você vê inicialmente é apenas a ponta do iceberg; clique sobre o estranho número hexadecimal para revelar mais uma ferramenta de depuração muito útil do Symfony2: o profiler.



É claro, você não vai querer mostrar essas ferramentas quando implantar a sua aplicação em produção. É por isso que você vai encontrar um outro front controller no diretório `web/` (`app.php`), que é otimizado para o ambiente de produção:

```
http://localhost/app.php/demo/hello/Fabien
```

E, se você usar o Apache com o `mod_rewrite` habilitado, poderá até omitir a parte `app.php` da URL:

```
http://localhost/demo/hello/Fabien
```

Por último, mas não menos importante, nos servidores de produção, você deve apontar seu diretório raiz web para o diretório `web/` para proteger sua instalação e ter uma URL ainda melhor:

```
http://localhost/demo/hello/Fabien
```

Nota: Note que as três URLs acima são fornecidas aqui apenas como **exemplos** de como uma URL parece quando o front controller de produção é usado (com ou sem mod_rewrite). Se você realmente experimentá-los em uma instalação do *Symfony Standard Edition* você receberá um erro 404 pois o *AcmeDemoBundle* está habilitado somente no ambiente dev e suas rotas importam o *app/config/routing_dev.yml*.

Para fazer a sua aplicação responder mais rápido, o Symfony2 mantém um cache sob o diretório `app/cache/`. No ambiente de desenvolvimento (`app_dev.php`), esse cache é liberado automaticamente sempre que fizer alterações em qualquer código ou configuração. Mas esse não é o caso do ambiente de produção (`app.php`) onde o desempenho é fundamental. É por isso que você deve sempre usar o ambiente de desenvolvimento ao desenvolver a sua aplicação.

Diferentes ambientes de uma dada aplicação diferem apenas na sua configuração. Na verdade, uma configuração pode herdar de outra:

```
# app/config/config_dev.yml
imports:
    - { resource: config.yml }

web_profiler:
    toolbar: true
    intercept_redirects: false
```

O ambiente `dev` (que carrega o arquivo de configuração `config_dev.yml`) importa o arquivo global `config.yml` e, em seguida, modifica-o, neste exemplo, habilitando a barra de ferramentas para debug web.

Considerações Finais

Parabéns! Você já teve a sua primeira amostra de código do Symfony2. Isso não foi tão difícil, foi? Há muito mais para explorar, mas, você já deve ter notado como o Symfony2 torna muito fácil implementar web sites de forma melhor e mais rápida. Se você está ansioso para aprender mais sobre o Symfony2, mergulhe na próxima seção: "A Visão".

1.1.2 A View

Depois de ler a primeira parte desse tutorial, você decidiu que o Symfony2 vale pelo menos mais 10 minutos? Boa escolha! Nessa segunda parte, você vai aprender sobre o sistema de template do Symfony2, o Twig. Ele é um sistema de templates para PHP flexível, rápido e seguro. Ele faz com que seus templates sejam mais legíveis e concisos e também os torna mais amigáveis para os web designers.

Nota: Em vez do Twig, você também pode usar `PHP` para os seus templates. Ambos são suportados pelo Symfony2.

Familiarizando-se com o Twig

Dica: Se quiser aprender a usar o Twig, nós recomendamos fortemente que leia a documentação oficial dele. Essa seção é apenas uma visão geral sobre os principais conceitos.

Um template Twig é um arquivo de texto que pode gerar qualquer tipo de conteúdo (HTML, XML, CSV, LaTex, ...). O Twig define dois tipos de delimitadores:

- `{{ ... }}`: Imprime uma variável ou o resultado de uma expressão;
- `{% ... %}`: Controla a lógica do template; é usado para executar loops `for` e instruções `if`, por exemplo.

Abaixo temos um template mínimo que ilustra alguns comandos básicos usando as duas váriaveis, `page_title` e `navigation`, que são passadas para o template:

```
<!DOCTYPE html>
<html>
    <head>
        <title>My Webpage</title>
    </head>
    <body>
        <h1>{{ page_title }}</h1>

        <ul id="navigation">
            {% for item in navigation %}
                <li><a href="{{ item.href }}">{{ item.caption }}</a></li>
            {% endfor %}
        </ul>
    </body>
</html>
```

Dica: Podem ser incluídos comentários nos templates usando o delimitador `{# ... #}`.

Para renderizar um template no Symfony, use o método `render` a partir do controller, e passe para ele todas as variáveis necessárias ao template:

```
$this->render('AcmeDemoBundle:Demo:hello.html.twig', array(
    'name' => $name,
));
```

As variáveis passadas para o template podem ser strings, arrays ou até objetos. O Twig abstrai a diferença entre eles e deixa acessar os "atributos" de uma variável usando dot notation (`.`):

```
{# array('name' => 'Fabien') #}
{{ name }}

{# array('user' => array('name' => 'Fabien')) #}
{{ user.name }}

{# force array lookup #}
{{ user['name'] }}

{# array('user' => new User('Fabien')) #}
{{ user.name }}
{{ user.getName }}

{# force method name lookup #}
{{ user.name() }}
{{ user.getName() }}

{# pass arguments to a method #}
{{ user.date('Y-m-d') }}
```

Nota: É importante saber que as chaves não fazem parte da variável mas sim do comando de impressão. Se você

acessar variáveis em tags não coloque as chaves em volta delas.

Decorando os Templates

É frequente em um projeto que os templates compartilhem elementos comuns, como os bem-conhecidos cabeçalho e rodapé. No Symfony2, gostamos de enxergar essa situação de uma forma diferente: um template pode ser decorado por outro. Funciona exatamente do mesmo jeito que nas classes PHP: a herança de templates permite que se construa o template base "layout", que contém todos os elementos comuns do seu site, e define "blocos" que os templates filhos podem sobrescrever.

O template `hello.html.twig` herda do `layout.html.twig`, graças a tag `extends`:

```
{# src/Acme/DemoBundle/Resources/views/Demo/hello.html.twig #}
{% extends "AcmeDemoBundle::layout.html.twig" %}

{% block title "Hello " ~ name %}

{% block content %}
    <h1>Hello {{ name }}!</h1>
{% endblock %}
```

A notação `AcmeDemoBundle::layout.html.twig` parece familiar, não é mesmo? Ela é a mesma notação usada para referenciar um template normal. A parte `::` significa simplesmente que o elemento controller está vazio, então o arquivo correspondente é guardado diretamente no diretório `Resources/views/`.

Agora, vamos dar uma olhada em um `layout.html.twig` simplificado:

```
{# src/Acme/DemoBundle/Resources/views/layout.html.twig #}
<div class="symfony-content">
    {% block content %}
    {% endblock %}
</div>
```

As tags `{% block %}` definem blocos que os templates filhos podem preencher. Tudo o que essas tags fazem é dizer ao sistema de template que um filho pode sobrescrever aquelas partes de seu template pai.

Nesse exemplo, o template `hello.html.twig` sobrescreve o bloco ``content`, que significa que o texto "Hello Fabien" é renderizado dentro do elemento `div.symfony-content`

Usando Tags, Filtros e Funções

Uma das melhores funcionalidades do Twig é sua extensibilidade por meio de tags, filtros e funções. O Symfony2 já vem com muitos desses embutidos facilitando o trabalho do designer de templates.

Incluindo outros Templates

A melhor forma de compartilhar um trecho de código entre vários templates distintos é criar um novo desses que possa ser incluído nos outros.

Crie um template `embedded.html.twig`:

```
{# src/Acme/DemoBundle/Resources/views/Demo/embedded.html.twig #}
Hello {{ name }}
```

E altere o template `index.html.twig` para incluí-lo:

```
{# src/Acme/DemoBundle/Resources/views/Demo/hello.html.twig #}
{% extends "AcmeDemoBundle::layout.html.twig" %}

{# override the body block from embedded.html.twig #}
{% block content %}
    {% include "AcmeDemoBundle:Demo:embedded.html.twig" %}
{% endblock %}
```

Incorporando outros Controllers

E o que fazer se você quiser incorporar o resultado de um outro controller em um template? Isso é muito útil quando estiver trabalhado com Ajax, ou quando o template incorporado precisa de alguma variável que não está disponível no template principal.

Suponha que você tenha criado uma action `fancy`, e quer incluí-la dentro do template `index`. Para fazer isso, use a tag `render`:

```
{# src/Acme/DemoBundle/Resources/views/Demo/index.html.twig #}
{% render "AcmeDemoBundle:Demo:fancy" with { 'name': name, 'color': 'green' } %}
```

Aqui, a string `AcmeDemoBundle:Demo:fancy` se refere a action `fancy` do controller `Demo`. Os argumentos (`name``e ``color`) agem como variáveis de requisições simuladas (como se `fancyAction` estivesse manipulando uma requisição totalmente nova) e ficam disponíveis para o controller:

```
// src/Acme/DemoBundle/Controller/DemoController.php

class DemoController extends Controller
{
    public function fancyAction($name, $color)
    {
        // create some object, based on the $color variable
        $object = ...;

        return $this->render('AcmeDemoBundle:Demo:fancy.html.twig', array('name' => $name, 'object' =
    }

    // ...
}
```

Criando Links entre Páginas

Quando estamos falando de aplicações web, a criação de links entre páginas é uma obrigação. Em vez de fazer "hardcode" das URLS nos templates, usamos a função `path` que sabe como gerar URLs baseando-se na configuração das rotas. Dessa forma, todas as URLs podem ser atualizadas facilmente apenas mudando essa configuração:

```
<a href="{{ path('_demo_hello', { 'name': 'Thomas' }) }}">Greet Thomas!</a>
```

A função `path` pega o nome da rota e um array de parâmetros como argumentos. O nome da rota é a chave principal sob a qual as rotas são referenciadas e os parâmetros são os valores dos marcadores definidos no padrão da rota:

```
// src/Acme/DemoBundle/Controller/DemoController.php
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Template;

/**
 * @Route("/hello/{name}", name="_demo_hello")
```

```
 * @Template()
 */
public function helloAction($name)
{
    return array('name' => $name);
}
```

Dica: A função `url` cria URLs *absolutas*: `{{ url('_demo_hello', { 'name': 'Thomas' }) }}`.

Incluindo Assets: imagens, JavaScripts e folhas de estilo

O que seria da Internet sem as imagens, os JavaScripts e as folhas de estilo? O Symfony2 fornece a função `asset` para lidar com eles de forma fácil:

```
<link href="{{ asset('css/blog.css') }}" rel="stylesheet" type="text/css" />

<img src="{{ asset('images/logo.png') }}" />
```

O objetivo principal da função `asset` é deixar sua aplicação mais portátil. Graças a ela, você pode mover o diretório raiz da aplicação para qualquer lugar no diretório web root sem mudar nem uma linha no código de seus templates.

Escapando Variáveis

O Twig é configurado por padrão para escapar automaticamente toda a saída de dados. Leia a documentação do Twig para aprender mais sobre como escapar a saída de dados e sobre a extensão Escaper.

Considerações Finais

O Twig é simples mas poderoso. Graças a inclusão de layouts, blocos, templates e actions, é muito fácil organizar seus templates de uma maneira lógica e extensível. No entanto se você não estiver confortável com o Twig sempre poderá usar templates PHP no Symfony sem problemas.

Você está trabalhando com o Symfony2 há apenas 20 minutos, mas já pode fazer coisas incríveis com ele. Esse é o poder do Symfony2. Aprender a base é fácil, e logo você aprenderá que essa simplicidade está escondida debaixo de uma arquitetura muito flexível.

Mas eu já estou me adiantando. Primeiro, você precisa aprender mais sobre o controller e esse é exatamente o assunto da próxima parte do tutorial. Pronto para mais 10 minutos de Symfony2?

1.1.3 O Controller

Ainda está com a gente depois das primeiras duas partes? Então você já está se tornando um viciado no Symfony2! Sem mais delongas, vamos descobrir o que os controllers podem fazer por você.

Usando Formatos

Atualmente, uma aplicação web deve ser capaz de entregar mais do que apenas páginas HTML. Desde XML para feeds RSS ou Web Services, até JSON para requisições Ajax, existem muitos formatos diferentes para escolher. Dar suporte para esses formatos no Symfony2 é simples. É só ajustar a rota, como aqui que acrescentando um valor padrão `xml` para a variável `_format`:

```
// src/Acme/DemoBundle/Controller/DemoController.php
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Template;

/**
 * @Route("/hello/{name}", defaults={"_format"="xml"}, name="_demo_hello")
 * @Template()
 */
public function helloAction($name)
{
    return array('name' => $name);
}
```

Usando o formato de requisição (como definido pelo valor `_format`), o Symfony2 automaticamente seleciona o template correto, nesse caso o `hello.xml.twig`:

```
<!-- src/Acme/DemoBundle/Resources/views/Demo/hello.xml.twig -->
<hello>
    <name>{{ name }}</name>
</hello>
```

Isso é tudo. Para os formatos padrão, o Symfony2 também irá escolher automaticamente o melhor cabeçalho `Content-Type` para a resposta. Se você quiser dar suporte para diferentes formatos numa única action, em vez disso use o marcador `{_format}` no padrão da rota:

```
// src/Acme/DemoBundle/Controller/DemoController.php
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Template;

/**
 * @Route("/hello/{name}.{_format}", defaults={"_format"="html"}, requirements={"_format"="html|xml|
 * @Template()
 */
public function helloAction($name)
{
    return array('name' => $name);
}
```

O controller agora será chamado por URLs parecidas com `/demo/hello/Fabien.xml` ou `/demo/hello/Fabien.json`.

A entrada `requirements` define expressões regulares que os marcadores precisam casar. Nesse exemplo, se você tentar requisitar `/demo/hello/Fabien.js` irá receber um erro HTTP 404 pois a requisição não casa com o requisito `_format`.

Redirecionamento e Encaminhamento

Se você quiser redirecionar o usuário para outra página, use o método `redirect()`:

```
return $this->redirect($this->generateUrl('_demo_hello', array('name' => 'Lucas')));
```

O método `generateUrl()` é o mesmo que a função `path()` que usamos nos templates. Ele pega o nome da rota e um array de parâmetros como argumentos e retorna a URL amigável associada.

Você também pode facilmente encaminhar a action para uma outra com o método `forward()`. Internamente, o Symfony faz uma "sub-requisição", e retorna o objeto `Response` daquela sub-requisição:

```
$response = $this->forward('AcmeDemoBundle:Hello:fancy', array('name' => $name, 'color' => 'green'));

// faça algo com a resposta ou a retorne diretamente
```

Pegando informação da Requisição

Além dos valores dos marcadores de rota, o controller tem acesso ao objeto `Request`:

```
$request = $this->getRequest();

$request->isXmlHttpRequest(); // essa é uma requisição Ajax?

$request->getPreferredLanguage(array('en', 'fr'));

$request->query->get('page'); // pega um parâmetro $_GET

$request->request->get('page'); // pega um parâmetro $_POST
```

Em um template, você também pode acessar o objeto `Request` via a variável `app.request`:

```
{{ app.request.query.get('page') }}

{{ app.request.parameter('page') }}
```

Persistindo os Dados na Sessão

Mesmo o protocolo HTTP sendo stateless (não tendo monitoração de estado), o Symfony fornece um objeto interessante que representa o cliente (seja ele uma pessoa real utilizando um navegador, um bot ou um web service). Entre duas requisições, o Symfony2 guarda os atributos num cookie usando sessões nativas do PHP.

É fácil guardar e recuperar a informação da sessão a partir de qualquer controller:

```
$session = $this->getRequest()->getSession();

// guarda um atributo para reutilização na próxima requisição do usuário
$session->set('foo', 'bar');

// em outro controller para outra requisição
$foo = $session->get('foo');

// usa um valor default se a chave não existe
$filters = $session->set('filters', array());
```

Você pode guardar pequenas mensagens que ficarão disponíveis apenas para a próxima requisição:

```
// guarda uma mensagem para a próxima requisição somente (em um controller)
$session->getFlashBag()->add('notice', 'Congratulations, your action succeeded!');

// exibe quaisquer mensagens no próximo pedido (no template)

{% for flashMessage in app.session.flashbag.get('notice') %}
    <div>{{ flashMessage }}</div>
{% endfor %}
```

Isso é útil quando você precisa definir uma mensagem de sucesso antes de redirecionar o usuário para outra página (que então mostrará a mensagem). Por favor note que quando você usa has() ao invés de get(), a mensagem flash não será apagada e, assim, permanece disponível durante os pedidos seguintes.

Protegendo Recursos

A versão Standard Edition do Symfony vem com uma configuração de segurança simples que atende as necessidades mais comuns:

```
# app/config/security.yml
security:
    encoders:
        Symfony\Component\Security\Core\User\User: plaintext

    role_hierarchy:
        ROLE_ADMIN:       ROLE_USER
        ROLE_SUPER_ADMIN: [ROLE_USER, ROLE_ADMIN, ROLE_ALLOWED_TO_SWITCH]

    providers:
        in_memory:
            memory:
                users:
                    user:  { password: userpass, roles: [ 'ROLE_USER' ] }
                    admin: { password: adminpass, roles: [ 'ROLE_ADMIN' ] }

    firewalls:
        dev:
            pattern:  ^/(_(profiler|wdt)|css|images|js)/
            security: false

        login:
            pattern:  ^/demo/secured/login$
            security: false

        secured_area:
            pattern:    ^/demo/secured/
            form_login:
                check_path: /demo/secured/login_check
                login_path: /demo/secured/login
            logout:
                path:   /demo/secured/logout
                target: /demo/
```

Essa configuração requer que os usuários se autentiquem para acessar qualquer URL começada por `/demo/secured/` e define dois usuários válidos: `user` e `admin`. Além disso o usuário `admin` tem uma permissão `ROLE_ADMIN`, que também inclui a permissão `ROLE_USER` (veja a configuração `role_hierarchy`).

Dica: Para melhorar a legibilidade, nessa nossa configuração simplificada as senhas são guardadas em texto puro, mas você pode usar algum algoritmo de hash ajustando a seção `encoders`.

Indo para a URL `http://localhost/app_dev.php/demo/secured/hello` você será automaticamente redirecionado para o formulário de login pois o recurso é protegido por um `firewall`.

Você também pode forçar a action para requisitar uma permissão especial usando a annotation `@Secure` no controller:

```
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Template;
use JMS\SecurityExtraBundle\Annotation\Secure;

/**
 * @Route("/hello/admin/{name}", name="_demo_secured_hello_admin")
```

```
 * @Secure(roles="ROLE_ADMIN")
 * @Template()
 */
public function helloAdminAction($name)
{
    return array('name' => $name);
}
```

Agora, se autentique como `user` (que não tem a permissão `ROLE_ADMIN`) e, a partir da página protegida hello, clique no link "Hello resource secured". O Symfony2 deve retornar um erro HTTP 403, indicando que o usuário está "proibido" de acessar o recurso.

Nota: A camada de segurança do Symfony2 é bem flexível e vem com muitos serviços de usuários (como no Doctrine ORM) e autenticação (como HTTP básico, HTTP digest ou certificados X509). Leia o capítulo "Segurança" do livro para mais informação de como usá-los e configurá-los.

Fazendo Cache dos Recursos

A medida que seu site começa a ter mais tráfego, você vai querer evitar fazer a geração dos mesmos recursos várias e várias vezes. O Symfony2 usa cabeçalhos de cache HTTP para gerenciar o cache dos recursos. Para estratégias simples de cache, use a annotation conveniente `@Cache()`:

```
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Template;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Cache;

/**
 * @Route("/hello/{name}", name="_demo_hello")
 * @Template()
 * @Cache(maxage="86400")
 */
public function helloAction($name)
{
    return array('name' => $name);
}
```

Nesse exemplo, o recurso ficará em cache por um dia. Mas você também pode usar validações em vez de expiração, ou uma combinação de ambos, se isso se encaixar melhor nas suas necessidades.

O cache de recursos é gerenciado pelo proxy reverso embutido no Symfony2. Mas como o cache é gerenciado usando cabeçalhos de cache HTTP normais, você pode substituir o proxy reverso com o Varnish ou o Squid e estender a sua aplicação de forma fácil.

Nota: Mas como se virar se você não puder fazer cache de páginas inteiras? O Symfony2 continua tendo a solução, via Edge Side Includes (ESI), que são suportados nativamente. Aprenda mais sobre isso lendo o capítulo "HTTP Cache" do livro.

Considerações Finais

Isso foi tudo, e acho que não gastamos nem 10 minutos. Fizemos uma breve introdução aos bundles na primeira parte e todas as funcionalidades sobre as quais aprendemos até agora são parte do bundle núcleo do framework. Graças aos bundles, tudo no Symfony2 pode ser estendido ou substituído. Esse é o tema da próxima parte do tutorial.

1.1.4 A Arquitetura

Você é meu herói! Quem imaginaria que você ainda estaria aqui após as três primeiras partes? Seus esforços serão bem recompensados em breve. As três primeiras partes não contemplaram profundamente a arquitetura do framework. Porque ela faz o Symfony2 destacar-se na multidão de frameworks, vamos mergulhar na arquitetura agora.

Compreendendo a estrutura de diretório

A estrutura de diretório de uma aplicação do Symfony2 é bastante flexível, mas a estrutura do diretório da distribuição *Standard Edition* reflete a estrutura típica e recomendada de uma aplicação Symfony2:

- `app/`: A configuração da aplicação;
- `src/`: O código PHP do projeto;
- `vendor/`: As dependências de terceiros;
- `web/`: O diretório raiz web.

O Diretório `web/`

O diretório raiz web é o local de todos os arquivos públicos e estáticos, como imagens, folhas de estilo e arquivos JavaScript. É também o local onde cada front controller reside:

```
// web/app.php
require_once __DIR__.'/../app/bootstrap.php.cache';
require_once __DIR__.'/../app/AppKernel.php';

use Symfony\Component\HttpFoundation\Request;

$kernel = new AppKernel('prod', false);
$kernel->loadClassCache();
$kernel->handle(Request::createFromGlobals())->send();
```

O kernel primeiro solicita o arquivo `bootstrap.php.cache`, que inicializa a estrutura e regista o autoloader (veja abaixo).

Como qualquer front controller, o `app.php` usa uma classe Kernel, `AppKernel`, para a inicialização da aplicação.

O Diretório `app/`

A classe `AppKernel` é o principal ponto de entrada da configuração da aplicação e, como tal, ele é armazenado no diretório `app/`.

Essa classe deve implementar dois métodos:

- `registerBundles()` que deve retornar um array de todos os bundles necessários para executar a aplicação.
- `registerContainerConfiguration()` que carrega a configuração da aplicação (veremos mais sobre isso depois).

O autoloading do PHP pode ser configurado via `app/autoload.php`:

```
// app/autoload.php
use Symfony\Component\ClassLoader\UniversalClassLoader;

$loader = new UniversalClassLoader();
$loader->registerNamespaces(array(
```

```
    'Symfony'          => array(__DIR__.'/../vendor/symfony/symfony/src', __DIR__.'/../vendor/bundles
    'Sensio'           => __DIR__.'/../vendor/bundles',
    'JMS'              => __DIR__.'/../vendor/jms/',
    'Doctrine\\Common' => __DIR__.'/../vendor/doctrine/common/lib',
    'Doctrine\\DBAL'   => __DIR__.'/../vendor/doctrine/dbal/lib',
    'Doctrine'         => __DIR__.'/../vendor/doctrine/orm/lib',
    'Monolog'          => __DIR__.'/../vendor/monolog/monolog/src',
    'Assetic'          => __DIR__.'/../vendor/kriswallsmith/assetic/src',
    'Metadata'         => __DIR__.'/../vendor/jms/metadata/src',
));
$loader->registerPrefixes(array(
    'Twig_Extensions_' => __DIR__.'/../vendor/twig/extensions/lib',
    'Twig_'            => __DIR__.'/../vendor/twig/twig/lib',
));

// ...

$loader->registerNamespaceFallbacks(array(
    __DIR__.'/../src',
));
$loader->register();
```

O `UniversalClassLoader` é usado para fazer o autoload dos arquivos que respeitam as normas técnicas de interoperabilidade para namespaces do PHP 5.3 ou a convenção de nomenclatura para classes do PEAR. Como você pode ver aqui, todas as dependências são armazenadas sob o diretório `vendor/`, mas isso é apenas uma convenção. Você pode armazená-las onde quiser, globalmente em seu servidor ou localmente em seus projetos.

Nota: Se você quiser saber mais sobre a flexibilidade do autoloader do Symfony2, leia o capítulo "O Componente ClassLoader".

Compreendendo o Sistema dos Bundles

Esta seção apresenta um dos maiores e mais poderosos recursos do Symfony2, o sistema de bundle.

Um bundle é como um plugin em outro software. Então por que ele é chamado de *bundle* de não de *plugin*? Porque *tudo* é um bundle no Symfony2, desde as funcionalidades do núcleo do framework até o código que você escreve para a sua aplicação. Os bundles são cidadãos de primeira classe no Symfony2. Isso lhe fornece a flexibilidade de usar funcionalidades pré-construídas que vêm em bundles de terceiros ou distribuir os seus próprios bundles. Isso torna mais fácil a tarefa de escolher quais recursos que serão habilitados na sua aplicação e otimizá-los da maneira que desejar. E, no final do dia, o código da sua aplicação é tão *importante* quanto o próprio framework.

Registrando um Bundle

Uma aplicação é composta de bundles, que foram definidos no método `registerBundles()` da classe `AppKernel`. Cada bundle é um diretório que contém uma única classe `Bundle` que descreve ele:

```
// app/AppKernel.php
public function registerBundles()
{
    $bundles = array(
        new Symfony\Bundle\FrameworkBundle\FrameworkBundle(),
        new Symfony\Bundle\SecurityBundle\SecurityBundle(),
        new Symfony\Bundle\TwigBundle\TwigBundle(),
        new Symfony\Bundle\MonologBundle\MonologBundle(),
```

```
        new Symfony\Bundle\SwiftmailerBundle\SwiftmailerBundle(),
        new Symfony\Bundle\DoctrineBundle\DoctrineBundle(),
        new Symfony\Bundle\AsseticBundle\AsseticBundle(),
        new Sensio\Bundle\FrameworkExtraBundle\SensioFrameworkExtraBundle(),
        new JMS\SecurityExtraBundle\JMSSecurityExtraBundle(),
    );

    if (in_array($this->getEnvironment(), array('dev', 'test'))) {
        $bundles[] = new Acme\DemoBundle\AcmeDemoBundle();
        $bundles[] = new Symfony\Bundle\WebProfilerBundle\WebProfilerBundle();
        $bundles[] = new Sensio\Bundle\DistributionBundle\SensioDistributionBundle();
        $bundles[] = new Sensio\Bundle\GeneratorBundle\SensioGeneratorBundle();
    }

    return $bundles;
}
```

Além do `AcmeDemoBundle` que nós já falamos, observe que o kernel também habilita outros bundles, como o `FrameworkBundle`, `DoctrineBundle`, `SwiftmailerBundle` e o `AsseticBundle`. Todos eles fazem parte do framework.

Configurando um Bundle

Cada bundle pode ser personalizado através dos arquivos de configuração escritos em YAML, XML ou PHP. Esta é a configuração padrão:

```
# app/config/config.yml
imports:
    - { resource: parameters.yml }
    - { resource: security.yml }

framework:
    #esi:             ~
    #translator:      { fallback: "%locale%" }
    secret:          "%secret%"
    router:          { resource: "%kernel.root_dir%/config/routing.yml" }
    form:            true
    csrf_protection: true
    validation:      { enable_annotations: true }
    templating:      { engines: ['twig'] } #assets_version: SomeVersionScheme
    default_locale:  "%locale%"
    session:
        auto_start:     true

# Twig Configuration
twig:
    debug:            "%kernel.debug%"
    strict_variables: "%kernel.debug%"

# Assetic Configuration
assetic:
    debug:          "%kernel.debug%"
    use_controller: false
    bundles:        [ ]
    # java: /usr/bin/java
    filters:
        cssrewrite: ~
```

```
        # closure:
        #     jar: "%kernel.root_dir%/java/compiler.jar"
        # yui_css:
        #     jar: "%kernel.root_dir%/java/yuicompressor-2.4.2.jar"

# Doctrine Configuration
doctrine:
    dbal:
        driver:   "%database_driver%"
        host:     "%database_host%"
        port:     "%database_port%"
        dbname:   "%database_name%"
        user:     "%database_user%"
        password: "%database_password%"
        charset:  UTF8

    orm:
        auto_generate_proxy_classes: "%kernel.debug%"
        auto_mapping: true

# Swiftmailer Configuration
swiftmailer:
    transport: "%mailer_transport%"
    host:      "%mailer_host%"
    username:  "%mailer_user%"
    password:  "%mailer_password%"

jms_security_extra:
    secure_controllers:  true
    secure_all_services: false
```

Cada entrada como `framework` define a configuração para um bundle específico. Por exemplo, `framework` configura o `FrameworkBundle` enquanto `swiftmailer` configura o `SwiftmailerBundle`.

Cada ambiente pode substituir a configuração padrão, ao fornecer um arquivo de configuração específico. Por exemplo, o ambiente `dev` carrega o arquivo `config_dev.yml`, que carrega a configuração principal (ou seja, `config.yml`) e, então, modifica ela para adicionar algumas ferramentas de depuração:

```
# app/config/config_dev.yml
imports:
    - { resource: config.yml }

framework:
    router:   { resource: "%kernel.root_dir%/config/routing_dev.yml" }
    profiler: { only_exceptions: false }

web_profiler:
    toolbar: true
    intercept_redirects: false

monolog:
    handlers:
        main:
            type:  stream
            path:  "%kernel.logs_dir%/%kernel.environment%.log"
            level: debug
        firephp:
            type:  firephp
            level: info
```

```
assetic:
    use_controller: true
```

Estendendo um Bundle

Além de ser uma boa forma de organizar e configurar seu código, um bundle pode estender um outro bundle. A herança do bundle permite substituir qualquer bundle existente a fim de personalizar seus controladores, templates ou qualquer um de seus arquivos. Aqui é o onde os nomes lógicos (por exemplo, `@AcmeDemoBundle/Controller/SecuredController.php`) são úteis: eles abstraem onde o recurso é realmente armazenado.

Nomes Lógicos de Arquivos Quando você quer fazer referência à um arquivo de um bundle, use esta notação: `@BUNDLE_NAME/path/to/file`; o Symfony2 irá resolver `@BUNDLE_NAME` para o caminho real do bundle. Por exemplo, o caminho lógico `@AcmeDemoBundle/Controller/DemoController.php` seria convertido para `src/Acme/DemoBundle/Controller/DemoController.php`, pois o Symfony conhece a localização do `AcmeDemoBundle`.

Nomes Lógicos de Controladores Para os controladores, você precisa referenciar os nomes de métodos usando o formato `BUNDLE_NAME:CONTROLLER_NAME:ACTION_NAME`. Por exemplo, `AcmeDemoBundle:Welcome:index` mapeia para o método `indexAction` da classe `Acme\DemoBundle\Controller\WelcomeController`.

Nomes Lógicos de Templates Para os templates, o nome lógico `AcmeDemoBundle:Welcome:index.html.twig` é convertido para o caminho de arquivo `src/Acme/DemoBundle/Resources/views/Welcome/index.html.twig`. Os templates tornam-se ainda mais interessantes quando você percebe que eles não precisam ser armazenados no sistema de arquivos. Você pode facilmente armazená-los em uma tabela do banco de dados, por exemplo.

Estendendo Bundles Se você seguir estas convenções, então você pode usar bundle inheritance para "sobrescrever" os arquivos, controladores ou templates. Por exemplo, você pode criar um bundle - `AcmeNewBundle` - e especificar que ele sobrescreve o `AcmeDemoBundle`. Quando o Symfony carregar o controlador `AcmeDemoBundle:Welcome:index`, ele irá primeiro verificar a classe `WelcomeController` em `AcmeNewBundle` e, se ela não existir, então irá verificar o `AcmeDemoBundle`. Isto significa que um bundle pode sobrescrever quase qualquer parte de outro bundle!

Você entende agora porque o Symfony2 é tão flexível? Compartilhe os seus bundles entre aplicações, armazene-os localmente ou globalmente, a escolha é sua.

Usando os Vendors

São grandes as probabilidades de que a sua aplicação dependerá de bibliotecas de terceiros. Estas devem ser armazenadas no diretório `vendor/`. Este diretório já contém as bibliotecas do Symfony2, a biblioteca do SwiftMailer, o ORM Doctrine, o sistema de template Twig e algumas outras bibliotecas e bundles de terceiros.

Entendendo o Cache e Logs

O Symfony2 é provavelmente um dos mais rápidos frameworks full-stack atualmente. Mas como pode ser tão rápido se ele analisa e interpreta dezenas de arquivos YAML e XML para cada pedido? A velocidade é, em parte, devido ao seu sistema de cache. A configuração da aplicação é analisada somente no primeiro pedido e depois compilada em código PHP comum, que é armazenado no diretório `app/cache/`. No ambiente de desenvolvimento, o Symfony2

é inteligente o suficiente para liberar o cache quando você altera um arquivo. Mas, no ambiente de produção, é sua a responsabilidade de limpar o cache quando você atualizar o seu código ou alterar sua configuração.

Ao desenvolver uma aplicação web, as coisas podem dar errado em muitos aspectos. Os arquivos de log no diretório `app/logs/` dizem tudo sobre os pedidos e ajudam a resolver os problemas rapidamente.

Utilizando a Interface da Linha de Comando

Cada aplicação vem com uma ferramenta de interface de linha de comando (`app/console`) que ajuda na manutenção da sua aplicação. Ela fornece comandos que aumentam a sua produtividade ao automatizar tarefas tediosas e repetitivas.

Execute-a sem argumentos para saber mais sobre suas capacidades:

```
$ php app/console
```

A opção `--help` ajuda a descobrir o uso de um comando:

```
$ php app/console router:debug --help
```

Considerações finais

Me chame de louco, mas, depois de ler esta parte, você deve estar confortável em mover as coisas e fazer o Symfony2 trabalhar para você. Tudo no Symfony2 é projetado para sair do seu caminho. Portanto, sinta-se livre para renomear e mover os diretórios como você desejar.

E isso é tudo para o início rápido. Desde testes até o envio de e-mails, você ainda precisa aprender muito para se tornar um mestre no Symfony2. Pronto para aprofundar nestes tópicos agora? Não procure mais - vá para o Livro oficial e escolha qualquer tema que você desejar.

- Panorama Geral >
- A View >
- O Controller >
- A Arquitetura

CAPÍTULO 2

Livro

Mergulhe no Symfony2 com os tópicos guia:

2.1 Livro

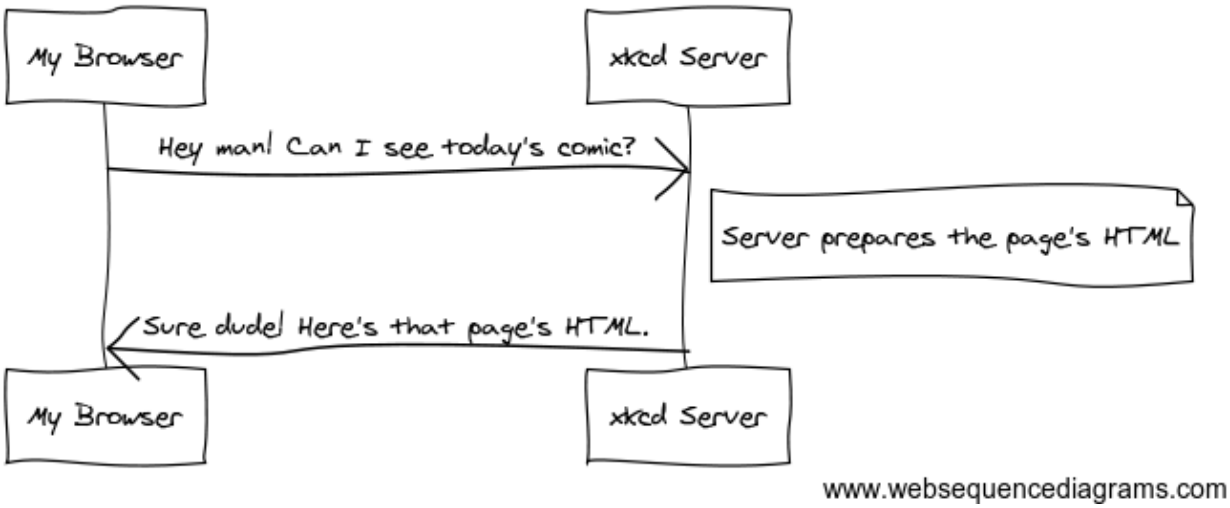
2.1.1 Fundamentos de Symfony e HTTP

Parabéns! Aprendendo sobre o Symfony2, você está no caminho certo para ser um desenvolvedor web mais *produtivo*, *bem preparado* e *popular* (na verdade, este último é por sua própria conta). O Symfony2 foi criado para voltar ao básico: desenvolver ferramentas para ajuda-lo a criar aplicações mais robustas de uma maneira mais rápida sem ficar no seu caminho. Ele foi construído baseando-se nas melhores ideias de diversas tecnologias: as ferramentas e conceitos que você está prestes a aprender representam o esforço de milhares de pessoas, realizado durante muitos anos. Em outras palavras, você não está apenas aprendendo o "Symfony", você está aprendendo os fundamentos da web, boas práticas de desenvolvimento e como usar diversas biblotecas PHP impressionantes, dentro e fora do Symfony2. Então, prepare-se.

Seguindo a filosofia do Symfony2, este capítulo começa explicando o conceito fundamental para o desenvolvimento web: o HTTP. Independente do seu conhecimento anterior ou linguagem de programação preferida, esse capítulo é uma **leitura obrigatória** para todos.

HTTP é simples

HTTP (Hypertext Transfer Protocol, para os geeks) é uma linguagem textual que permite que duas máquinas se comuniquem entre si. É só isso! Por exemplo, quando você vai ler a última tirinha do xkcd, acontece mais ou menos a seguinte conversa:



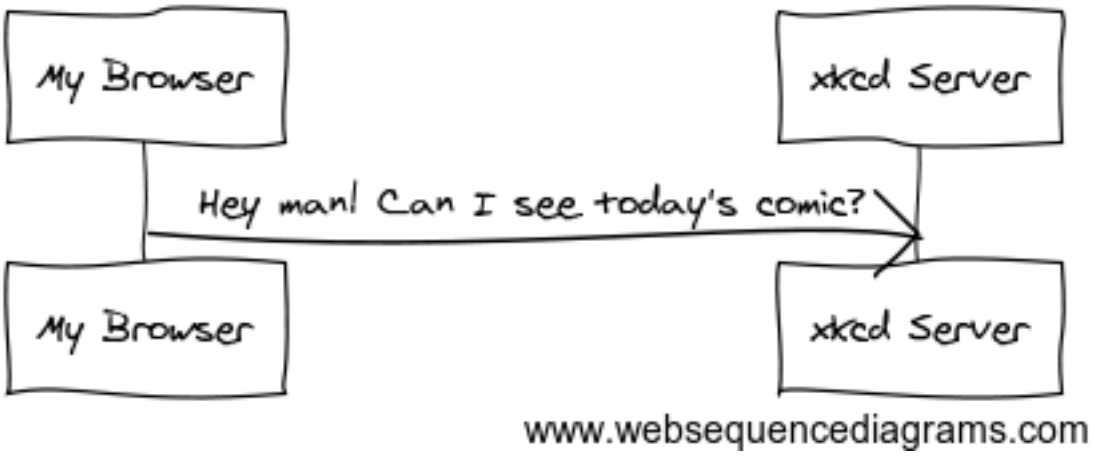
Apesar da linguagem real ser um pouco mais formal, ainda assim ela é bastante simples. HTTP é o termo usado para descrever essa linguagem simples baseada em texto. Não importa como você desenvolva para a web, o objetivo do seu servidor *sempre* será entender simples requisições de texto e enviar simples respostas de texto.

O Symfony2 foi criado fundamentado nessa realidade. Você pode até não perceber, mas o HTTP é algo que você utilizada todos os dias. Com o Symfony2 você irá aprender a domina-lo.

Primeiro Passo: O Cliente envia uma Requisição

Toda comunicação na web começa com uma *requisição*. Ela é uma mensagem de texto criada por um cliente (por exemplo, um navegador, um app para iPhone etc) em um formato especial conhecido como HTTP. O cliente envia essa requisição para um servidor e, então, espera pela resposta.

Veja a primeira parte da interação (a requisição) entre um navegador e o servidor web do xkcd:



No linguajar do HTTP, essa requisição se parece com isso:

```
GET / HTTP/1.1
Host: xkcd.com
Accept: text/html
User-Agent: Mozilla/5.0 (Macintosh)
```

Essa simples mensagem comunica *tudo* o que é necessário sobre o recurso exato que o cliente está requisitando. A primeira linha de uma requisição HTTP é a mais importante e contém duas coisas: a URI e o método HTTP.

A URI (por exemplo, `/`, `/contact` etc) é um endereço único ou localização que identifica o recurso que o cliente quer. O método HTTP (por exemplo, `GET`) define o que você quer *fazer* com o recurso. Os métodos HTTP são os *verbos* da requisição e definem algumas maneiras comuns de agir em relação ao recurso:

GET	Recupera o recurso do servidor
POST	Cria um recurso no servidor
PUT	Atualiza um recurso no servidor
DELETE	Exclui um recurso do servidor

Tendo isso em mente, você pode imaginar como seria uma requisição HTTP para excluir uma postagem específica de um blog, por exemplo:

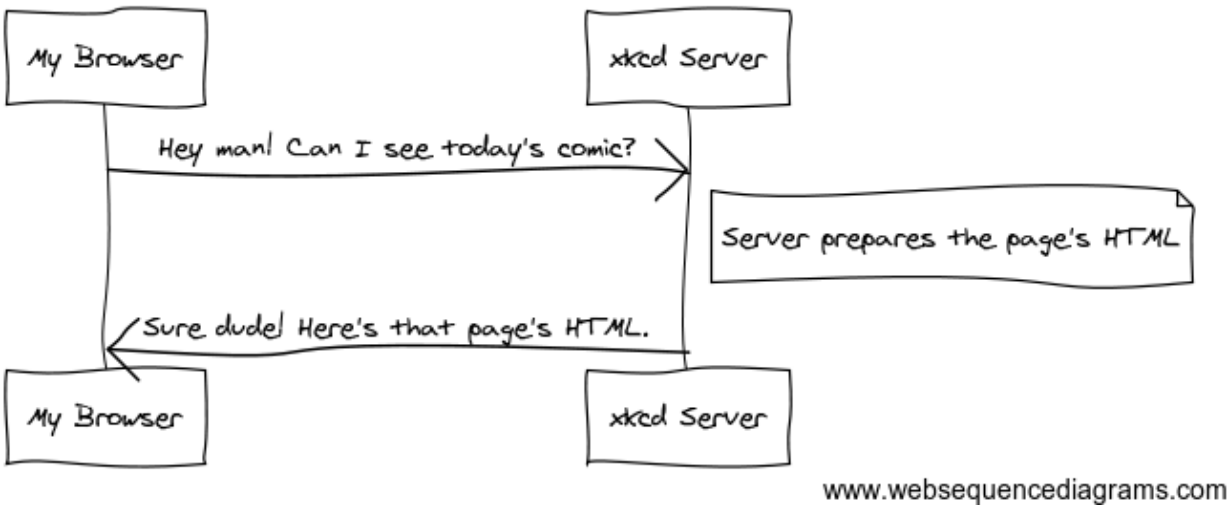
```
DELETE /blog/15 HTTP/1.1
```

Nota: Existem na verdade nove métodos definidos pela especificação HTTP, mas a maioria deles não são muito utilizados ou suportados. Na realidade, muitos dos navegadores modernos não suportam os métodos `PUT` e `DELETE`.

Além da primeira linha, uma requisição HTTP invariavelmente contém outras linhas de informação chamadas de cabeçalhos da requisição. Os cabeçalhos podem fornecer uma vasta quantidade de informações, tais como o `Host` que foi requisitado, os formatos de resposta que o cliente aceita (`Accept`) e a aplicação que o cliente está utilizando para enviar a requisição (`User-Agent`). Muitos outros cabeçalhos existem e podem ser encontrados na Wikipedia, no artigo List of HTTP header fields

Segundo Passo: O Servidor envia uma resposta

Uma vez que o servidor recebeu uma requisição, ele sabe exatamente qual recurso o cliente precisa (através do URI) e o que o cliente quer fazer com ele (através do método). Por exemplo, no caso de uma requisição GET, o servidor prepara o o recurso e o retorna em uma resposta HTTP. Considere a resposta do servidor web do xkcd:



Traduzindo para HTTP, a resposta enviada para o navegador será algo como:

```
HTTP/1.1 200 OK
Date: Sat, 02 Apr 2011 21:05:05 GMT
Server: lighttpd/1.4.19
Content-Type: text/html
```

```
<html>
  <!-- HTML for the xkcd comic -->
</html>
```

A resposta HTTP contém o recurso requisitado (nesse caso, o conteúdo HTML), bem como outras informações. A primeira linha é especialmente importante e contém o código de status da resposta HTTP (nesse caso, 200). Esse código de status é uma representação geral da resposta enviada à requisição do cliente. A requisição foi bem sucedida? Ocorreu algum erro? Existem diferentes códigos de status para indentificar sucesso, um erro, ou que o cliente precisa fazer alguma coisa (por exemplo, redirecionar para outra página). Uma lista completa pode ser encontrada na Wikipedia, no artigo List of HTTP status codes.

Assim como uma requisição, uma resposta HTTP também contém informações adicionais conhecidas como cabeçalhos HTTP. Por exemplo, um cabeçalho importante nas respostas HTTP é o `Content-Type`. O conteúdo de um mesmo recurso pode ser retornado em vários formatos diferentes, incluindo HTML, XML ou JSON, só para citar alguns. O cabeçalho `Content-Type` diz ao cliente qual é o formato que está sendo retornado.

Existem diversos outros cabeçalhos, alguns deles bastante poderosos. Certos cabeçalhos, por exemplo, podem ser utilizados para criar um poderoso sistema de cache.

Requisições, Respostas e o Desenvolvimento Web

Essa conversação de requisição-resposta é o processo fundamental que dirige toda a comunicação na web. Apesar de tão importante e poderoso esse processo, ainda assim, é inevitavelmente simples.

O fato mais importante é: independente da linguagem que você utiliza, o tipo de aplicação que você desenvolva (web, mobile, API em JSON) ou a filosofia de desenvolvimento que você segue, o objetivo final da aplicação **sempre** será entender cada requisição e criar e enviar uma resposta apropriada.

O Symfony foi arquitetado para atender essa realidade.

Dica: Para aprender mais sobre a especificação HTTP, leia o original HTTP 1.1 RFC ou HTTP Bis, que trata-se de um esforço para facilitar o entendimento da especificação original. Para verificar as requisições e respostas enviadas enquanto navega em um site, você pode utilizar a extensão do Firefox chamada Live HTTP Headers.

Requisições e Respostas no PHP

Como interagir com a "requisição" e criar uma "resposta" utilizando o PHP? Na verdade, o PHP abstrai um pouco desse processo:

```
<?php
$uri = $_SERVER['REQUEST_URI'];
$foo = $_GET['foo'];

header('Content-type: text/html');
echo 'The URI requested is: '.$uri;
echo 'The value of the "foo" parameter is: '.$foo;
```

Por mais estranho que possa parecer, essa pequena aplicação, de fato, lê informações da requisição HTTP e a está utilizando para criar um resposta HTTP. Em vez de interpretar a requisição pura, o PHP prepara algumas variáveis superglobais , tais como `$_SERVER` e `$_GET`, que contém toda a informação da requisição. Da mesma forma, em vez de retornar o texto da resposta no formato do HTTP, você pode utilizar a função `header()` para criar os cabeçalhos e simplesmente imprimir o que será o conteúdo da mensagem da reposta. O PHP irá criar uma reposta HTTP verdadeira que será retornada para o cliente.

```
HTTP/1.1 200 OK
Date: Sat, 03 Apr 2011 02:14:33 GMT
Server: Apache/2.2.17 (Unix)
Content-Type: text/html

The URI requested is: /testing?foo=symfony
The value of the "foo" parameter is: symfony
```

Requisições e Respostas no Symfony

O Symfony fornece uma alternativa à abordagem feita com o PHP puro, utilizando duas classes que permitem a interação com as requisições e respostas HTTP de uma maneira mais fácil. A classe `Request` é uma simples representação orientada a objetos de uma requisição HTTP. Com ela, você tem todas as informações da requisição nas pontas dos dedos:

```
use Symfony\Component\HttpFoundation\Request;

$request = Request::createFromGlobals();

// the URI being requested (e.g. /about) minus any query parameters
$request->getPathInfo();

// retrieve GET and POST variables respectively
$request->query->get('foo');
$request->request->get('bar');

// retrieves an instance of UploadedFile identified by foo
$request->files->get('foo');

$request->getMethod();          // GET, POST, PUT, DELETE, HEAD
$request->getLanguages();       // an array of languages the client accepts
```

Como um bônus, a classe `Request` faz um monte de trabalho com o qual você nunca precisará se preocupar. Por exemplo, o método `isSecure()` verifica os três valores diferentes que o PHP utiliza para indicar ser o usuário está utilizando uma conexão segura (`https`, por exemplo).

O Symfony também fornece a classe `Response`: uma simples representação em PHP de uma resposta HTTP. Assim é possível que sua aplicação utilize uma interface orientada a objetos para construir a resposta que precisa ser enviada ao cliente:

```
use Symfony\Component\HttpFoundation\Response;
$response = new Response();

$response->setContent('<html><body><h1>Hello world!</h1></body></html>');
$response->setStatusCode(200);
$response->headers->set('Content-Type', 'text/html');

// prints the HTTP headers followed by the content
$response->send();
```

Com tudo isso, mesmo que o Symfony não oferecesse mais nada, você já teria um kit de ferramentas para facilmente acessar informações sobre a requisição e uma interface orientada a objetos para criar a resposta. Mesmo depois de aprender muitos dos poderosos recursos do Symfony, tenha em mente que o objetivo da sua aplicação sempre será *interpretar uma requisição e criar a resposta apropriada baseada na lógica da sua aplicação*.

Dica: As classes `Request` e `Response` fazem parte de um componente do Symfony chamado

`HttpFoundation`. Esse componente pode ser utilizado de forma independente ao framework e também possui classes para tratar sessões e upload de arquivos.

A Jornada da Requisição até a Resposta

Como o próprio HTTP, os objetos `Request` e `Response` são bastante simples. A parte difícil de se construir uma aplicação é escrever o que acontece entre eles. Em outras palavras, o trabalho de verdade é escrever o código que interpreta a a requisição e cria a resposta.

A sua aplicação provavelmente faz muitas coisas como enviar emails, tratar do envio de formulários, salvar coisas no banco de dados, renderizar páginas HTML e proteger o conteúdo com segurança. Como cuidar de tudo isso e ainda ter um código organizado e de fácil manutenção?

O Symfony foi criado para que ele resolva esses problemas, não você.

O Front Controller

Tradicionalmente, aplicações são construídas para que cada página do site seja um arquivo físico:

```
index.php
contact.php
blog.php
```

Existem diversos problemas para essa abordagem, incluindo a falta de flexibilidade das URLs (e se você quiser mudar o arquivo `blog.php` para `news.php` sem quebrar todos os seus links?) e o fato de que cada arquivo *deve* ser alterado manualmente para incluir um certo conjunto de arquivos essenciais de forma que a segurança, conexões com banco de dados e a "aparência" do site continue consistente.

Uma solução muito melhor é utilizar um front controller: um único arquivo PHP que trata todas as requisições enviadas para a sua aplicação. Por exemplo:

`/index.php`	executa `index.php`
`/index.php/contact`	executa `index.php`
`/index.php/blog`	executa `index.php`

Dica: Utilizando o `mod_rewrite` do Apache (ou o equivalente em outros servidores web), as URLs podem ser simplificadas facilmente para ser somente `/`, `/contact` e `/blog`.

Agora, cada requisição é tratada exatamente do mesmo jeito. Em vez de arquivos PHP individuais para executar cada URL, o front controller *sempre* será executado, e o roteamento de cada URL para diferentes partes da sua aplicação é feito internamente. Assim resolve-se os dois problemas da abordagem original. Quase todas as aplicações modernas fazem isso - incluindo apps como o Wordpress.

Mantenha-se Organizado

Dentro do front controller, como você sabe qual página deve ser renderizada e como renderiza-las de uma maneira sensata? De um jeito ou de outro, você precisará verificar a URI requisitada e executar partes diferentes do seu código dependendo do seu valor. Isso pode acabar ficando feio bem rápido:

```
// index.php

$request = Request::createFromGlobals();
$path = $request->getPathInfo(); // the URL being requested
```

```
if (in_array($path, array('', '/')) {
    $response = new Response('Welcome to the homepage.');
} elseif ($path == '/contact') {
    $response = new Response('Contact us');
} else {
    $response = new Response('Page not found.', 404);
}
$response->send();
```

Resolver esse problema pode ser difícil. Felizmente é *exatamente* o que o Symfony foi projetado para fazer.

O Fluxo de uma Aplicação Symfony

Quando você deixa que o Symfony cuide de cada requisição, sua vida fica muito mais fácil. O framework segue um simples padrão para toda requisição:

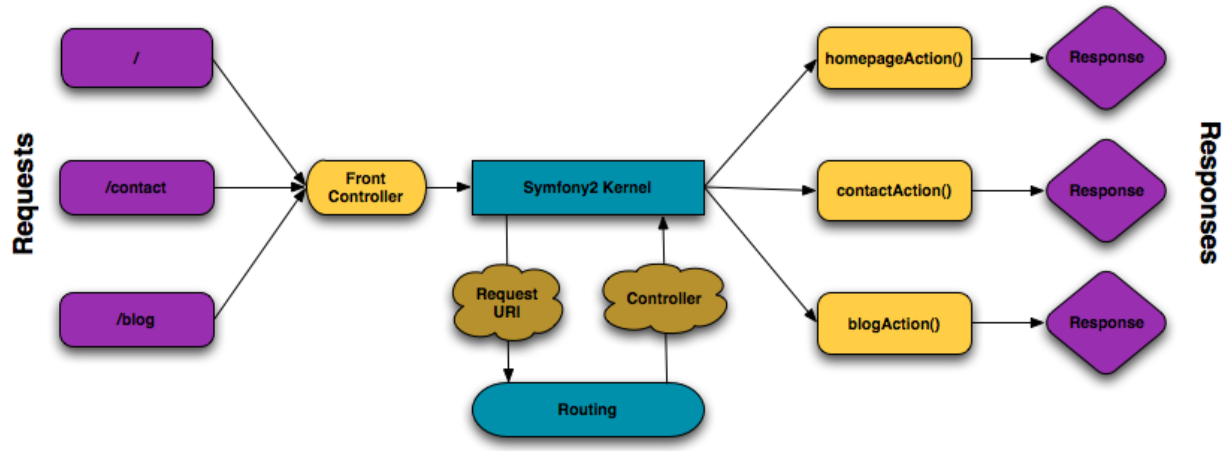


Fig. 2.1: As requisições recebidas são interpretadas pelo roteamento e passadas para as funções controller que retornam objetos do tipo `Response`.

Cada "página" do seu site é definida no arquivo de configuração de roteamento que mapeia diferentes URLs para diferentes funções PHP. O trabalho de cada função, chamadas de controller, é usar a informação da requisição - junto com diversas outras ferramentas disponíveis no Symfony - para criar e retornar um objeto `Response`. Em outras palavras, o *seu* código deve estar nas funções controller: lá é onde você interpreta a requisição e cria uma resposta.

É fácil! Vamos fazer uma revisão:

- Cada requisição executa um arquivo front controller;
- O sistema de roteamento determina qual função PHP deve ser executada, baseado na informação da requisição e na configuração de roteamento que você criou;
- A função PHP correta é executada, onde o seu código cria e retorna o objeto `Response` apropriado.

Uma Requisição Symfony em Ação

Sem entrar em muitos detalhes, vamos ver esse processo em ação. Suponha que você quer adicionar a página `/contact` na sua aplicação Symfony. Primeiro, adicione uma entrada para `/contact` no seu arquivo de configuração de roteamento:

```
contact:
    pattern:  /contact
    defaults: { _controller: AcmeDemoBundle:Main:contact }
```

Nota: Esse exemplo utiliza `YAML` para definir a configuração de roteamento. Essa configuração também pode ser escrita em outros formatos, tais como XML ou PHP.

Quando alguém visitar a página `/contact`, essa rota será encontrada e o controller específico será executado. Como você irá aprender no capítulo sobre roteamento, a string `AcmeDemoBundle:Main:contact` é uma sintaxe encurtada para apontar para o método `contactAction` dentro de uma classe chamada `MainController`:

```
class MainController
{
    public function contactAction()
    {
        return new Response('<h1>Contact us!</h1>');
    }
}
```

Nesse exemplo extremamente simples, o controller simplesmente cria um objeto `Response` com o HTML "<h1>Contact us!</h1>". No capítulo sobre controller, você irá aprender como um controller pode renderizar templates, fazendo com que o seu código de "apresentação" (por exemplo, qualquer coisa que gere HTML) fique em um arquivo de template separado. Assim deixamos o controller livre para se preocupar apenas com a parte complicada: interagir com o banco de dados, tratar os dados enviados ou enviar emails.

Symfony2: Construa sua aplicação, não suas Ferramentas

Agora você sabe que o objetivo de qualquer aplicação é interpretar cada requisição recebida e criar uma resposta apropriada. Conforme uma aplicação cresce, torna-se mais difícil de manter o seu código organizado e de fácil manutenção. Invariavelmente, as mesmas tarefas complexas continuam a aparecer: persistir dados no banco, renderizar e reutilizar templates, tratar envios de formulários, enviar emails, validar entradas dos usuários e cuidar da segurança.

A boa notícia é que nenhum desses problemas é único. O Symfony é um framework cheio de ferramentas para você construir a sua aplicação e não as suas ferramentas. Com o Symfony2, nada é imposto: você é livre para utilizar o framework completo ou apenas uma parte dele.

Ferramentas Independentes: Os *Componentes* do Symfony2

Então, o que *é* o Symfony2? Primeiramente, trata-se de uma coleção de vinte bibliotecas independentes que podem ser utilizadas dentro de *qualquer* projeto PHP. Essas bibliotecas, chamadas de *Components do Symfony2*, contém coisas úteis para praticamente qualquer situação, independente de como o seu projeto é desenvolvido. Alguns desses componentes são:

- HttpFoundation - Contém as classes `Request` e `Response`, bem como outras classes para tratar de sessões e upload de arquivos;
- Routing - Um poderoso e rápido sistema de roteamento que permite mapear uma URI específica (por exemplo, `/contact`) para uma informação sobre como a requisição deve ser tratada (por exemplo, executar o método `contactAction()`);
- Form - Um framework completo e flexível para criar formulários e tratar os dados enviados por eles;
- Validator Um sistema para criar regras sobre dados e validar se os dados enviados pelos usuários seguem ou não essas regras;

- ClassLoader Uma biblioteca de autoloading que faz com que classes PHP possam ser utilizadas sem precisar adicionar manualmente um `require` para cada arquivo que as contém;
- Templating Um conjunto de ferramentas para renderizar templates, tratar da herança de templates (por exemplo, um template decorado com um layout) e executar outras tarefas comuns relacionadas a templates;
- Security - Uma biblioteca poderosa para tratar qualquer tipo de segurança dentro de sua aplicação;
- Translation Um framework para traduzir strings na sua aplicação.

Cada um desses componentes funcionam de forma independente e podem ser utilizados em *qualquer* projeto PHP, não importa se você utiliza o Symfony2 ou não. Cada parte foi feita para ser utilizada e substituída quando for necessário.

A solução completa: O *framework* Symfony2

Então, o que *é* o *framework* Symfony2? Ele é uma biblioteca PHP que realiza duas tarefas distintas:

1. Fornecer uma seleção de componentes (os componentes do Symfony2, por exemplo) e bibliotecas de terceiros (por exemplo, a `Swiftmailer`, utilizada para enviar emails);
2. Fornecer as configurações necessárias e uma "cola" para manter todas as peças juntas.

O objetivo do framework é integrar várias ferramentas independentes para criar uma experiência consistente para o desenvolvedor. Até próprio próprio framework é um pacote Symfony2 (um plugin, por exemplo) que pode ser configurado ou completamente substituído.

O Symfony2 fornece um poderoso conjunto de ferramentas para desenvolver aplicações web rapidamente sem impor nada. Usuários normais podem iniciar o desenvolvimento rapidamente utilizando uma distribuição do Symfony2, que contém o esqueleto de um projeto com as princpais itens padrão. Para os usuários mais avançados, o céu é o limite.

2.1.2 Symfony2 versus o PHP puro

Por que usar o Symfony2 é melhor do que abrir um arquivo e sair escrevendo PHP puro?

Se você nunca utilizou um framework PHP, não está familiarizado com a filosofia MVC ou está apenas interessado em entender todo esse *hype* sobre o Symfony2, este capítulo é para você. Em vez de *dizer* que o Symfony2 permite que você desenvolva mais rápido e melhor do que com PHP puro, você vai ver por si mesmo.

Nesse capítulo, você irá escrever uma simples aplicação em PHP puro, e, então, refatora-la para deixa-la mais organizada. Você vai viajar no tempo, vendo as decisões sobre o porquê o desenvolvimento web evoluiu com o passar dos tempos para onde ele está agora.

Ao final, você verá como o Symfony2 pode resgata-lo das tarefas simples e coloca-lo de volta no controle do seu código.

Um simples Blog em PHP puro

Nesse capítulo, você vai construir uma aplicação para um blog utilizando apenas o PHP puro. Para começar, crie uma única página que exibe as postagens armazenadas no banco de dados. Escrever isso em PHP puro é rápido e simples:

```
<?php
// index.php

$link = mysql_connect('localhost', 'myuser', 'mypassword');
mysql_select_db('blog_db', $link);

$result = mysql_query('SELECT id, title FROM post', $link);
?>
```

```
<html>
    <head>
        <title>List of Posts</title>
    </head>
    <body>
        <h1>List of Posts</h1>
        <ul>
            <?php while ($row = mysql_fetch_assoc($result)): ?>
            <li>
                <a href="/show.php?id=<?php echo $row['id'] ?>">
                    <?php echo $row['title'] ?>
                </a>
            </li>
            <?php endwhile; ?>
        </ul>
    </body>
</html>

<?php
mysql_close($link);
```

Simples de escrever, rápido de executar e, conforme sua aplicação crescer, impossível de manter. Existem diversos problemas que precisam ser tratados:

- **Sem verificações de erros**: E se a conexão com o banco de dados falhar?
- **Organização pobre**: Se a aplicação crescer, esse arquivo também irá crescer e ficará impossível de dar manutenção. Onde você deve colocar o código que cuida de tratar os envios de formulários? Como você valida os dados? Onde você deve colocar o código que envia emails?
- **Dificuldade para reutilizar código**: Uma vez que tudo está em um único arquivo, não há como reutilizar qualquer parte dele em outras "páginas" do blog.

Nota: Um outro problema não mencionado aqui é o fato do banco de dados estar amarrado ao MySQL. Apesar de não ser tratado aqui, o Symfony2 integra-se totalmente com o Doctrine, uma biblioteca dedicada a abstração de banco de dados e mapeamento.

Vamos ao trabalho de resolver esses problemas e mais ainda.

Isolando a Apresentação

O código pode ter ganhos imediatos ao separar a "lógica" da aplicação do código que prepara o HTML para "apresentação":

```
<?php
// index.php

$link = mysql_connect('localhost', 'myuser', 'mypassword');
mysql_select_db('blog_db', $link);

$result = mysql_query('SELECT id, title FROM post', $link);

$posts = array();
while ($row = mysql_fetch_assoc($result)) {
    $posts[] = $row;
}
```

```
mysql_close($link);

// include the HTML presentation code
require 'templates/list.php';
```

Agora o código HTML está armazenado em um arquivo separado (`templates/list.php`), que é um arquivo HTML que utiliza um sintaxe PHP parecida com a de templates:

```
<html>
    <head>
        <title>List of Posts</title>
    </head>
    <body>
        <h1>List of Posts</h1>
        <ul>
            <?php foreach ($posts as $post): ?>
            <li>
                <a href="/read?id=<?php echo $post['id'] ?>">
                    <?php echo $post['title'] ?>
                </a>
            </li>
            <?php endforeach; ?>
        </ul>
    </body>
</html>
```

Por convenção, o arquivo que contém toda a lógica da aplicação - `index.php` - é conhecido como "controller". O termo controller é uma palavra que você vai escutar bastante, independente da linguagem ou framework você utilize. Ela refere-se a área do *seu* código que processa as entradas do usuário e prepara uma resposta.

Nesse caso, nosso controller prepara os dados do banco de dados e então inclui um template para apresenta-los. Com o controller isolado, você pode facilmente mudar *apenas* o arquivo de template caso precise renderizar os posts de blog em algum outro formato (por exemplo, `list.json.php` para o formato JSON).

Isolando a Lógica (Domínio) da Aplicacão

Por enquanto a aplicação tem apenas uma página. Mas e se uma segunda página precisar utilizar a mesma conexão com o banco de dados, ou até o mesmo array de posts do blog? Refatore o código de forma que o comportamento principal e as funções de acesso aos dados da aplicação fiquem isolados em um novo arquivo chamado `model.php`:

```
<?php
// model.php

function open_database_connection()
{
    $link = mysql_connect('localhost', 'myuser', 'mypassword');
    mysql_select_db('blog_db', $link);

    return $link;
}

function close_database_connection($link)
{
    mysql_close($link);
}

function get_all_posts()
```

```
{
    $link = open_database_connection();

    $result = mysql_query('SELECT id, title FROM post', $link);
    $posts = array();
    while ($row = mysql_fetch_assoc($result)) {
        $posts[] = $row;
    }
    close_database_connection($link);

    return $posts;
}
```

Dica: O nome `model.php` foi utilizado porque a lógica e o acesso aos dados de uma aplicação são tradicionalmente conhecidos como a camada de "modelo". Em uma aplicação bem organizada, a maioria do código representando as suas "regras de negócio" devem estar apenas no model (em vez de estar em um controller). Ao contrário desse exemplo, somente uma parte do model (ou nenhuma) está realmente relacionada ao banco de dados.

Agora o controller (`index.php`) ficou bem simples:

```
<?php
require_once 'model.php';

$posts = get_all_posts();

require 'templates/list.php';
```

Agora, a única tarefa do controller é recuperar os dados da camada de modelo da sua aplicação (o model) e chamar o template para renderiza-los. Esse é um exemplo bem simples do padrão model-view-controller.

Isolando o Layout

Até esse ponto a aplicação foi refatorada em três partes distintas, oferecendo várias vantagens e a oportunidade de reutilizar quase qualquer coisa em outras páginas.

A única parte do código que *não pode* ser reutilizada é o layout da página. Conserte isso criando um novo arquivo chamado `layout.php`:

```
<!-- templates/layout.php -->
<html>
    <head>
        <title><?php echo $title ?></title>
    </head>
    <body>
        <?php echo $content ?>
    </body>
</html>
```

Assim o template (`templates/list.php`) pode ficar mais simples "extendendo" o layout:

```
<?php $title = 'List of Posts' ?>

<?php ob_start() ?>
    <h1>List of Posts</h1>
    <ul>
        <?php foreach ($posts as $post): ?>
```

```
        <li>
            <a href="/read?id=<?php echo $post['id'] ?>">
                <?php echo $post['title'] ?>
            </a>
        </li>
        <?php endforeach; ?>
    </ul>
<?php $content = ob_get_clean() ?>

<?php include 'layout.php' ?>
```

Agora você foi apresentado a uma metodologia que permite a reutilização do layout. Infelizmente, para fazer isso, você é forçado a utilizar no template algumas funções feias do PHP (`ob_start()`, `ob_get_clean()`). O Symfony2 utiliza o componente `Templating` que permite realizar isso de uma maneira limpa e fácil. Logo você verá esse componente em ação.

Adicionando a página "show" ao Blog

A página "list" foi refatorada para que o código fique mais organizado e reutilizável. Para provar isso, adicione ao blog uma página chamada "show", que exibe um único post identificado pelo parâmetro `id`.

Para começar, crie uma nova função no arquivo `model.php` que recupera o post com base no id informado:

```
// model.php
function get_post_by_id($id)
{
    $link = open_database_connection();

    $id = mysql_real_escape_string($id);
    $query = 'SELECT date, title, body FROM post WHERE id = '.$id;
    $result = mysql_query($query);
    $row = mysql_fetch_assoc($result);

    close_database_connection($link);

    return $row;
}
```

Em seguida, crie um novo arquivo chamado `show.php` - o controller para essa nova página:

```
<?php
require_once 'model.php';

$post = get_post_by_id($_GET['id']);

require 'templates/show.php';
```

Por fim, crie um novo arquivo de template - `templates/show.php` - para renderizar individualmente o post do blog:

```
<?php $title = $post['title'] ?>

<?php ob_start() ?>
    <h1><?php echo $post['title'] ?></h1>

    <div class="date"><?php echo $post['date'] ?></div>
    <div class="body">
        <?php echo $post['body'] ?>
```

```
    </div>
<?php $content = ob_get_clean() ?>

<?php include 'layout.php' ?>
```

Criar a segunda página foi bastante fácil e nenhum código foi duplicado. Ainda assim, essa página criou mais alguns problemas persistentes que um framework pode resolver para você. Por exemplo, se o parâmetro `id` não for informado, ou for inválido, a página irá quebrar. Seria mais interessante exibir uma página de erro 404, mas isso ainda não pode ser feito de uma maneira fácil. Pior ainda, caso você esqueça de tratar o `id` utilizando a função `mysql_real_escape_string()`, todo o seu banco de dados estará correndo o risco de sofrer ataques de SQL injection.

Um problema ainda maior é que cada controller deve incluir o arquivo `model.php` individualmente. O que acontece se cada controller, de repente, precisar incluir um arquivo adicional para executar alguma outra tarefa global (impor segurança, por exemplo)? Da maneira como está agora, esse código teria que ser adicionado em cada arquivo controller. Se você esquecer de incluir algo em algum arquivo espero que não seja algo relacionado a segurança...

Um "Front Controller" para a salvação

A solução é utilizar um front controller: um único arquivo PHP que irá processar *todas* as requisições. Com um front controller, as URIs vão mudar um pouco, mas começam a ficar mais flexíveis:

```
Without a front controller
/index.php          => Blog post list page (index.php executed)
/show.php           => Blog post show page (show.php executed)

With index.php as the front controller
/index.php          => Blog post list page (index.php executed)
/index.php/show     => Blog post show page (index.php executed)
```

Dica: O `index.php` pode ser removido da URI se você estiver utilizando regras de rewrite no Apache (ou algo equivalente). Nesse caso, a URI resultante para a página show será simplesmente `/show`.

Ao utilizar um front controller, um único arquivo PHP (nesse caso o `index.php`) irá renderizar *todas* as requisições. Para a página show do blog, o endereço `/index.php/show` irá, na verdade, executar o arquivo `index.php`, que agora é responsável por redirecionar as requisições internamente baseado na URI completa. Como você pode ver, um front controller é uma ferramente bastante poderosa.

Criando o Front Controller

Você está prestes a dar um **grande** passo com a sua aplicação. Com um arquivo para gerenciar todas as suas requisições, você pode centralizar coisas como segurança, configurações e roteamento. Nessa aplicação, o arquivo `index.php` deve ser esperto o suficiente para renderizar a página com a lista de posts *ou* a página com um único post baseado na URI da requisição:

```
<?php
// index.php

// load and initialize any global libraries
require_once 'model.php';
require_once 'controllers.php';

// route the request internally
```

```
$uri = $_SERVER['REQUEST_URI'];
if ($uri == '/index.php') {
    list_action();
} elseif ($uri == '/index.php/show' && isset($_GET['id'])) {
    show_action($_GET['id']);
} else {
    header('Status: 404 Not Found');
    echo '<html><body><h1>Page Not Found</h1></body></html>';
}
```

Por questão de organização, ambos os controllers (os antigos arquivos `index.php` e `show.php`) agora são funções e cada uma foi movida para um arquivo separado, chamado `controllers.php`:

```
function list_action()
{
    $posts = get_all_posts();
    require 'templates/list.php';
}

function show_action($id)
{
    $post = get_post_by_id($id);
    require 'templates/show.php';
}
```

Sendo um front controller, `index.php` agora tem um papel inteiramente novo, que inclui carregar as bibliotecas principais e rotear a aplicação de forma que um dos controllers (as funções `list_action()` e `show_action()`) seja chamado. Na verdade, o front controller está começando a ficar bastante parecido com o mecanismo do Symfony2 utilizado para tratar e redirecionar as requisições:

Dica: Uma outra vantagem do front controller é ter URLs flexíveis. Note que a URL para a página que exibe um post no blog pode mudar de `/show` para `/read` alterando o código apenas em um único lugar. Antes, um arquivo teria que ser renomeado. No Symfony2 as URLs podem ser ainda mais flexíveis.

Até agora, a aplicação evoluiu de um único arquivo PHP para para uma estrutura organizada que permite a reutilização de código. Você deve estar mais feliz, mas longe de estar satisfeito. Por exemplo, o sistema de "roteamento" ainda não é consistente e não reconhece que a página de listagem (`index.php`) também pode ser acessada via `/` (se as regras de rewrite foram adicionadas no Apache). Além disso, em vez de desenvolver o blog, boa parte do tempo foi gasto trabalhando na "arquitetura" do código (por exemplo, roteamento, execução de controllers, templates etc). Mais tempo ainda será necessário para tratar o envio de formulários, validação das entradas, logs e segurança. Por que você tem que reinventar soluções para todos esses problemas?

Adicione um toque de Symfony2

Symfony2 para a salvação. Antes de realmente utilizar o Symfony2, você precisa ter certeza que o PHP sabe onde encontrar as classes do framework. Isso pode ser feito com o autoloader fornecido pelo Symfony. Um autoloader é uma ferramenta que permite a utilização de classes PHP sem a necessidade de incluir os seus arquivos explicitamente.

Primeiro, faça o download do symfony e o coloque no diretório `vendor/symfony/symfony/`. A seguir, crie um o arquivo `app/bootstrap.php`. Utilize-o para dar `require` dos dois arquivos da aplicação e para configurar o autoloader:

```
<?php
// bootstrap.php
require_once 'model.php';
```

```
require_once 'controllers.php';
require_once 'vendor/symfony/symfony/src/Symfony/Component/ClassLoader/UniversalClassLoader.php';

$loader = new Symfony\Component\ClassLoader\UniversalClassLoader();
$loader->registerNamespaces(array(
    'Symfony' => __DIR__.'/../vendor/symfony/symfony/src',
));

$loader->register();
```

Esse código diz ao autoloader onde estão as classes do `Symfony`. Com isso, você pode começar a utilizar as classes sem precisar de um `require` para os arquivos que as contém.

Dentro da filosofia do Symfony está a idéia de que a principal tarefa de uma aplicação é interpretar cada requisição e retornar uma resposta. Para essa finalidade, o Symfony2 fornece as classes `Request` e `Response`. Elas são representações orientadas a objetos da requisição HTTP pura sendo processada e da resposta HTTP sendo retornada. Utilize-as para melhorar o blog:

```
<?php
// index.php
require_once 'app/bootstrap.php';

use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;

$request = Request::createFromGlobals();

$uri = $request->getPathInfo();
if ($uri == '/') {
    $response = list_action();
} elseif ($uri == '/show' && $request->query->has('id')) {
    $response = show_action($request->query->get('id'));
} else {
    $html = '<html><body><h1>Page Not Found</h1></body></html>';
    $response = new Response($html, 404);
}

// echo the headers and send the response
$response->send();
```

Agora os controller são responsáveis por retornar um objeto `Response`. Para tornar isso mais fácil, você pode adicionar uma nova função chamada `render_template()`, que, a propósito, funciona de forma um pouco parecida com o mecanismo de template do Symfony2:

```
// controllers.php
use Symfony\Component\HttpFoundation\Response;

function list_action()
{
    $posts = get_all_posts();
    $html = render_template('templates/list.php', array('posts' => $posts));

    return new Response($html);
}

function show_action($id)
{
    $post = get_post_by_id($id);
    $html = render_template('templates/show.php', array('post' => $post));
```

```
    return new Response($html);
}

// helper function to render templates
function render_template($path, array $args)
{
    extract($args);
    ob_start();
    require $path;
    $html = ob_get_clean();

    return $html;
}
```

Ao adicionar uma pequena parte do Symfony2, a aplicação ficou mais flexível e confiável. A classe `Request` fornece uma maneira segura para acessar informações sobre a requisição HTTP. Especificamente, o método `getPathInfo()` retorna a URI limpa (sempre retornando `/show` e nunca `/index.php/show`). Assim, mesmo que o usuário utilize `/index.php/show`, a aplicação é inteligente o suficiente para direcionar a requisição para `show_action()`.

O objeto `Response` dá flexibilidade ao construir a resposta HTTP, permitindo a adição de cabeçalhos HTTP e conteúdo através de um interface orientada a objetos. Apesar das respostas nessa aplicação ainda serem simples, essa flexibilidade será útil conforme a aplicação crescer.

A aplicação de exemplo no Symfony2

O blog já passou por um *longo* caminho, mas ele ainda tem muito código para uma aplicação tão simples. Por esse caminho, nós também inventamos um simples sistema de roteamento e um método utilizando `ob_start()` e `ob_get_clean()` para renderiar templates. Se, por alguma razão, você precisasse continuar a construir esse "framework" do zero, você poderia pelo menos utilizar isoladamente os components Routing e Templating do Symfony, que já resolveriam esses problemas.

Em vez de re-resolver problemas comuns, você pode deixar que o Symfony2 cuide deles pra você. Aqui está um exemplo da mesma aplicação, agora feito com o Symfony2:

```
<?php
// src/Acme/BlogBundle/Controller/BlogController.php
namespace Acme\BlogBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class BlogController extends Controller
{
    public function listAction()
    {
        $posts = $this->get('doctrine')->getEntityManager()
            ->createQuery('SELECT p FROM AcmeBlogBundle:Post p')
            ->execute();

        return $this->render('AcmeBlogBundle:Blog:list.html.php', array('posts' => $posts));
    }

    public function showAction($id)
    {
        $post = $this->get('doctrine')
            ->getEntityManager()
```

```
            ->getRepository('AcmeBlogBundle:Post')
            ->find($id);

        if (!$post) {
            // cause the 404 page not found to be displayed
            throw $this->createNotFoundException();
        }

        return $this->render('AcmeBlogBundle:Blog:show.html.php', array('post' => $post));
    }
}
```

Os dois controller ainda estão bastante leves. Cada um utiliza a biblioteca de ORM Doctrine para recuperar objetos do banco de dados e o componente `Templating` para renderizar e retornar um objeto *Response*. O template list ficou um pouco mais simples:

```
<!-- src/Acme/BlogBundle/Resources/views/Blog/list.html.php -->
<?php $view->extend('::layout.html.php') ?>

<?php $view['slots']->set('title', 'List of Posts') ?>

<h1>List of Posts</h1>
<ul>
    <?php foreach ($posts as $post): ?>
    <li>
        <a href="<?php echo $view['router']->generate('blog_show', array('id' => $post->getId())) ?>
            <?php echo $post->getTitle() ?>
        </a>
    </li>
    <?php endforeach; ?>
</ul>
```

O layout está praticamente idêntico:

```
<!-- app/Resources/views/layout.html.php -->
<html>
    <head>
        <title><?php echo $view['slots']->output('title', 'Default title') ?></title>
    </head>
    <body>
        <?php echo $view['slots']->output('_content') ?>
    </body>
</html>
```

Nota: Vamos deixar o template da página show como um exercício para você, uma vez que é trivial cria-lo com base no template da página list

Quando o mecanismo do Symfony2 (chamado de `Kernel`) é iniciado, ele precisa de um mapa que indique quais controllers devem ser executados de acordo com a requisição. A configuração de roteamento contém essa informação em um formato legível:

```
# app/config/routing.yml
blog_list:
    pattern:  /blog
    defaults: { _controller: AcmeBlogBundle:Blog:list }

blog_show:
```

```
    pattern:  /blog/show/{id}
    defaults: { _controller: AcmeBlogBundle:Blog:show }
```

Agora que o Symfony2 está cuidando dessas tarefas simples, o front controller ficou extremamente simples. Uma vez que ele faz tão pouco, você nunca mais terá que mexer nele depois de criado (e se você estiver utilizando uma distribuição do Symfony2, você nem mesmo precisará cria-lo!):

```
<?php
// web/app.php
require_once __DIR__.'/../app/bootstrap.php';
require_once __DIR__.'/../app/AppKernel.php';

use Symfony\Component\HttpFoundation\Request;

$kernel = new AppKernel('prod', false);
$kernel->handle(Request::createFromGlobals())->send();
```

A única tarefa do front controller é iniciar o mecanismo (`Kernel`) do Symfony2 e passar para ele o objeto `Request` que deve ser manuseado. Então o Symfony utiliza o mapa de rotas para determinar qual controller chamar. Assim como antes, o método controller é responsável por retornar o objeto `Response`. Não há muito mais que ele precise fazer.

Para uma representação visual de como o Symfony2 trata cada requisição, veja o *diagrama de fluxo da requisição*.

Onde é vantagem utilizar o Symfony2

Nos próximos capítulos você irá aprender mais sobre cada parte do Symfony funciona e a organização recomendada para um projeto. Por enquanto, vamos ver como a migração do PHP puro para o Symfony2 facilitou a sua vida:

- A sua aplicação agora tem um **código limpo e organizado de forma consistente** apesar do Symfony não te forçar a isso). Isso aumenta a **usabilidade** e permite que novos desenvolvedores sejam produtivos no seu projeto de uma maneira mais rápida.
- 100% do código que você escreveu é para a *sua* aplicação. Você **não precisa desenvolver ou manter ferramentas de baixo nível** como autoloading, roteamento, ou renderização nos controllers.
- O Symfony2 te dá **acesso a ferramentas open source** como Doctrine e os componentes Templating, Security, Form, Validation e Translation (só para citar alguns).
- A aplicação agora faz uso de **URLs totalmente flexíveis** graças ao componente `Routing`.
- A arquitetura do Symfony2 centrada no HTTP te dá acesso a poderosas ferramentas como **HTTP caching** feito pelo **cache interno de HTTP do Symfony2** ou por ferramentas ainda mais poderosas como o ‘‘Varnish‘_. Esse assunto será tratado em um próximo capítulo sobre caching.

E talvez o melhor de tudo, ao utilizar o Symfony2, você tem acesso a todo um conjunto de **ferramentas open source de alta qualidade desenvolvidas pela comunidade do Symfony2**! Para mais informações, visite o site Symfony2Bundles.org

Melhores templates

Se você optar por utiliza-lo, o Symfony2 vem com um sistema de template padrão chamado Twig que torna mais fácil a tarefa de escrever templates e os deixa mais fácil de ler. Isso significa que a aplicação de exemplo pode ter ainda menos código! Pegue como exemplo o template list escrito com o Twig:

```
{# src/Acme/BlogBundle/Resources/views/Blog/list.html.twig #}
```

```
{% extends "::layout.html.twig" %}
{% block title %}List of Posts{% endblock %}

{% block body %}
    <h1>List of Posts</h1>
    <ul>
        {% for post in posts %}
        <li>
            <a href="{{ path('blog_show', { 'id': post.id }) }}">
                {{ post.title }}
            </a>
        </li>
        {% endfor %}
    </ul>
{% endblock %}
```

O template `layout.html.twig` correspondente também fica mais fácil de escrever:

```
{# app/Resources/views/layout.html.twig #}

<html>
    <head>
        <title>{% block title %}Default title{% endblock %}</title>
    </head>
    <body>
        {% block body %}{% endblock %}
    </body>
</html>
```

O Twig é bem suportado no Symfony2. E, mesmo que os templates em PHP sempre serão suportados pelo framework, continuaremos a discutir sobre as muitas vantagens do Twig. Para mais informações, veja o capítulo sobre templates.

Aprenda mais no Cookbook

- `/cookbook/templating/PHP`
- Como definir Controladores como Serviços

2.1.3 Instalando e Configurando o Symfony

O objetivo deste capítulo é te deixar com uma aplicação pronta e funcionando, feita com o Symfony. Felizmente, o Symfony oferece o que chamamos de "distribuições", que são projetos básicos e funcionais que você pode baixar e utilizar como base para começar a desenvolver imediatamente.

Dica: Se o que você procura são instruções sobre a melhor maneira de criar um projeto e armazena-lo por meio de um sistema de controle de versão, veja *Utilizando Controle de Versão*.

Instalando uma Distribuição do Symfony2

Dica: Primeiro, certifique-se de que você tem um servidor web (Apache, por exemplo) com a versão mais recente possível do PHP (é recomendado o PHP 5.3.8 ou superior). Para mais informações sobre os requisitos do Symfony2,

veja a referência sobre requisitos. Para informações sobre a configuração de seu específico root do servidor web, verifique a seguinte documentação: Apache | Nginx .

O Symfony2 tem pacotes chamados de "distribuições", que são aplicações totalmente funcionais que já vem com as bibliotecas básicas do framework, uma seleção de alguns pacotes úteis, uma estrutura de diretórios com tudo o necessário e algumas configurações padrão. Ao baixar uma distribuição, você está baixando o esqueleto de uma aplicação funcional que pode ser utilizado imediatamente para começar a desenvolver.

Comece acessando a página de download do Symfony2 em http://symfony.com/download. Nessa página, você verá *Symfony Standard Edition*, que é a principal distribuição do Symfony2. Existem duas formas de iniciar o seu projeto:

Opção 1) Composer

Composer é uma biblioteca de gerenciamento de dependências para PHP, que você pode usar para baixar a Edição Standard do Symfony2.

Comece fazendo o download do Composer em qualquer lugar em seu computador local. Se você tem o curl instalado, é tão fácil como:

```
curl -s https://getcomposer.org/installer | php
```

Nota: Se o seu computador não está pronto para usar o Composer, você verá algumas recomendações ao executar este comando. Siga as recomendações para que o Composer funcione corretamente.

O Composer é um arquivo executável PHAR, que você pode usar para baixar a Distribuição Standard:

```
php composer.phar create-project symfony/framework-standard-edition /path/to/webroot/Symfony 2.1.x-de
```

Dica: Para uma versão exata, substitua *2.1.x-dev* com a versão mais recente do Symfony (por exemplo, 2.1.1). Para mais detalhes, consulte a **'Página de Instalação do Symfony'_**

Este comando pode demorar alguns minutos para ser executado pois o Composer baixa a Distribuição Padrão, juntamente com todas as bibliotecas vendor de que ela precisa. Quando terminar, você deve ter um diretório parecido com o seguinte:

```
path/to/webroot/ <- your web root directory
    Symfony/ <- the new directory
        app/
            cache/
            config/
            logs/
        src/
            ...
        vendor/
            ...
        web/
            app.php
            ...
```

Opção 2) Fazer download de um arquivo

Você também pode fazer download de um arquivo da Edição Standard. Aqui, você vai precisar fazer duas escolhas:

- Faça o download do arquivo `tgz` ou `zip` - ambos são equivalentes, faça o download daquele que você está mais confortável em usar;
- Faça o download da distribuição com ou sem vendors. Se você está pensando em usar mais bibliotecas de terceiros ou bundles e gerenciá-los através do Composer, você provavelmente deve baixar "sem vendors".

Baixe um dos arquivos em algum lugar sob o diretório raiz do seu servidor web local e descompacte-o. A partir de uma linha de comando UNIX, isto pode ser feito com um dos seguintes comandos (substituindo `###` com o seu nome real do arquivo):

```
# for .tgz file
$ tar zxvf Symfony_Standard_Vendors_2.1.###.tgz

# for a .zip file
$ unzip Symfony_Standard_Vendors_2.1.###.zip
```

Se você baixou o arquivo "sem vendors", você definitivamente precisa ler a próxima seção.

Você pode facilmente substituir a estrutura de diretórios padrão. Veja Como Substituir a Estrutura de Diretório Padrão do Symfony para mais informações.

Atualizando os Vendors

Neste ponto, você baixou um projeto Symfony totalmente funcional em que você vai começar a desenvolver a sua própria aplicação. Um projeto Symfony depende de um número de bibliotecas externas. Estas são baixadas no diretório *vendor/* do seu projeto através de uma biblioteca chamada *Composer_*.

Dependendo de como você baixou o Symfony, você pode ou não precisar fazer a atualização de seus vendors agora. Mas, a atualização de seus vendors é sempre segura, e garante que você tem todas as bibliotecas vendor que você precisa.

Passo 1: Obter o *Composer* _ (O excelente novo sistema de pacotes do PHP)

```
curl -s http://getcomposer.org/installer | php
```

Certifique-se de que você baixou o `composer.phar` no mesmo diretório onde o arquivo `composer.json` encontra-se (por padrão, no raiz de seu projeto Symfony).

Passo 2: Instalar os vendors

```
$ php composer.phar install
```

Este comando faz o download de todas as bibliotecas vendor necessárias - incluindo o Symfony em si - dentro do diretório `vendor/`.

Nota: Se você não tem o `curl` instalado, você também pode apenas baixar o arquivo `installer` manualmente em http://getcomposer.org/installer. Coloque este arquivo em seu projeto e execute:

```
php installer
php composer.phar install
```

Dica: Ao executar `php composer.phar install` ou `php composer.phar update`, o composer vai executar comandos de pós instalação/atualização para limpar o cache e instalar os assets. Por padrão, os assets serão copiados para o seu diretório `web`. Para criar links simbólicos em vez de copiar os assets, você pode adicionar uma entrada no nó `extra` do seu arquivo composer.json com a chave `symfony-assets-install` e o valor `symlink`:

```
"extra": {
    "symfony-app-dir": "app",
    "symfony-web-dir": "web",
    "symfony-assets-install": "symlink"
}
```

Ao passar `relative` ao invés de ``symlink` para o symfony-assets-install, o comando irá gerar links simbólicos relativos.

Configuração e Instalação

Nesse ponto, todas as bibliotecas de terceiros necessários encontram-se no diretório `vendor/`. Você também tem um instalação padrão da aplicação em `app/` e alguns códigos de exemplo no diretório `src/`.

O Symfony2 tem um script para testar as configuração do servidor de forma visual, que ajuda garantir que o servidor web e o PHP estão configurados para o framework. Utilize a seguinte URL para verificar a sua configuração:

```
http://localhost/config.php
```

Se algum problema foi encontrado, ele deve ser corrigido agora, antes de prosseguir.

Configurando as Permissões

Um problema comum é que os diretórios `app/cache` e `app/logs` devem ter permissão de escrita para o servidor web e para o usuário da linha de comando. Em um sistema UNIX, se o usuário do seu servidor web for diferente do seu usuário da linha de comando, você pode executar os seguintes comandos para garantir que as permissões estejam configuradas corretamente. Mude o `www-data` para o usuário do servidor web e o `yourname` para o usuário da linha de comando:

1. Utilizando ACL em um sistema que suporta chmod +a

Muitos sistemas permitem que você utilize o comando `chmod +a`. Tente ele primeiro e se der erro tente o próximo método:

```
rm -rf app/cache/*
rm -rf app/logs/*

sudo chmod +a "www-data allow delete,write,append,file_inherit,directory_inherit" app/cache app/logs
sudo chmod +a "yourname allow delete,write,append,file_inherit,directory_inherit" app/cache app/logs
```

2. Utilizando ACL em um sistema que não suporta chmod +a

Alguns sistemas não suportam o comando `chmod +a`, mas suportam um outro chamado `setfacl`. Pode ser necessário que você habilite o suporte a ACL na sua partição e instale o setfacl antes de utiliza-lo (esse é o caso no Ubuntu, por exemplo) da seguinte maneira:

```
sudo setfacl -R -m u:www-data:rwx -m u:yourname:rwx app/cache app/logs
sudo setfacl -dR -m u:www-data:rwx -m u:yourname:rwx app/cache app/logs
```

3. Sem utilizar ACL

Se você não tem acesso para alterar a ACL de diretórios, será necessário alterar a umask para que os diretórios de cache e log tenham permissão de escrita para o grupo ou para todos (vai depender se o usuário do servidor web e o usuário da linha de comando estão no mesmo grupo). Para isso, coloque a seguinte linha no começo dos arquivos `app/console`, `web/app.php` e `web/app_dev.php`:

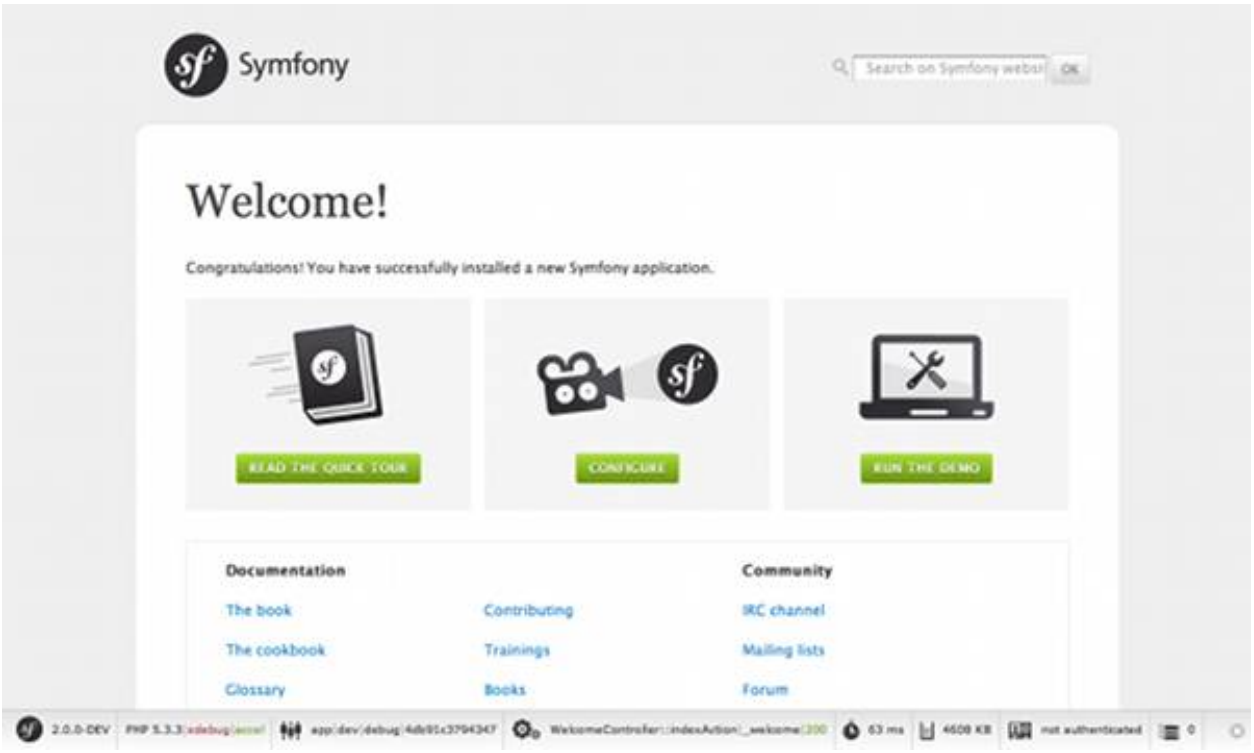
```
umask(0002); // This will let the permissions be 0775

// or

umask(0000); // This will let the permissions be 0777
```

Note que se você tem acesso a ACL no seu servidor, esse será o método recomendado, uma vez que alterar a umask não é uma operação thread-safe.

Quando tudo estiver feito, clique em "Go to the Welcome page" para acessar a sua primeira webpage Symfony2 "real":

```
http://localhost/app_dev.php/
```

O Symfony2 deverá lhe dar as boas vindas e parabeniza-lo pelo trabalho duro até agora!



Iniciando o Desenvolvimento

Agora que você tem uma aplicação Symfony2 totalmente funcional, você pode começar o desenvolvimento! A sua distribuição deve conter alguns códigos de exemplo - verifique o arquivo `README.rst` incluído na distribuição (você pode abri-lo como um arquivo de texto) para aprender sobre os exemplos incluídos e como você pode removê-los mais tarde.

Se você é novo no Symfony, junte-se a nós em "`page_creation`", onde você aprenderá como criar páginas, mudar configurações e tudo mais que precisará para a sua nova aplicação.

Certifique-se também verificar o Cookbook, que contém uma grande variedade de artigos sobre a resolução de problemas específicos com Symfony.

Utilizando Controle de Versão

Se você está utilizando um sistema de controle de versão como `Git` ou `Subversion`, você pode instala-lo e começar a realizar os commits do seu projeto normalmente. A edição padrão do Symfony *é* o ponto inicial para o seu novo projeto.

Para instruções específicas sobre a melhor maneira de configurar o seu projeto para ser armazenado no git, veja Como Criar e Armazenar um Projeto Symfony2 no git.

Ignorando o diretório `vendor/`

Se você baixou o arquivo *sem itens de terceiros* (without vendors), você pode ignorar todo o diretório `vendor/` com segurança e não enviá-lo para o controle de versão. No `Git`, isso é feito criando e o arquivo `.gitignore` e adicionando a seguinte linha:

```
/vendor/
```

Agora, o diretório vendor não será enviado para o controle de versão. Isso é bom (na verdade, é ótimo!) porque quando alguém clonar ou fizer check out do projeto, ele/ela poderá simplesmente executar o script `php composer.phar install` para instalar todas as bibliotecas vendor necessárias.

2.1.4 Controlador

Um controlador é uma função PHP que você cria e que pega informações da requisição HTTP para criar e retornar uma resposta HTTP (como um objeto `Response` do Symfony2). A resposta pode ser uma página HTML, um documento XML, um array JSON serializado, uma imagem, um redirecionamento, um erro 404 ou qualquer coisa que você imaginar. O controlador contém toda e qualquer lógica arbitrária que *sua aplicação* precisa para renderizar o conteúdo de uma página.

Para ver quão simples é isso, vamos ver um controlador do Symfony2 em ação. O seguinte controlador deve renderizar uma página que mostra apenas `Hello world!`:

```
use SymfonyComponentHttpFoundationResponse;

public function helloAction() {

    return new Response('Hello world!');

}
```

O objetivo de um controlador é sempre o mesmo: criar e retornar um objeto `Response`. Ao longo do caminho, ele pode ler informações da requisição, carregar um recurso do banco de dados, mandar um e-mail ou gravar informações na sessão do usuário. Mas em todos os casos, o controlador acabará retornando o objeto `Response` que será mandado de volta para o cliente.

Não há nenhuma mágica e nenhum outro requisito para se preocupar! Aqui temos alguns exemplos comuns:

- O *Controlador A* prepara um objeto `Response` representando o conteúdo da página inicial do site.
- O *Controlador B* lê o parâmetro `slug` da requisição para carregar uma entrada do blog no banco de dados e cria um um objeto `Response` mostrando o blog. Se o `slug` não for encontrado no banco de dados, ele cria e retorna um objeto `Response` com um código de status 404.
- O *Controlador C* trata a o envio de um formulário de contato. Ele lê a informação do formulário a partir da requisição, salva a informação de contato no banco de dados e envia por e-mail a informação de contato para o webmaster. Finalmente, ele cria um objeto `Response` que redireciona o navegador do cliente para a página "thank you" do formulário de contato.

O Ciclo de Vida da Requisição, Controlador e Resposta

Toda requisição tratada por um projeto com Symfony 2 passa pelo mesmo ciclo de vida simples. O framework cuida das tarefas repetitivas e por fim executa um controlador onde reside o código personalizado da sua aplicação:

1. Toda requisição é tratada por um único arquivo front controlador (por exemplo, `app.php` ou `app_dev.php`) que inicializa a aplicação;
2. O `Router` lê a informação da requisição (por exemplo, a URI), encontra uma rota que casa com aquela informação e lê o parâmetro `_controller` da rota;
3. O controlador que casou com a rota é executado e o código dentro do controlador cria e retorna um objeto `Response`;
4. Os cabeçalhos HTTP e o conteúdo do objeto `Response` são enviados de volta para o cliente.

Criar uma página é tão fácil quanto criar um controlador (#3) e fazer uma rota que mapeie uma URL para aquele controlador (#2).

Nota: Embora tenha um nome similar, um "front controller" é diferente dos "controladores" dos quais vamos falar nesse capítulo. Um front controller é um pequeno arquivo PHP que fica no seu diretório web e através do qual todas as requisições são direcionadas. Uma aplicação típica terá um front controller de produção (por exemplo, `app.php`) e um front controller de desenvolvimento (por exemplo, `app_dev.php`). Provavelmente você nunca precisará editar, visualizar ou se preocupar com os front controllers da sua aplicação.

Um Controlador Simples

Embora um controlador possa ser qualquer código PHP que possa ser chamado (uma função, um método em um objeto ou uma `Closure`), no Symfony2 um controlador geralmente é um único método dentro de um objeto controlador. Os controladores também são chamados de *ações*:

```
1  // src/Acme/HelloBundle/Controller/HelloController.php
2
3  namespace Acme\HelloBundle\Controller;
4  use Symfony\Component\HttpFoundation\Response;
5
6  class HelloController
7  {
8      public function indexAction($name)
9      {
10       return new Response('<html><body>Hello '.$name.'!</body></html>');
11     }
12 }
```

Dica: Note que o *controlador* é o método `indexAction`, que fica dentro de uma *classe controladora* (`HelloController`). Não se confunda com a nomenclatura: uma *classe controladora* é apenas um forma conveniente de agrupar vários controladores/ações juntos. Geralmente a classe controladora irá agrupar vários controladores/ações (por exemplo, `updateAction`, `deleteAction` etc).

Esse controlador é bem simples, mas vamos explicá-lo:

- *linha 3*: O Symfony2 se beneficia da funcionalidade de namespace do PHP 5.3 colocando a classe controladora inteira dentro de um namespace. A palavra chave `use` importa a classe `Response` que nosso controlador tem que retornar.
- *linha 6*: O nome da classe é a concatenação de um nome para a classe controlador (ou seja, `Hello`) com a palavra `Controller`. Essa é uma convenção que fornece consistência aos controladores e permite que eles sejam referenciados usando apenas a primeira parte do nome (ou seja, `Hello`) na configuração de roteamento.
- *linha 8*: Toda ação em uma classe controladora é sufixada com `Action` e é referenciada na configuração de roteamento pelo nome da ação (`index`). Na próxima seção, você criará uma rota que mapeia uma URI para essa action. Você aprenderá como os marcadores de posição das rotas (`{name}`) tornam-se argumentos no método da action (`$name`).
- *linha 10*: O controlador cria e retorna um objeto `Response`.

Mapeando uma URL para um Controlador

O novo controlador retorna uma página HTML simples. Para ver realmente essa página no seu navegador você precisa criar uma rota que mapeia um padrão específico de URL para o controlador:

- *YAML*

```
# app/config/routing.yml
hello:
    pattern:      /hello/{name}
    defaults:     { _controller: AcmeHelloBundle:Hello:index }
```

- *XML*

```
<!-- app/config/routing.xml -->
<route id="hello" pattern="/hello/{name}">
    <default key="_controller">AcmeHelloBundle:Hello:index</default>
</route>
```

- *PHP*

```
// app/config/routing.php
$collection->add('hello', new Route('/hello/{name}', array(
    '_controller' => 'AcmeHelloBundle:Hello:index',
)));
```

Agora, acessar `/hello/ryan` executa o controlador `HelloController::indexAction()` e passa `ryan` para a variável `$name`. A criação de uma "página" significa simplesmente criar um método controlador e associar uma rota.

Note a sintaxe usada para referenciar o controlador: `AcmeHelloBundle:Hello:index`. O Symfony2 usa uma notação flexível de string para referenciar diferentes controladores. Essa é a sintaxe mais comum e diz ao Symfony2 para buscar por uma classe controladora chamada `helloController` dentro de um bundle chamado `AcmeHelloBundle`. Então o método `indexAction()` é executado.

Para mais detalhes sobre o formato de string usado para referenciar diferentes controladores, veja *Padrão de nomeação do Controlador*.

Nota: Esse exemplo coloca a configuração de roteamento diretamente no diretório `app/config/`. Uma forma melhor de organizar suas rotas é colocar cada uma das rotas no bundle a qual elas pertencem. Para mais informações, veja *Incluindo Recursos Externos de Roteamento*.

Dica: Você pode aprender muito mais sobre o sistema de roteamento no capítulo Roteamento.

Parâmetros de Rota como Argumentos do Controlador

Você já sabe que o parâmetro `_controller` em `AcmeHelloBundle:Hello:index` se refere ao método `HelloController::indexAction()` que está dentro do bundle `AcmeHelloBundle`. O que é mais interessante são os argumentos que são passados para o método:

```
<?php
// src/Acme/HelloBundle/Controller/HelloController.php

namespace Acme\HelloBundle\Controller;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
```

```
class HelloController extends Controller
{
    public function indexAction($name)
    {
      // ...
    }
}
```

O controlador tem um único argumento, `$name`, que corresponde ao parâmetro `{name}` da rota casada (`ryan` no nosso exemplo). Na verdade quando executa seu controlador, o Symfony2 casa cada um dos argumentos do controlador com um parâmetro da rota casada. Veja o seguinte exemplo:

- *YAML*

```
# app/config/routing.yml
hello:
    pattern:      /hello/{first_name}/{last_name}
    defaults:     { _controller: AcmeHelloBundle:Hello:index, color: green }
```

- *XML*

```
<!-- app/config/routing.xml -->
<route id="hello" pattern="/hello/{first_name}/{last_name}">
    <default key="_controller">AcmeHelloBundle:Hello:index</default>
    <default key="color">green</default>
</route>
```

- *PHP*

```
// app/config/routing.php
$collection->add('hello', new Route('/hello/{first_name}/{last_name}', array(
    '_controller' => 'AcmeHelloBundle:Hello:index',
    'color'       => 'green',
)));
```

O controlador dessa rota pode receber vários argumentos:

```
public function indexAction($first_name, $last_name, $color) {

    // ...

}
```

Observe que tanto as variáveis de marcadores de posição (`{first_name}`, `{last_name}`) quanto a váriavel padrão `color` estão disponíveis como argumentos no controlador. Quando uma rota é casada, as variáveis marcadoras de posição são mescladas com as variáveis `default` criando um array que fica disponível para o seu controlador.

O mapeamento de parâmetros de rota com argumentos do controlador é fácil e flexível. Tenha em mente as seguintes orientações enquanto estiver desenvolvendo.

- **A ordem dos argumentos do controlador não importa**

  O Symfony é capaz de casar os nomes dos parâmetros da rota com os nomes das variáveis na assinatura do método do controlador. Em outras palavras, ele sabe que o parâmetro `{last_name}` casa com o argumento `$last_name`. Os argumentos do controlador podem ser totalmente reordenados e continuam funcionando perfeitamente:

  ```
  public function indexAction($last_name, $color, $first_name) {

      // ..

  }
  ```

- **Todo argumento obrigatório do controlador tem que corresponder a um parâmetro de roteamento**

  O seguinte deveria lançar uma `RuntimeException` porque não existe nenhum parâmetro `foo` definido na rota:

  ```
  public function indexAction($first_name, $last_name, $color, $foo) {
      // ..
  }
  ```

  Deixando o argumento opcional, no entanto, tudo corre bem. O seguinte exemplo não lança uma exceção:

  ```
  public function indexAction($first_name, $last_name, $color, $foo = 'bar') {
      // ..
  }
  ```

- **Nem todos os parâmetros de roteamento precisam ser argumentos no seu controlador**

  Se, por exemplo, `last_name` não for importante para o seu controlador, você pode omitir inteiramente ele:

  ```
  public function indexAction($first_name, $color) {
      // ..
  }
  ```

Dica: Cada uma das rotas tem um parâmetro `_route` especial, que é igual ao nome da rota que foi casada (por exemplo, `hello`). Embora não seja útil geralmente, ele também fica disponível como um argumento do controlador.

O `Request` como um Argumento do Controlador

Por conveniência, você também pode fazer com que o Symfony passe o objeto `Request` como um argumento para seu controlador. Isso é conveniente especialmente quando você estiver trabalhando com formulários, por exemplo:

```
use SymfonyComponentHttpFoundationRequest;

public function updateAction(Request $request) {
    $form = $this->createForm(...);

    $form->bind($request); // ...
}
```

A Classe Base do Controlador

Por conveniência, o Symfony2 vem com uma classe `Controller` base que ajuda com algumas das tarefas mais comuns dos controladores e fornece às suas classes controladoras acesso à qualquer recurso que elas possam precisar. Estendendo essa classe `Controller`, você se beneficia com vários métodos helper.

Adicione a instrução `use` no topo da sua classe `Controller` e então modifique o `HelloController` para estendê-lo:

```
// src/Acme/HelloBundle/Controller/HelloController.php

namespace Acme\HelloBundle\Controller;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Response;

class HelloController extends Controller
{
    public function indexAction($name)
    {
      return new Response('<html><body>Hello '.$name.'!</body></html>');
    }
}
```

Isso não muda realmente nada o jeito que seu controlador trabalha. Na próxima seção você aprenderá sobre os métodos helper que a classe controladora base disponibiliza. Esses métodos são apenas atalhos para usar funcionalidades do núcleo do Symfony2 que estão disponíveis para você usando ou não a classe base `Controller`. Uma boa maneira de ver a funcionalidade do núcleo em ação é olhar a própria classe `Controller`.

Dica: Estender a classe base é *opcional* no Symfony; ela contém atalhos úteis mas nada que seja mandatório. Você também pode estender `Symfony\Component\DependencyInjection\ContainerAware`. O objeto contêiner de serviços então será acessível por meio da propriedade `container`.

Nota: Você também pode definir seus Controllers como Serviços.

Tarefas Comuns dos Controladores

Embora virtualmente um controlador possa fazer qualquer coisa, a maioria dos controladores irão realizar as mesmas tarefas básicas repetidas vezes. Essas tarefas, como redirecionamentos, direcionamentos, renderização de templates e acesso a serviços nucleares são muitos fáceis de gerenciar no Symfony2.

Redirecionando

Se você quiser redirecionar o usuário para outra página, use o método `redirect()`:

```
public function indexAction()
{
    return $this->redirect($this->generateUrl('homepage'));
}
```

O método `generateUrl()` é apenas uma função helper que gera a URL de uma determinada rota. Para mais informações, veja o capítulo Roteamento.

Por padrão, o método `redirect()` efetua um redirecionamento 302 (temporário). Para realizar um redirecionamento 301 (permanente), modifique o segundo argumento:

```
public function indexAction()
{
    return $this->redirect($this->generateUrl('homepage'), 301);
}
```

Dica: O método `redirect()` é simplesmente um atalho que cria um objeto `Response` especializado em redirecionar o usuário. Ele é equivalente a:

```
use Symfony\Component\HttpFoundation\RedirectResponse;

return new RedirectResponse($this->generateUrl('homepage'));
```

Direcionando

Você também pode facilmente direcionar internamente para outro controlador com o método `forward()`. Em vez de redirecionar o navegador do usuário, ele faz uma sub-requisição interna e chama o controlador especificado. O método `forward()` retorna o objeto `Response` que é retornado pelo controlador:

```
public function indexAction($name)
{
    $response = $this->forward('AcmeHelloBundle:Hello:fancy', array(
        'name'  => $name,
        'color' => 'green'
    ));

    // pode modificar a resposta ou retorná-la diretamente

    return $response;
}
```

Note que o método *forward()* usa a mesma representação em string do controlador que foi usada na configuração de roteamento. Nesse caso, a classe controlador alvo será `HelloController` dentro de `AcmeHelloBundle`. O array passado para o método se torna os argumentos no controlador resultante. Essa mesma interface é usada quando se embutem controladores em templates (veja *Incorporação de Controllers*). O método controlador alvo deve se parecer com o seguinte:

```
public function fancyAction($name, $color)
{
    // ... cria e retorna um objeto Response
}
```

E da mesma forma, quando criamos um controlador para uma rota, a ordem dos argumentos para `fancyAction` não importa. O Symfony2 combina os nomes das chaves dos índices (por exemplo, `name`) com os nomes dos argumentos do método (por exemplo, `$name`). Se você mudar a ordem dos argumentos, o Symfony2 continuará passando os valores corretos para cada variável.

Dica: Assim como em outros métodos do `Controller` base, o método `forward` é apenas um atalho para uma funcionalidade nuclear do Symfony2. Um direcionamento pode ser realizado diretamente por meio do serviço `http_kernel`. Um direcionamento retorna um objeto `Response`:

```
$httpKernel = $this->container->get('http_kernel');
$response = $httpKernel->forward('AcmeHelloBundle:Hello:fancy', array(
    'name'  => $name,
    'color' => 'green',
));
```

Renderizando Templates

Apesar de não ser um requisito, a maioria dos controladores irá, no fim das contas, renderizar um template que é responsável por gerar o HTML (ou outro formato) para o controlador. O método `renderView()` renderiza um template e retorna seu conteúdo. O conteúdo do template pode ser usado para criar um objeto `Response`:

```
$content = $this->renderView('AcmeHelloBundle:Hello:index.html.twig', array('name' => $name));

return new Response($content);
```

Isso pode ser feito até em um único passo usando o método `render()`, que retorna um objeto `Response` com o conteúdo do template:

```
return $this->render('AcmeHelloBundle:Hello:index.html.twig', array('name' => $name));
```

Em ambos os casos, o template `Resources/views/Hello/index.html.twig` dentro do `AcmeHelloBundle` será renderizado.

O sistema de template do Symfony é explicado com mais detalhes no capítulo Templating.

Dica: O método `renderView` é um atalho para usar diretamente o serviço `templating`. O serviço `templating` também pode ser usado diretamente:

```
$templating = $this->get('templating');
$content = $templating->render('AcmeHelloBundle:Hello:index.html.twig', array('name' => $name));
```

Acessando outros Serviços

Quando se estende a classe controladora base, você pode acessar qualquer um dos serviços Symfony2 através do método `get()`. Aqui estão alguns dos serviços mais comuns que você pode precisar:

```
$request = $this->getRequest();

$templating = $this->get('templating');

$router = $this->get('router');

$mailer = $this->get('mailer');
```

Existem outros inúmeros serviços disponíveis e você é encorajado a definir os seus próprios. Para listar todos os serviços disponíveis, use o comando do console `container:debug`:

```
php app/console container:debug
```

Para mais informações, veja o capítulo Container de Serviço.

Gerenciando Erros e Páginas 404

Quando algo não for encontrado, você deve usar de forma correta o protocolo HTTP e retornar uma resposta 404. Para isso, você lançará um tipo especial de exceção. Se estiver estendendo a classe controlador base, faça o seguinte:

```
public function indexAction()
{
    $product = // retrieve the object from database
    if (!$product) {
```

```
        throw $this->createNotFoundException('The product does not exist');
    }

    return $this->render(...);
}
```

O método `createNotFoundException()` cria um objeto especial `NotFoundHttpException`, que no fim dispara uma resposta HTTP 404 de dentro do Symfony.

É lógico que você é livre para lançar qualquer classe `Exception` no seu controlador - o Symfony irá retornar automaticamente uma resposta HTTP código 500.

```
throw new \Exception('Something went wrong!');
```

Em todo caso, uma página de erro estilizada é mostrada para o usuário final e uma página de erro com informações de debug completa é mostrada para o desenvolvedor (no caso de visualizar a página no modo debug). Ambas as páginas podem ser personalizadas. Para detalhes, leia a receita "Como personalizar as páginas de erro" no cookbook.

Gerenciando a Sessão

O Symfony2 fornece um objeto de sessão muito bom que você pode usar para guardar informações sobre o usuário (seja ele uma pessoa real usando um navegador, um robô ou um web service) entre requisições. Por padrão, o Symfony2 guarda os atributos em um cookie usando as sessões nativas do PHP.

O armazenamento e a recuperação de informações da sessão são feitos facilmente de qualquer controlador:

```
$session = $this->getRequest()->getSession();

// store an attribute for reuse during a later user request
$session->set('foo', 'bar');

// in another controller for another request
$foo = $session->get('foo');

// use a default value if the key doesn't exist
$filters = $session->get('filters', array());
```

Esses atributos permanecerão no usuário até o fim da sessão.

Mensagens Flash

Você também guardar pequenas mensagens que serão armazenadas na sessão do usuário apenas por uma requisição. Isso é útil no processamento de formulários: você pode redirecionar o usuário e mostrar uma mensagem especial na requisição *seguinte*. Esses tipos de mensagens são chamadas de mensagens "flash".

Por exemplo, imagine que você esteja processando a submissão de um formulário:

```
public function updateAction()
{
    $form = $this->createForm(...);

    $form->bind($this->getRequest());
    if ($form->isValid()) {
        // do some sort of processing

        $this->get('session')->getFlashBag()->add('notice', 'Your changes were saved!');
```

```
        return $this->redirect($this->generateUrl(...));
    }

    return $this->render(...);
}
```

Depois do processamento da requisição, o controlador define uma mensagem flash *notice* e então faz o redirecionamento. O nome (`notice`) não é importante - é apenas o que você usa para identificar o tipo da mensagem.

No template da próxima action, o código a seguir poderia ser usado para renderizar a mensagem `notice`:

- *Twig*

```
{% for flashMessage in app.session.flashbag.get('notice') %}
    <div class="flash-notice">
        {{ flashMessage }}
    </div>
{% endfor %}
```

- *PHP*

```
<?php foreach ($view['session']->getFlashBag()->get('notice') as $message): ?>
    <div class="flash-notice">
        <?php echo "<div class='flash-error'>$message</div>" ?>
    </div>
<?php endforeach; ?>
```

Por definição, as mensagens flash são feitas para existirem por exatamente uma requisição (elas "se vão num instante" - "gone in a flash"). Elas foram projetadas para serem usadas entre redirecionamentos exatamente como você fez nesse exemplo.

O Objeto Response

O único requisito de um controlador é retornar um objeto `Response`. A classe `Response` é uma abstração PHP em volta da resposta HTTP - a mensagem em texto cheia de cabeçalhos HTTP e conteúdo que é mandado de volta para o cliente:

```
// create a simple Response with a 200 status code (the default)
$response = new Response('Hello '.$name, 200);

// create a JSON-response with a 200 status code
$response = new Response(json_encode(array('name' => $name)));
$response->headers->set('Content-Type', 'application/json');
```

Dica: A propriedade `headers` é a classe `HeaderBag` com vários métodos úteis para ler e modificar os cabeçalhos do `Response`. Os nomes dos cabeçalhos são normalizados de forma que usar `Content-Type` seja equivalente a `content-type` ou mesmo `content_type`.

O Objeto Request

Além dos valores nos marcadores de roteamento, o controlador também tem acesso ao objeto `Request` quando está estendendo a classe `Controller` base:

```
$request = $this->getRequest();

$request->isXmlHttpRequest(); // is it an Ajax request?

$request->getPreferredLanguage(array('en', 'fr'));

$request->query->get('page'); // get a $_GET parameter

$request->request->get('page'); // get a $_POST parameter
```

Assim como com o objeto `Response`, os cabeçalhos da requisição são guardados em um objeto `HeaderBag` e são facilmente acessados.

Considerações Finais

Sempre que criar uma página, no final você precisará escrever algum código que contenha a lógica dessa página. No Symfony, isso é chamado de controlador, e ele é uma função PHP que faz tudo que for necessário para no fim retornar o objeto `Response` final que será retornado ao usuário.

Para facilitar a vida, você pode escolher estender uma classe `Controller` base, que contém métodos que são atalhos para muitas tarefas comuns dos controladores. Por exemplo, uma vez que você não queira colocar código HTML no seu controlador, você pode usar o método `render()` para renderizar e retornar o conteúdo de um template.

Em outros capítulos, você verá como o controlador pode ser usado para persistir e buscar objetos em um banco de dados, processar submissões de formulários, gerenciar cache e muito mais.

Saiba mais no Cookbook

- Como personalizar as páginas de erro
- Como definir Controladores como Serviços

2.1.5 Roteamento

URLs bonitas são uma obrigação absoluta para qualquer aplicação web séria. Isto significa deixar para trás URLs feias como `index.php?article_id=57` em favor de algo como `/read/intro-to-symfony`.

Ter flexibilidade é ainda mais importante. E se você precisasse mudar a URL de uma página de `/blog` para `/news`? Quantos links você precisaria para investigá-los e atualizar para fazer a mudança ? Se você está usando o roteador do Symfony, a mudança é simples.

O roteador do Symfony2 deixa você definir URLs criativas que você mapeia para diferentes áreas de sua aplicação. No final deste capítulo, você será capaz de : * Criar rotas complexas que mapeiam para os controladores * Gerar URLs dentro de templates e controladores * Carregar recursos de roteamento de pacotes (ou algum lugar a mais) * Depurar suas rotas

Roteamento em Ação

Um *rota* é um mapa de um padrão URL para um controlador. Por exemplo, suponha que você queira ajustar qualquer URL como `/blog/my-post` ou `/blog/all-about-symfony` e enviá-la ao controlador que pode olhar e mudar aquela entrada do blog. A rota é simples:

- *YAML*

```
    # app/config/routing.yml
    blog_show:
        pattern:   /blog/{slug}
        defaults:  { _controller: AcmeBlogBundle:Blog:show }
```

- *XML*

```
    <!-- app/config/routing.xml -->
    <?xml version="1.0" encoding="UTF-8" ?>
    <routes xmlns="http://symfony.com/schema/routing"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://symfony.com/schema/routing http://symfony.com/schema/routing/rout

        <route id="blog_show" pattern="/blog/{slug}">
            <default key="_controller">AcmeBlogBundle:Blog:show</default>
        </route>
    </routes>
```

- *PHP*

```
    // app/config/routing.php
    use Symfony\Component\Routing\RouteCollection;
    use Symfony\Component\Routing\Route;

    $collection = new RouteCollection();
    $collection->add('blog_show', new Route('/blog/{slug}', array(
        '_controller' => 'AcmeBlogBundle:Blog:show',
    )));

    return $collection;
```

O padrão definido pela rota `blog_show` age como `/blog/*` onde o coringa é dado pelo nome `slug`. Para a URL `/blog/my-blog-post`, a variável `slug` obtém um valor de `my-blog-post`, que está disponível para você usar em seu controlador (continue lendo).

O parâmetro `_controller` é uma chave especial que avisa o Symfony qual controlador deveria ser executado quando uma URL corresponde a essa rota. A string `_controller` é chamada *logical name*. Ela segue um padrão que aponta para uma classe e método PHP específico:

```
// src/Acme/BlogBundle/Controller/BlogController.php

namespace Acme\BlogBundle\Controller;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class BlogController extends Controller
{
    public function showAction($slug)
    {
        $blog = // use the $slug varible to query the database

        return $this->render('AcmeBlogBundle:Blog:show.html.twig', array(
            'blog' => $blog,
        ));
    }
}
```

Parabéns ! Você agora criou sua primeira rota conectou ela a um controlador. Agora, quando você visitar `/blog/my-post`, o controlador `showAction` será executado e a variável `$slug` será igual a `my-post`.

Esse é o objetivo do roteador do Symfony2: mapear a URL de uma requisição para um controlador. Ao longo do caminho, você aprenderá todos os tipos de truques que tornam o mapeamento fácil, mesmo das URLS mais complexas.

Roteamento: Por debaixo do capuz

Quando uma requisição é feita para sua aplicação, ela contém um endereço para o "recurso" exato que o cliente está requisitando.Esse endereço é chamado de URL, (ou URI), e poderia ser `/contact`, `/blog/read-me`, ou qualquer coisa a mais. Considere a seguinte requisição de exemplo :

```
GET /blog/my-blog-post
```

O objetido do sistema de roteamento do Symfony2 é analisar esta URL e determinar qual controlador deveria ser executado. O processo interior parece isso:

1. A requisação é controlada pelo front controller do Symfony2 front controller (ex: `app.php`);
2. O núcleo do Symfony2 (ex: Kernel) pergunta ao roteador para inspecionar a requisição;
3. O roteador ajusta a URL recebida para uma rota específica e retorna informação sobre a rota, incluindo o controlador que deveria ser executado;
4. O kernel do Symfony2 executa o controlador, que retorna por último um objeto `Response`.

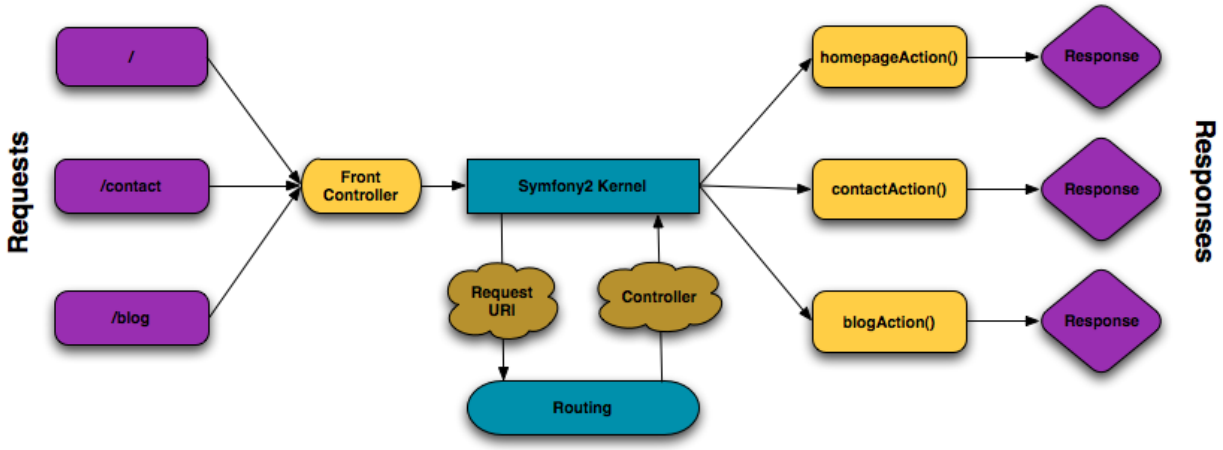


Fig. 2.2: A camada de roteamento é uma ferramenta que traduza a URL recebida em um controlador específico para executar.

Criando rotas

Symfony carrega todas as rotas para sua aplicação de um arquivo de configuração de roteamento. O arquivo é geralmente `app/config/routing.yml`, mas pode ser configurado para ser qualquer coisa (incluindo um arquivo XML ou PHP) via arquivo de configuração de aplicação:

- *YAML*

```
# app/config/config.yml
framework:
    # ...
    router:        { resource: "%kernel.root_dir%/config/routing.yml" }
```

- *XML*

```
<!-- app/config/config.xml -->
<framework:config ...>
    <!-- ... -->
    <framework:router resource="%kernel.root_dir%/config/routing.xml" />
</framework:config>
```

- *PHP*

```
// app/config/config.php
$container->loadFromExtension('framework', array(
    // ...
    'router'        => array('resource' => '%kernel.root_dir%/config/routing.php'),
));
```

Dica: Mesmo que toda as rotas sejam carregadas de um arquivo só, é uma prática comum incluir recursos de roteamento adicionais de dentro do arquivo. Veja a seção:ref:*routing-include-external-resources* para mais informação.

Configuração de rota básica

Definir uma rota é fácil, e uma aplicação típica terá um monte de rotas. A basic route consists of just two parts: the `pattern` to match and a `defaults` array:

- *YAML*

```
_welcome:
    pattern:   /
    defaults:  { _controller: AcmeDemoBundle:Main:homepage }
```

- *XML*

```
<?xml version="1.0" encoding="UTF-8" ?>

<routes xmlns="http://symfony.com/schema/routing"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/routing http://symfony.com/schema/routing/rout

    <route id="_welcome" pattern="/">
        <default key="_controller">AcmeDemoBundle:Main:homepage</default>
    </route>

</routes>
```

- *PHP*

```
use Symfony\Component\Routing\RouteCollection;
use Symfony\Component\Routing\Route;

$collection = new RouteCollection();
$collection->add('_welcome', new Route('/', array(
    '_controller' => 'AcmeDemoBundle:Main:homepage',
)));

return $collection;
```

A rota combina a homepage (`/`) e mapeia ele para o controlador `AcmeDemoBundle:Main:homepage`. A string `_controller` é traduzida pelo Symfony2 em uma função verdadeira do PHP e exectudada. Aquele processo irá ser

explicado brevemente na seção *Padrão de nomeação do Controlador*.

Roteando com Espaços reservados

Claro que o sistema de roteamento suporta rotas muito mais interessantes. Muitas rotas irão conter uma ou mais chamadas de espaços reservados "coringa":

- *YAML*

```
blog_show:
    pattern:   /blog/{slug}
    defaults:  { _controller: AcmeBlogBundle:Blog:show }
```

- *XML*

```
<?xml version="1.0" encoding="UTF-8" ?>

<routes xmlns="http://symfony.com/schema/routing"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/routing http://symfony.com/schema/routing/rout

    <route id="blog_show" pattern="/blog/{slug}">
        <default key="_controller">AcmeBlogBundle:Blog:show</default>
    </route>
</routes>
```

- *PHP*

```
use Symfony\Component\Routing\RouteCollection;
use Symfony\Component\Routing\Route;

$collection = new RouteCollection();
$collection->add('blog_show', new Route('/blog/{slug}', array(
    '_controller' => 'AcmeBlogBundle:Blog:show',
)));

return $collection;
```

O padrão irá corresponder a qualquer coisa que pareça `/blog/*`. Melhor ainda, o valor correspondendo ao espaço reservado `{slug}` estará disponível no seu controlador. Em outras palavras, se a URL é `/blog/hello-world`, uma variável `$slug`, com o valor de `hello-world`, estará disponível no controlador. Isto pode ser usado, por exemplo, para carregar o post do blog correspondendo àquela string.

Este padrão *não* irá, entretanto, simplesmente ajustar `/blog`. Isso é porque, por padrão, todos os espaços reservados são requeridos. Isto pode ser mudado ao adicionar um valor de espaço reservado ao array `defaults`.

Espaços reservados Requeridos e Opcionais

Para tornar as coisas mais excitantes, adicione uma nova rota que mostre uma lista de todos os posts do blog para essa aplicação de blog imaginária:

- *YAML*

```
blog:
    pattern:   /blog
    defaults:  { _controller: AcmeBlogBundle:Blog:index }
```

- *XML*

```xml
<?xml version="1.0" encoding="UTF-8" ?>

<routes xmlns="http://symfony.com/schema/routing"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/routing http://symfony.com/schema/routing/rout

    <route id="blog" pattern="/blog">
        <default key="_controller">AcmeBlogBundle:Blog:index</default>
    </route>
</routes>
```

- *PHP*

```php
use Symfony\Component\Routing\RouteCollection;
use Symfony\Component\Routing\Route;

$collection = new RouteCollection();
$collection->add('blog', new Route('/blog', array(
    '_controller' => 'AcmeBlogBundle:Blog:index',
)));

return $collection;
```

Até agora, essa rota é tão simples quanto possível - contém nenhum espaço reservado e só irá corresponder à URL exata `/blog`. Mas e se você precisar dessa rota para suportar paginação, onde `/blog/2` mostre a segunda página do entradas do blog ? Atualize a rota para ter uma nova `{page}` de espaço reservado:

- *YAML*

```yaml
blog:
    pattern:   /blog/{page}
    defaults:  { _controller: AcmeBlogBundle:Blog:index }
```

- *XML*

```xml
<?xml version="1.0" encoding="UTF-8" ?>

<routes xmlns="http://symfony.com/schema/routing"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/routing http://symfony.com/schema/routing/rout

    <route id="blog" pattern="/blog/{page}">
        <default key="_controller">AcmeBlogBundle:Blog:index</default>
    </route>
</routes>
```

- *PHP*

```php
use Symfony\Component\Routing\RouteCollection;
use Symfony\Component\Routing\Route;

$collection = new RouteCollection();
$collection->add('blog', new Route('/blog/{page}', array(
    '_controller' => 'AcmeBlogBundle:Blog:index',
)));

return $collection;
```

Como o espaço reservado `{slug}` anterior, o valor correspondendo a `{page}` estará disponível dentro do seu controlador. Este valor pode ser usado para determinar qual conjunto de posts do blog mostrar para determinada página.

Mas espere ! Como espaços reservados são requeridos por padrão, essa rota não irá mais corresponder simplesmente a `/blog`. Ao invés disso, para ver a página 1 do blog, você precisaria usar a URL `/blog/1`! Como não há meios para uma aplicação web ricase comportar, modifique a rota para fazer o parâmetro `{page}` opcional. Isto é feito ao incluir na coleção `defaults`:

- *YAML*

```
blog:
    pattern:   /blog/{page}
    defaults:  { _controller: AcmeBlogBundle:Blog:index, page: 1 }
```

- *XML*

```
<?xml version="1.0" encoding="UTF-8" ?>

<routes xmlns="http://symfony.com/schema/routing"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/routing http://symfony.com/schema/routing/rout

    <route id="blog" pattern="/blog/{page}">
        <default key="_controller">AcmeBlogBundle:Blog:index</default>
        <default key="page">1</default>
    </route>
</routes>
```

- *PHP*

```
use Symfony\Component\Routing\RouteCollection;
use Symfony\Component\Routing\Route;

$collection = new RouteCollection();
$collection->add('blog', new Route('/blog/{page}', array(
    '_controller' => 'AcmeBlogBundle:Blog:index',
    'page' => 1,
)));

return $collection;
```

Ao adicionar `page` para a chave `defaults`, o espaço reservado `{page}` não é mais requerido. A URL `/blog` irá corresponder a essa rota e o valor do parâmetro `page` será fixado para `1`. A URL `/blog/2` irá também corresponder, atribuindo ao parâmetro `page` o valor `2`. Perfeito.

/blog	{page} = 1
/blog/1	{page} = 1
/blog/2	{page} = 2

Adicionando Requisitos

Dê uma rápida olhada nos roteamentos que foram criados até agora:

- *YAML*

```
blog:
    pattern:   /blog/{page}
    defaults:  { _controller: AcmeBlogBundle:Blog:index, page: 1 }

blog_show:
    pattern:   /blog/{slug}
    defaults:  { _controller: AcmeBlogBundle:Blog:show }
```

- *XML*

```
<?xml version="1.0" encoding="UTF-8" ?>

<routes xmlns="http://symfony.com/schema/routing"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/routing http://symfony.com/schema/routing/rout

    <route id="blog" pattern="/blog/{page}">
        <default key="_controller">AcmeBlogBundle:Blog:index</default>
        <default key="page">1</default>
    </route>

    <route id="blog_show" pattern="/blog/{slug}">
        <default key="_controller">AcmeBlogBundle:Blog:show</default>
    </route>
</routes>
```

- *PHP*

```
use Symfony\Component\Routing\RouteCollection;
use Symfony\Component\Routing\Route;

$collection = new RouteCollection();
$collection->add('blog', new Route('/blog/{page}', array(
    '_controller' => 'AcmeBlogBundle:Blog:index',
    'page' => 1,
)));

$collection->add('blog_show', new Route('/blog/{show}', array(
    '_controller' => 'AcmeBlogBundle:Blog:show',
)));

return $collection;
```

Você pode apontar o problema ? Perceba que ambas as rotas tem padrão que combinam URL's que pareçam `/blog/*`. O roteador do Symfony irá sempre escolher a **primeira** rota correspondente que ele encontra. Em outras palavras, a rota `blog_show` *nunca* será correspondida. Ao invés disso, uma URL como `/blog/my-blog-post` irá corresponder à primeira rota (`blog`) e retorna um valor sem sentido de `my-blog-post` ao parâmetro `{page}`.

URL	route	parameters
/blog/2	blog	{page} = 2
/blog/my-blog-post	blog	{page} = my-blog-post

A resposta para o problema é adicionar mais *requisitos* de rota. As rotas neste exemplo funcionariam perfeitamente se o padrão `/blog/{page}` *somente* correspondesse a URLs onde a porção `{page}` fosse um integer. Felizmente, requisitos de expressões regulares podem facilmente ser adicionados para cada parâmetro. Por exemplo:

- *YAML*

```
blog:
    pattern:   /blog/{page}
    defaults:  { _controller: AcmeBlogBundle:Blog:index, page: 1 }
    requirements:
        page:  \d+
```

- *XML*

```
<?xml version="1.0" encoding="UTF-8" ?>
```

```
<routes xmlns="http://symfony.com/schema/routing"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/routing http://symfony.com/schema/routing/rout

    <route id="blog" pattern="/blog/{page}">
        <default key="_controller">AcmeBlogBundle:Blog:index</default>
        <default key="page">1</default>
        <requirement key="page">\d+</requirement>
    </route>
</routes>
```

- *PHP*

```
use Symfony\Component\Routing\RouteCollection;
use Symfony\Component\Routing\Route;

$collection = new RouteCollection();
$collection->add('blog', new Route('/blog/{page}', array(
    '_controller' => 'AcmeBlogBundle:Blog:index',
    'page' => 1,
), array(
    'page' => '\d+',
)));

return $collection;
```

O requisito `\d+` é uma expressão regular que diz o valor do parâmetro `{page}` deve ser um dígito (em outras palavras, um número). A rota‘‘blog‘‘ ainda irá corresponder a uma URL como `/blog/2` (porque 2 é um número), mas não irá mair corresponder a URL como `/blog/my-blog-post` (porque `my-blog-post` *não* é um número).

Como resultado, uma URL como `/blog/my-blog-post` não irá corresponder apropriadamente à rota `blog_show`.

URL	rota	parâmetros
/blog/2	blog	{page} = 2
/blog/my-blog-post	blog_show	{slug} = my-blog-post

Rotas prematuras sempre Vencem

Isso tudo significa que a ordem das rotas é muito importante. Se a rota `blog_show` foi colocada acima da rota `blog`, a URL `/blog/2`‘‘corresponderia a ‘‘`blog_show` ao invés de `blog` já que o parâmetro‘‘{slug}‘‘ de `blog_show` não tem requisitos. By using proper ordering and clever requirements, you can accomplish just about anything.

Como os requisitos de parâmetros são expressões regulares, a complexidade e flexibilidade de cada requisito é inteiramente de sua responsabilidade. Suponha que a página inicial de sua aplicação está disponível em dois idiomas diferentes, baseada na URL:

- *YAML*

```
homepage:
    pattern:   /{culture}
    defaults:  { _controller: AcmeDemoBundle:Main:homepage, culture: en }
    requirements:
        culture:  en|fr
```

- *XML*

```xml
<?xml version="1.0" encoding="UTF-8" ?>

<routes xmlns="http://symfony.com/schema/routing"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/routing http://symfony.com/schema/routing/rout

    <route id="homepage" pattern="/{culture}">
        <default key="_controller">AcmeDemoBundle:Main:homepage</default>
        <default key="culture">en</default>
        <requirement key="culture">en|fr</requirement>
    </route>
</routes>
```

- *PHP*

```php
use Symfony\Component\Routing\RouteCollection;
use Symfony\Component\Routing\Route;

$collection = new RouteCollection();
$collection->add('homepage', new Route('/{culture}', array(
    '_controller' => 'AcmeDemoBundle:Main:homepage',
    'culture' => 'en',
), array(
    'culture' => 'en|fr',
)));

return $collection;
```

Para requisições recebidas, a parte `{culture}` da URL é comparada com a expressão regular `(en|fr)`.

/	{culture} = en
/en	{culture} = en
/fr	{culture} = fr
/es	*won't match this route*

Adicionando Requisição de Método HTTP

Em adição à URL, você também pode ajustar o "método" da requisição recebida (em outras palavras, GET, HEAD, POST, PUT, DELETE).Suponha que você tenha um formulário de contato com dois controladores - um para exibir o formulário (em uma requisição GET) e uma para processar o formulário quando ele é enviado (em uma requisição POST). Isto pode ser acompanhando com a seguinte configuração de rota:

- *YAML*

```yaml
contact:
    pattern:  /contact
    defaults: { _controller: AcmeDemoBundle:Main:contact }
    requirements:
        _method:  GET

contact_process:
    pattern:  /contact
    defaults: { _controller: AcmeDemoBundle:Main:contactProcess }
    requirements:
        _method:  POST
```

- *XML*

```xml
<?xml version="1.0" encoding="UTF-8" ?>

<routes xmlns="http://symfony.com/schema/routing"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/routing http://symfony.com/schema/routing/rout

    <route id="contact" pattern="/contact">
        <default key="_controller">AcmeDemoBundle:Main:contact</default>
        <requirement key="_method">GET</requirement>
    </route>

    <route id="contact_process" pattern="/contact">
        <default key="_controller">AcmeDemoBundle:Main:contactProcess</default>
        <requirement key="_method">POST</requirement>
    </route>
</routes>
```

- *PHP*

```php
use Symfony\Component\Routing\RouteCollection;
use Symfony\Component\Routing\Route;

$collection = new RouteCollection();
$collection->add('contact', new Route('/contact', array(
    '_controller' => 'AcmeDemoBundle:Main:contact',
), array(
    '_method' => 'GET',
)));

$collection->add('contact_process', new Route('/contact', array(
    '_controller' => 'AcmeDemoBundle:Main:contactProcess',
), array(
    '_method' => 'POST',
)));

return $collection;
```

Apesar do fato que estas duas rotas tem padrões idênticos (`/contact`), a primeira rota irá aceitar somente requisições GET e a segunda rota irá somente aceitar requisições POST. Isso significa que você pode exibir o formulário e enviar o formulário pela mesma URL, enquanto usa controladores distintos para as duas ações.

Nota: Se nenhum valor `_method` em `requitement` for especificado, a rota irá aceitar todos os metodos.

Como os outros requisitos, o requisito `_method` é analisado como uma expressão regular. Para aceitar requisições `GET` *ou* `POST`, você pode usar `GET|POST`.

Exemplo avançado de roteamento

Até esse ponto, você tem tudo que você precisa para criar uma poderosa estrutura de roteamento em Symfony. O exemplo seguinte mostra quão flexível o sistema de roteamento pode ser:

- *YAML*

```yaml
article_show:
  pattern:  /articles/{culture}/{year}/{title}.{_format}
  defaults: { _controller: AcmeDemoBundle:Article:show, _format: html }
```

```
    requirements:
        culture:  en|fr
        _format:  html|rss
        year:     \d+
```

- *XML*

```
<?xml version="1.0" encoding="UTF-8" ?>

<routes xmlns="http://symfony.com/schema/routing"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/routing http://symfony.com/schema/routing/rout

    <route id="article_show" pattern="/articles/{culture}/{year}/{title}.{_format}">
        <default key="_controller">AcmeDemoBundle:Article:show</default>
        <default key="_format">html</default>
        <requirement key="culture">en|fr</requirement>
        <requirement key="_format">html|rss</requirement>
        <requirement key="year">\d+</requirement>
    </route>
</routes>
```

- *PHP*

```
use Symfony\Component\Routing\RouteCollection;
use Symfony\Component\Routing\Route;

$collection = new RouteCollection();
$collection->add('homepage', new Route('/articles/{culture}/{year}/{title}.{_format}', array(
    '_controller' => 'AcmeDemoBundle:Article:show',
    '_format' => 'html',
), array(
    'culture' => 'en|fr',
    '_format' => 'html|rss',
    'year' => '\d+',
)));

return $collection;
```

Como você viu, essa rota só irá funcionar se a parte `{culture}` da URL ou é `en` ou `fr` e se `{year}` é um número. Esta rota também mostra como você pode usar um período entre espaços reservados ao invés de uma barra. URLs que correspondam a esta rota poderia parecer como:

- `/articles/en/2010/my-post`
- `/articles/fr/2010/my-post.rss`

O Parâmetro de Roteamento Especial `_format`

Esse exemplo também resslta o parâmetro de roteamento especial `_format`. Quando usa esse parâmetro, o valor correspondido se torna o "formato requisitado" do ojeto `Request`. Ultimamente, o formato requisitado é usado para certas coisas como as configurar o `Content-Type` da resposta (ex: um formato de requisição `json` traduz em um `Content-Type` de `application/json`). Ele também pode ser usado no controlador para alterar um template diferente para cada valor de `_format`. O parâmetro `_format` é um modo muito poderoso para alterar o mesmo conteúdo em formatos diferentes.

Parâmetros de Roteamento Especiais

Como você viu, cada parâmetro de roteamento ou valor padrão está eventualmente disponível como um argumento no método do controlador. Adicionalmente, existem três parâmetros que são especiais: cada um adiciona uma parte única de funcionalidade dentro da sua aplicação:

- `_controller`: Como você viu, este parâmetro é usado para determinar qual controlador é executado quando a rota é correspondida;
- `_format`: Usado para fixar o formato de requisição (*read more*);
- `_locale`: Usado para fixar a localidade no pedido (*read more*);

Dica: Se você usar o parâmetro `_locale` na rota, aquele valor será também armazenado na sessão, então, os pedidos posteriores mantêm a mesma localidade.

Padrão de nomeação do Controlador

Cada rota deve ter um parâmetro `_controller`, que ordena qual controlador deveria ser executado quando uma rota é correspondida. Esse parâmetro usa um padrão de string simples chamado *logical controller name*, que o Symfony mapeia para uma classe e método PHP específico. O padrão tem três partes, cada uma separada por dois pontos:

bundle:**controller**:**action**

Por exemplo, um valor `_controller` de `AcmeBlogBundle:Blog:show` significa:

Bundle	Classe do Controlador	Nome do Método
AcmeBlogBundle	BlogController	showAction

O controlador poderia parecer assim:

```
// src/Acme/BlogBundle/Controller/BlogController.php

namespace Acme\BlogBundle\Controller;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class BlogController extends Controller
{
    public function showAction($slug)
    {
        // ...
    }
}
```

Perceba que Symfony adiciona a string `Controller` para o nome da classe (`Blog` => `BlogController`) e `Action` para o nome do método (`show` => `showAction`).

Você também poderia referir a esse controler usando os nomes totalmente qualificados de classe e método:`Acme\BlogBundle\Controller\BlogController::showAction`. Mas se você seguir alguma convenções simples, o nome lógico é mais conciso e permite mais flexibilidade.

Nota: Em complemento ao utilizar o nome lógico ou o nome de classe totalmente qualificado, Symfony suporta um terceiro modo de referir a um controlador. Esse método usa somente um separador de dois pontos (ex: `service_name:indexAction`) e referir um controlador como um serviço (veja Como definir Controladores como Serviços).

Parâmetros de Rota e Argumentos de Controlador

Os parâmetros de rota (ex: `{slug}`) são especialmente importantes porque cada um é disponibilizado como um argumento para o método do controlador:

```
public function showAction($slug)
{
  // ...
}
```

Na realidade, a coleção inteira `defaults` é mesclada com um valor de parâmetro para formar um único array. Cada chave daquele array está disponível como um argumento no controlador.

Em outras palavras, para cada argumento do método do seu controlador, Symfony procura por um parâmetro de rota daquele nome e atribui o valor para aquele argumento. No exemplo avançado acima, qualquer combinação (em qualquer ordem) das seguintes variáveis poderia ser usada como argumentos para o método `showAction()`:

- `$culture`
- `$year`
- `$title`
- `$_format`
- `$_controller`

Como os espaços resercados e a coleção `defaults` são mesclados juntos, mesmo a variável `$_controller` está disponível. Para uma discussão mais detalhada, veja *Parâmetros de Rota como Argumentos do Controlador*.

Dica: Você também pode usar uma variável especial `$_route`, que é fixada para o nome da rota que foi correspondida.

Incluindo Recursos Externos de Roteamento

Todas as rotas são carregadas por um arquivo de configuração individual - geralmente `app/config/routing.yml` (veja *Criando Rotas* acima). É comum, entretanto, que você queria carregar recursos de outros lugares, como um arquivo de roteamento que resida dentro de um pacote. Isso pode ser feito mediante "importação" daquele arquivo:

- *YAML*

```
# app/config/routing.yml
acme_hello:
    resource: "@AcmeHelloBundle/Resources/config/routing.yml"
```

- *XML*

```
<!-- app/config/routing.xml -->
<?xml version="1.0" encoding="UTF-8" ?>

<routes xmlns="http://symfony.com/schema/routing"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/routing http://symfony.com/schema/routing/rout

    <import resource="@AcmeHelloBundle/Resources/config/routing.xml" />
</routes>
```

- *PHP*

```php
// app/config/routing.php
use Symfony\Component\Routing\RouteCollection;

$collection = new RouteCollection();
$collection->addCollection($loader->import("@AcmeHelloBundle/Resources/config/routing.php"));

return $collection;
```

Nota: Quando importar recursos do YAML, a chave (ex: `acme_hello`) é insignificante. Somente esteja certo que é única, então nenhuma outra linha a sobrescreverá.

A chave `resource` carrega o recurso de determinado roteamento. Neste exemplo o recurso é um atalho inteiro para o arquivo, onde a sintaxe do atalho `@AcmeHelloBundle` resolve o atalho daquele pacote. O arquivo importado poderia parecer algo como isso:

- *YAML*

```yaml
 # src/Acme/HelloBundle/Resources/config/routing.yml
acme_hello:
     pattern:  /hello/{name}
     defaults: { _controller: AcmeHelloBundle:Hello:index }
```

- *XML*

```xml
<!-- src/Acme/HelloBundle/Resources/config/routing.xml -->
<?xml version="1.0" encoding="UTF-8" ?>

<routes xmlns="http://symfony.com/schema/routing"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/routing http://symfony.com/schema/routing/rout

    <route id="acme_hello" pattern="/hello/{name}">
        <default key="_controller">AcmeHelloBundle:Hello:index</default>
    </route>
</routes>
```

- *PHP*

```php
// src/Acme/HelloBundle/Resources/config/routing.php
use Symfony\Component\Routing\RouteCollection;
use Symfony\Component\Routing\Route;

$collection = new RouteCollection();
$collection->add('acme_hello', new Route('/hello/{name}', array(
    '_controller' => 'AcmeHelloBundle:Hello:index',
)));

return $collection;
```

As rotas daquele arquivo são analisadas e carregadas da mesma forma que o arquivo principal de roteamento.

Prefixando Rotas Importadas

Você também pode escolher providenciar um "prefixo" para as rotas importadas. Por exemplo suponha que você queira que a rota `acme_hello` tnha um padrão final de `/admin/hello/{name}` ao invés de simplesmente `/hello/{name}`:

- *YAML*

```
# app/config/routing.yml
acme_hello:
    resource: "@AcmeHelloBundle/Resources/config/routing.yml"
    prefix:   /admin
```

- *XML*

```
<!-- app/config/routing.xml -->
<?xml version="1.0" encoding="UTF-8" ?>

<routes xmlns="http://symfony.com/schema/routing"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/routing http://symfony.com/schema/routing/rout

    <import resource="@AcmeHelloBundle/Resources/config/routing.xml" prefix="/admin" />
</routes>
```

- *PHP*

```
// app/config/routing.php
use Symfony\Component\Routing\RouteCollection;

$collection = new RouteCollection();
$collection->addCollection($loader->import("@AcmeHelloBundle/Resources/config/routing.php"), '/a

return $collection;
```

A string `/admin` irá agora ser prefixada ao padrão de cada rota carregada do novo recurso de roteamento.

Visualizando e Depurando Rotas

Enquanto adiciona e personalizar rotas, é útil ser capaz de visualizar e obter informação detalhada sobre suas rotas. Um grande modo para ver cada rota em sua aplicação é pelo comando de console `router:debug`. Execute o seguinte comando a partir da raiz de seu projeto.

```
php app/console router:debug
```

O comando irá imprimir uma lista útil de *todas* as rotas configuradas em sua aplicação:

```
homepage              ANY       /
contact               GET       /contact
contact_process       POST      /contact
article_show          ANY       /articles/{culture}/{year}/{title}.{_format}
blog                  ANY       /blog/{page}
blog_show             ANY       /blog/{slug}
```

Você também pode obter informação muito específica em uma rota individual ao incluir o nome da rota após o comando:

```
php app/console router:debug article_show
```

Novo na versão 2.1: O comando `router:match` foi adicionado no Symfony 2.1

Você pode verificar, se houver, que rota corresponde à um caminho com o comando de console `router:match`:

```
$ php app/console router:match /articles/en/2012/article.rss
Route "article_show" matches
```

Gerando URLs

O sistema de roteamento deveria também ser usado para gerar URLs. Na realidade, roteamento é um sistema bi-direcional: mapeando a URL para um controlador+parâmetros e parâmetros+rota de voltar para a URL. Os métodos `match()` e `generate()` formam esse sistema bi-directional. Considere a rota `blog_show` de um exemplo anterior:

```
$params = $router->match('/blog/my-blog-post');
// array('slug' => 'my-blog-post', '_controller' => 'AcmeBlogBundle:Blog:show')

$uri = $router->generate('blog_show', array('slug' => 'my-blog-post'));
// /blog/my-blog-post
```

Para gerar a URL, você precisa especificar o nome da rota (ex: `blog_show`) e quaisquer coringas(ex: `slug = my-blog-post`) usado no padrão para aquela rota. Com essa informação, qualquer URL pode ser facilmente gerada:

```
class MainController extends Controller
{
    public function showAction($slug)
    {
      // ...

      $url = $this->get('router')->generate('blog_show', array('slug' => 'my-blog-post'));
    }
}
```

Em uma sessão futura, você irá aprender como gerar URLs a partir de templates.

Dica: Se o frontend de sua aplicação usa requisições AJAX, você poderia querer ser capaz de ferar URLs em JavaScript baseados na sua configuração de roteamento. Ao usar FOSJsRoutingBundle, você poderia fazer exatamente isso:

```
var url = Routing.generate('blog_show', { "slug": 'my-blog-post}});
```

Para mais informações, veja a documentação para aquele pacote.

Gerando URLs Absolutas

Por padrão, o roteador irá gerar URLs relativas (ex: `/blog`). Para gerar uma URL absoluta, simplesmente passe `true` ao terceiro argumento do método `generate()`:

```
$router->generate('blog_show', array('slug' => 'my-blog-post'), true);
// http://www.example.com/blog/my-blog-post
```

Nota: O host que é usado quando gera uma URL absoluta é o host do objeto `Request` atual. Isso é detectado automaticamente baseado na informação do servidor abastecida pelo PHP. Quando gerar URLs absolutas para rodar scripts a partir da linha de comando, você precisará fixar manualmente o host no objeto `Request`:

```
$request->headers->set('HOST', 'www.example.com');
```

Gerando URLs com Strings de Consulta

O método `generate` pega um array de valores coringa para gerar a URI. Mas se você passar valores extras, eles serão adicionados ao URI como uma string de consulta:

```
$router->generate('blog', array('page' => 2, 'category' => 'Symfony'));
// /blog/2?category=Symfony
```

Gerando URLs de um template

O lugar mais comum para gerar uma URL é pelo template, ao fazer vinculação entre páginas na sua aplicação.Isso é feito da mesma forma que antes, mas usando uma função helper de template:

- *Twig*

```
<a href="{{ path('blog_show', { 'slug': 'my-blog-post' }) }}">
  Read this blog post.
</a>
```

- *PHP*

```
<a href="<?php echo $view['router']->generate('blog_show', array('slug' => 'my-blog-post')) ?>">
    Read this blog post.
</a>
```

URLs absolutas também podem ser geradas.

- *Twig*

```
<a href="{{ url('blog_show', { 'slug': 'my-blog-post' }) }}">
  Read this blog post.
</a>
```

- *PHP*

```
<a href="<?php echo $view['router']->generate('blog_show', array('slug' => 'my-blog-post'), true
    Read this blog post.
</a>
```

Sumário

Roteamento é um sistema para mapear a URL de requisições recebidas para a função do controlador que deveria ser chamada para processar a requisição. Em ambas permite a você especificar URLs bonitas e manter a funcionalidade de sua aplicação desacoplada daquelas URLs. Roteamento é um mecanismo de duas vias, significando que também deveria ser usada para gerar URLs.

Aprenda mais do Cookbook

- Como forçar as rotas a usar sempre HTTPS ou HTTP

2.1.6 Criando e usando Templates

Como você sabe o controller é responsável por controlar cada requisição que venha de uma aplicação Symfony2. Na realidade, o controller delega muito do trabalho pesado para outros lugares então aquele código pode ser testado e

reusado. Quando um controller precisa gerar HTML, CSS e qualquer outro conteúdo, ele entrega o trabalho para o engine de template. Nesse capítulo, você irá aprender como escrever templates poderosas que podem ser usada para retornar conteúdo para o usuário, preencher corpo de e-mail, e mais. Você irá aprender atalhos, maneiras espertas de extender templates e como reusar código de template.

Templates

Um template é simplesmente um arquivo de texto que pode gerar qualquer formato baseado em texto (HTML, XML, CSV, LaTeX ...). O tipo mais familiar de template é um template em *PHP* - um arquivo de texto analisado pelo PHP que contém uma mistura de texto e código PHP:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Welcome to Symfony!</title>
    </head>
    <body>
        <h1><?php echo $page_title ?></h1>

        <ul id="navigation">
            <?php foreach ($navigation as $item): ?>
                <li>
                    <a href="<?php echo $item->getHref() ?>">
                        <?php echo $item->getCaption() ?>
                    </a>
                </li>
            <?php endforeach; ?>
        </ul>
    </body>
</html>
```

Mas Symfony2 empacota até mesmo uma linguagem muito poderosa de template chamada Twig. Twig permite a você escrever templates consisos e legíveis que são mais amigáveis para web designers e, de certa forma, mais poderosos que templates de PHP:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Welcome to Symfony!</title>
    </head>
    <body>
        <h1>{{ page_title }}</h1>

        <ul id="navigation">
            {% for item in navigation %}
                <li><a href="{{ item.href }}">{{ item.caption }}</a></li>
            {% endfor %}
        </ul>
    </body>
</html>
```

Twig define dois tipos de sintaxe especiais:

- `{{ ... }}`: "Diga algo": exibe uma variável ou o resultado de uma expressão para o template;
- `{% ... %}`: "Faça algo": uma **tag** que controla a lógica do template; ela é usada para executar certas sentenças como for-loops por exemplo.

Nota: Há uma terceira sintaxe usada para criar comentários `{# this is a comment #}`. Essa sintaxe pode ser usada através de múltiplas linhas, parecidas com a sintaxe equivalente em PHP `/* comment */`.

Twig também contém **filtros**, que modificam conteúdo antes de serem interpretados. O seguinte filtro transforma a variável `title` toda em letra maiúscula antes de interpretá-la:

```
{{ title | upper }}
```

Twig vem com uma longa lista de tags e **'filtros'_** que estão disponíveis por padrão. Você pode até mesmo **'adicionar suas próprias extensões'_** para o Twig quando precisar.

Dica: Registrar uma extensão Twig é tão fácil quanto criar um novo serviço e atribuir tag nele com `twig.extension` tag.

Como você verá através da documentação, Twig também suporta funções e nova funções podem ser facilmente adicionadas. Por exemplo, a seguinte função usa uma tag padrão `for` e a função `cycle` para então imprimir dez tags div, alternando entre classes `odd` e `even`:

```
{% for i in 0..10 %}
  <div class="{{ cycle(['odd', 'even'], i) }}">
    <!-- some HTML here -->
  </div>
{% endfor %}
```

Durante este capítulo, exemplos de template serão mostrados tanto em Twig como PHP.

Por que Twig?

Templates Twig são feitas para serem simples e não irão processar tags PHP. Isto é pelo design: o sistema de template do Twig é feito para expressar apresentação, não lógica de programa. Quanto mais você usa Twig, mais você irá apreciar e beneficiar desta distinção. E claro, você será amado por web designers de todos os lugares. Twig pode também fazer coisas que PHP não pode, como por exemplo herança verdadeira de template (Templates do Twig compilam classes PHP que herdam uma da outra), controle de espaço em branco, caixa de areia, e a inclusão de funções personalizadas e filtros que somente afetam templates. Twig contém pequenos recursos que fazem escrita de templates mais fácil e mais concisa. Considere o seguinte exemplo, que combina um loop com uma sentença lógia `if`:

```
<ul>
    {% for user in users %}
        <li>{{ user.username }}</li>
    {% else %}
        <li>No users found</li>
    {% endfor %}
</ul>
```

Cache do Template Twig

Twig é rápido. Cada template Twig é compilado para uma classe nativa PHP que é processada na execução. As classes compiladas são localizadas no diretório `app/cache/{environment}/twig` (onde `{environment}` é o ambiente, como `dev` ou `prod`), e em alguns casos pode ser útil durante a depuração. Veja environments-summary para mais informações de ambientes.

Quando o modo `debug` é habilitado (comum no ambiente `dev`), um template Twig será automaticamente recompilado quando mudanças são feitas nele. Isso significa que durante o desenvolvimento você pode alegremente fazer mudanças para um template Twig e imediatamente ver as mudanças sem precisar se preocupar sobre limpar qualquer cache.

Quando o modo `debug` é desabilitado (comum no ambiente `prod`), entretanto, você deve limpar o cache do diretório Twig para que então os templates Twig se regenerem. Lembre de fazer isso quando distribuir sua aplicação.

Herança e Layouts de Template

Mais frequentemente que não, templates compartilham elementos comuns em um projeto, como o header, footer, sidebar ou outros. Em Symfony2, nós gostamos de pensar sobre esse problema de forma diferente: um template pode ser decorado por outro. Isso funciona exatemente da mesma forma como classes PHP: herança de template permite você construir um "layout" de template base que contenha todos os elementos comuns de seu site definidos como **blocos** (pense em "classe PHP com métodos base"). Um template filho pode extender o layout base e sobrepor os blocos (pense "subclasse PHP que sobreponha certos métodos de sua classe pai").

Primeiro, construa um arquivo de layout de base:

- *Twig*

```
{# app/Resources/views/base.html.twig #}
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
        <title>{% block title %}Test Application{% endblock %}</title>
    </head>
    <body>
        <div id="sidebar">
            {% block sidebar %}
            <ul>
                <li><a href="/">Home</a></li>
                <li><a href="/blog">Blog</a></li>
            </ul>
            {% endblock %}
        </div>

        <div id="content">
            {% block body %}{% endblock %}
        </div>
    </body>
</html>
```

- *PHP*

```
<!-- app/Resources/views/base.html.php -->
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
        <title><?php $view['slots']->output('title', 'Test Application') ?></title>
    </head>
    <body>
        <div id="sidebar">
            <?php if ($view['slots']->has('sidebar')): ?>
                <?php $view['slots']->output('sidebar') ?>
            <?php else: ?>
                <ul>
```

```
                        <li><a href="/">Home</a></li>
                        <li><a href="/blog">Blog</a></li>
                    </ul>
                <?php endif; ?>
            </div>

            <div id="content">
                <?php $view['slots']->output('body') ?>
            </div>
        </body>
    </html>
```

Nota: Apesar da discussão sobre herança de template ser em termos do Twig, a filosofia é a mesma entre templates Twig e PHP.

Este template define o esqueleto do documento base HTML de um página simples de duas colunas. Neste exemplo, três áreas `{% block %}` são definidas (`title`, `sidebar` e `body`). Cada bloco pode ser sobreposto por um template filho ou largado com sua implementação padrão. Esse template poderia também ser processado diretamente. Neste caso os blocos `title`, `sidebar` e `body` blocks deveriam simplesmente reter os valores padrão neste template.

Um template filho poderia ser como este:

- *Twig*

```
{# src/Acme/BlogBundle/Resources/views/Blog/index.html.twig #}
{% extends '::base.html.twig' %}

{% block title %}My cool blog posts{% endblock %}

{% block body %}
    {% for entry in blog_entries %}
        <h2>{{ entry.title }}</h2>
        <p>{{ entry.body }}</p>
    {% endfor %}
{% endblock %}
```

- *PHP*

```
<!-- src/Acme/BlogBundle/Resources/views/Blog/index.html.php -->
<?php $view->extend('::base.html.php') ?>

<?php $view['slots']->set('title', 'My cool blog posts') ?>

<?php $view['slots']->start('body') ?>
    <?php foreach ($blog_entries as $entry): ?>
        <h2><?php echo $entry->getTitle() ?></h2>
        <p><?php echo $entry->getBody() ?></p>
    <?php endforeach; ?>
<?php $view['slots']->stop() ?>
```

Nota: O template pai é idenficado por uma sintaxe especial de string (`::base.html.twig`) que indica que o template reside no diretório `app/Resources/views` do projeto. Essa convenção de nomeamento é explicada inteiramente em *Nomeação de Template e Localizações*.

A chave para herança template é a tag `{% extends %}`. Ela avisa o engine de template para primeiro avaliar o template base, que configura o layout e define vários blocos. O template filho é então processado, ao ponto que os blocos

`title` e `body` do template pai sejam substituídos por aqueles do filho. Dependendo do valor de `blog_entries`, a saída poderia parecer com isso:

```
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
        <title>My cool blog posts</title>
    </head>
    <body>
        <div id="sidebar">
            <ul>
                <li><a href="/">Home</a></li>
                <li><a href="/blog">Blog</a></li>
            </ul>
        </div>

        <div id="content">
            <h2>My first post</h2>
            <p>The body of the first post.</p>

            <h2>Another post</h2>
            <p>The body of the second post.</p>
        </div>
    </body>
</html>
```

Perceba que como o template filho não definiu um bloco `sidebar`, o valor do template pai é usado no lugar. Conteúdo dentro de uma tag `{% block %}` em um template pai é sempre usado por padrão.

Você pode usar muitos níveis de herança quanto quiser. Na próxima sessão, um modelo comum de herança de três níveis será explicado assim como os templates são organizados dentro de um projeto Symfony2.

Quando trabalhar com herança de template, aqui estão algumas dicas para guardar na cabeça:

- Se você usa `{% extends %}` em um template, ele deve ser a primeira tag naquele template.
- Quanto mais tags `{% block %}` você tiver no template base, melhor. Lembre, templates filhos não precisam definir todos os blocos do pai, então criar tantos blocos em seus templates base quanto você quiser e dar a cada um padrão sensato. Quanto mais blocos seus templates base tiverem, mais flexível seu layout será.
- Se você achar você mesmo duplicando conteúdo em um determinado número de templates, isto provavelmente significa que você deveria mover aquele conteúdo para um `{% block %}` no template pai. Em alguns casos, uma solução melhor pode ser mover o conteúdo para um novo template e `incluir` ele (veja *Incluir outras Templates*).
- Se você precisa obter o conteúdo de um bloco do template pai, você pode usar a função `{{ parent() }}`. Isso é útil se você quiser adicionar ao conteúdo de um bloco pai ao invés de sobrepor ele:

```
    .. code-block:: html+jinja

        {% block sidebar %}
            <h3>Table of Contents</h3>
            ...
            {{ parent() }}
        {% endblock %}
```

Nomeação de Template e Localizações

Por padrão, templates podem residir em duas localizações diferentes:

- `app/Resources/views/`: O diretório de aplicação de `views` pode abrigar templates bases para toda a aplicação(ex: os layout de sua aplicação) assim como os tempates que sobrepõem templates de pacote (veja *Sobrepondo Templates de Pacote*);

  application-wide base templates (i.e. your application's layouts) as well as templates that override bundle templates (see *Sobrepondo Templates de Pacote*);

- `path/to/bundle/Resources/views/`: Cada pacote abriga as templates dele no diretório `Resources/views` (e sub-diretórios). A maioria dos templates irá residir dentro de um pacote.

Symfony2 usa a sintaxe de string **bundle**:**controller**:**template** para templates. Isso permite vários tipos diferente de template, cada um residindo em uma localização especifica:

- `AcmeBlogBundle:Blog:index.html.twig`: Esta sintaxe é usada para especificar um template para uma página específica. As três partes do string, cada uma separada por dois pontos, (`:`), signitca o seguinte:
  - `AcmeBlogBundle`: (*bundle*) o template reside entro de `AcmeBlogBundle` (e.g. `src/Acme/BlogBundle`);
  - **`Blog`: (*controller*) indica que o template reside dentro do** sub-diretório `Blog` de `Resources/views`;
  - `index.html.twig`: (*template*) o verdadeiro nome do arquivo é `index.html.twig`.

  Assumindo que o `AcmeBlogBundle` reside em `src/Acme/BlogBundle`, o atalho final para o layout seria `src/Acme/BlogBundle/Resources/views/Blog/index.html.twig`.

- `AcmeBlogBundle::layout.html.twig`: Essa sintaxe refere ao template base que é específica para `AcmeBlogBundle`. Since the middle, "controller", portion is missing (e.g. `Blog`), the template lives at `Resources/views/layout.html.twig` inside `AcmeBlogBundle`.

- `::base.html.twig`: Esta sintaxe refere a uma template base para toda a aplicação ou layout. Perceba que a string começa com dois sinais de dois pontos (`::`), significando que ambas as partes *bundle controller* estão faltando. Isto significa que o template não é localizado em qualquer pacote, mas sim na raiz do diretório `app/Resources/views/`.

Na seção *Sobrepondo Templates de Pacote*, você irá descobrir como cada template reside dentro do `AcmeBlogBundle`, por exemplo, pode ser sobreposto ao colocar um template de mesmo nome no diretório `app/Resources/AcmeBlogBundle/views/`. Isso dá o poder de sobrepor templates de qualquer pacote pago.

Dica: Esperançosamente, a sintaxe de nomeação de template parece familiar - é a mesma convenção para nomeação usada para referir para *Padrão de nomeação do Controlador*.

Sufixo de Template

O formato **bundle**:**controller**:**template** de cada template especifica *onde* o arquivo de template está localizado. Cada nome de template também tem duas expressões que especificam o *formato* e *engine* para aquela template.

- **AcmeBlogBundle:Blog:index.html.twig** - formato HTML, engine Twig
- **AcmeBlogBundle:Blog:index.html.php** - formato HTML, engine PHP
- **AcmeBlogBundle:Blog:index.css.twig** - formato CSS, engine Twig

Por padrão, qualquer template Symfony2 ou pode ser escrito em Twig ou em PHP, e a última parte da extensão (ex: `.twig` ou `.php`) especifica qual dessas duas *engines* deveria ser usada. A primeira parte da extensão, (ex: `.html`, `.css`, etc) é o formato final que o template irá gerar. Ao contrário de engine, que determina como Symfony2 analisa o template, isso é simplesmente uma tática organizacional em caso do mesmo recurso precisar ser transformado como

HTML(`index.html.twig`), XML (`index.xml.twig`), ou qualquer outro formato. Para mais informaçõess, leia a seção *Debugging*.

Nota: As "engines" disponíveis podem ser configurados e até mesmo ter novas engines adicionadas. Veja *Configuração de Template* para mais detalhes.

Tags e Helpers

Você já entende as bases do templates, como eles são chamados e como usar herança de template. As partes mais difíceis estão realmente atrás de você. Nesta seção, você irá aprender sobre um grande grande grupo de ferramentas disponíveis para ajudar a desempenhar as tarefas de template mais comuns como incluir outras templates, vincular páginas e incluir imagens.

Symfony2 vem acompanhado com várias tags Twig especializadas e funções que facilitam o trabalho do designer de template. Em PHP, o sistema de template providencia um sistema extenso de *helper* que providencia funcionalidades úteis no contexto de template.

Nós realmente vimos umas poucas tags Twig construídas (`{% block %}` e `{% extends %}`) como exemplo de um helper PHP (`$view['slots']`). Vamos aprender um pouco mais.

Incluir outras Templates

Você irá frequntemente querer incluir a mesma template ou fragmento de código em várias páginas diferentes. Por exemplo, em uma aplicação com "artigos de notícias", a exibição do artigo no código do template poderia ser usada numa página de detalhes do artigo, num a página exibindo os artigos mais populares, ou em uma lista dos últimos artigos.

Quando você precisa reutilizar um pedaço de um código PHP, você tipicamente move o código para uma nova classe ou função PHP. O mesmo é verdade para template. Ao mover o código do template reutilizado em um template próprio, ele pode ser incluído em qualquer outro template. Primeiro, crie o template que você precisará reutilizar.

- *Twig*

```
{# src/Acme/ArticleBundle/Resources/views/Article/articleDetails.html.twig #}
<h2>{{ article.title }}</h2>
<h3 class="byline">by {{ article.authorName }}</h3>

<p>
  {{ article.body }}
</p>
```

- *PHP*

```
<!-- src/Acme/ArticleBundle/Resources/views/Article/articleDetails.html.php -->
<h2><?php echo $article->getTitle() ?></h2>
<h3 class="byline">by <?php echo $article->getAuthorName() ?></h3>

<p>
  <?php echo $article->getBody() ?>
</p>
```

Incluir este template de qualquer outro template é fácil:

- *Twig*

```
{# src/Acme/ArticleBundle/Resources/Article/list.html.twig #}
{% extends 'AcmeArticleBundle::layout.html.twig' %}

{% block body %}
    <h1>Recent Articles<h1>

    {% for article in articles %}
        {% include 'AcmeArticleBundle:Article:articleDetails.html.twig' with {'article': article
    {% endfor %}
{% endblock %}
```

- *PHP*

```
<!-- src/Acme/ArticleBundle/Resources/Article/list.html.php -->
<?php $view->extend('AcmeArticleBundle::layout.html.php') ?>

<?php $view['slots']->start('body') ?>
    <h1>Recent Articles</h1>

    <?php foreach ($articles as $article): ?>
        <?php echo $view->render('AcmeArticleBundle:Article:articleDetails.html.php', array('art
    <?php endforeach; ?>
<?php $view['slots']->stop() ?>
```

O template é incluído usando a tag `{% include %}`. Perceba que o nome do template segue a mesma convenção típica. O template `articleDetails.html.twig` usa uma variável `article`. Isso é passado por um template `list.html.twig` usando o comando `with`.

Dica: A sintaxe `{'article': article}` é a sintaxe Twig padrão para hash maps (ex: um array com chaves nomeadas). Se nós precisarmos passá-lo em elementos múltiplos, ele poderia ser algo como: `{'foo': foo, 'bar': bar}`.

Incorporação de Controllers

Em alguns casos, você precisa fazer mais do que incluir um template simples. Suponha que você tenha uma barra lateral no seu layout que contenha os três artigos mais recentes. Recuperar os três artigos podem incluir consultar a base de dados ou desempenhar outra lógica pesada que não pode ser a partir de um template.

A solução é simplesmnte incorporar o resultado de um controller inteiro para seu template. Primeiro, crie o controller que retorne um certo número de artigos recentes :

```
// src/Acme/ArticleBundle/Controller/ArticleController.php

class ArticleController extends Controller
{
    public function recentArticlesAction($max = 3)
    {
        // make a database call or other logic to get the "$max" most recent articles
        $articles = ...;

        return $this->render('AcmeArticleBundle:Article:recentList.html.twig', array('articles' => $a
    }
}
```

A template `recentList` é perfeitamente straightforward:

- *Twig*

```
{# src/Acme/ArticleBundle/Resources/views/Article/recentList.html.twig #}
{% for article in articles %}
  <a href="/article/{{ article.slug }}">
      {{ article.title }}
  </a>
{% endfor %}
```

- *PHP*

```
<!-- src/Acme/ArticleBundle/Resources/views/Article/recentList.html.php -->
<?php foreach ($articles as $article): ?>
    <a href="/article/<?php echo $article->getSlug() ?>">
        <?php echo $article->getTitle() ?>
    </a>
<?php endforeach; ?>
```

Nota: Perceba que nós fizemos uma gambiarra e fizemos um hardcode no artigo URL desse exemplo (ex: `/article/*slug*`). Isso é uma prática ruim. Na próxima seção, você irá aprender como fazer isso corretamente.

Para incluir um controller, você irá precisar referir a ela usando a sintaxe de string padrão para controllers (isto é, **bundle**:**controller**:**action**):

- *Twig*

```
{# app/Resources/views/base.html.twig #}
...

<div id="sidebar">
    {% render "AcmeArticleBundle:Article:recentArticles" with {'max': 3} %}
</div>
```

- *PHP*

```
<!-- app/Resources/views/base.html.php -->
...

<div id="sidebar">
    <?php echo $view['actions']->render('AcmeArticleBundle:Article:recentArticles', array('max'
</div>
```

Sempre quando você pensar que você precisa de uma variável ou uma peça de informação que você não tenha acesso em um template, considere transformar o controller. Controllers são rápidos para executar e promovem uma boa organização e utilização do código.

Conteúdo Assíncrono com hinclude.js

Novo na versão 2.1: suporte ao hinclude.js foi adicionado no Symfony 2.1

Os controladores podem ser incorporados assíncronamente usando a biblioteca javascript **hinclude.js_**. Como o conteúdo incorporado vêm de outra página (ou controlador, neste assunto), o Symfony2 usa o helper padrão `render` para configurar tags `hinclude`:

- *Twig*

```
{% render '...:news' with {}, {'standalone': 'js'} %}
```

- *PHP*

```
<?php echo $view['actions']->render('...:news', array(), array('standalone' => 'js')) ?>
```

Nota: **hinclude.js_** nprecisa ser incluído em sua página para funcionar.

Conteúdo padrão (enquanto carregar ou se o javascript está desabilitado) pode ser definido globalmente na configuração da sua aplicação:

- *YAML*

```
# app/config/config.yml
framework:
    # ...
    templating:
        hinclude_default_template: AcmeDemoBundle::hinclude.html.twig
```

- *XML*

```
<!-- app/config/config.xml -->
<framework:config>
    <framework:templating hinclude-default-template="AcmeDemoBundle::hinclude.html.twig" />
</framework:config>
```

- *PHP*

```
// app/config/config.php
$container->loadFromExtension('framework', array(
    // ...
    'templating'      => array(
        'hinclude_default_template' => array('AcmeDemoBundle::hinclude.html.twig'),
    ),
));
```

Vinculação às Páginas

Criar links para outras página em sua aplicação é uma das tarefas mais comuns para um template. Ao invés de fazer um hardcode nas URLs nos templates, use a função do Twig `path` (ou o helper `router` no PHP) para gerar URLs baseadas na configuração de roteamento. Mais tarde, se você quiser modificar a URL de uma página particular, tudo que você precisará fazer é mudar as configurações de roteamento; os templates irão automatricamente gerar a nova URL.

Primeiro, vincule a página "_welcome" , que é acessível pela seguinte configuração de roteamento:

- *YAML*

```
_welcome:
    pattern:  /
    defaults: { _controller: AcmeDemoBundle:Welcome:index }
```

- *XML*

```
<route id="_welcome" pattern="/">
    <default key="_controller">AcmeDemoBundle:Welcome:index</default>
</route>
```

- *PHP*

```
$collection = new RouteCollection();
$collection->add('_welcome', new Route('/', array(
    '_controller' => 'AcmeDemoBundle:Welcome:index',
)));

return $collection;
```

Para vincular à página, apenas use a função Twig `path` e refira para a rota:

- *Twig*

```
<a href="{{ path('_welcome') }}">Home</a>
```

- *PHP*

```
<a href="<?php echo $view['router']->generate('_welcome') ?>">Home</a>
```

Como esperado, isso irá gerar a URL `/`. Vamos ver como isso irá funcionar com uma rota mais complicada:

- *YAML*

```
article_show:
    pattern:  /article/{slug}
    defaults: { _controller: AcmeArticleBundle:Article:show }
```

- *XML*

```
<route id="article_show" pattern="/article/{slug}">
    <default key="_controller">AcmeArticleBundle:Article:show</default>
</route>
```

- *PHP*

```
$collection = new RouteCollection();
$collection->add('article_show', new Route('/article/{slug}', array(
    '_controller' => 'AcmeArticleBundle:Article:show',
)));

return $collection;
```

Neste caso, você precisa especificar tanto o nome da rota (`article_show`) como um valor para o parâmetro `{slug}`. Usando esta rota, vamos revisitar o template `recentList` da sessão anterior e vincular aos artigos corretamente:

- *Twig*

```
{# src/Acme/ArticleBundle/Resources/views/Article/recentList.html.twig #}
{% for article in articles %}
  <a href="{{ path('article_show', { 'slug': article.slug }) }}">
      {{ article.title }}
  </a>
{% endfor %}
```

- *PHP*

```
<!-- src/Acme/ArticleBundle/Resources/views/Article/recentList.html.php -->
<?php foreach ($articles in $article): ?>
    <a href="<?php echo $view['router']->generate('article_show', array('slug' => $article->getS
        <?php echo $article->getTitle() ?>
```

```
        </a>
    <?php endforeach; ?>
```

Dica: Você também pode gerar uma URL absoluta ao usar a função `url` do Twig:

```
<a href="{{ url('_welcome') }}">Home</a>
```

O mesmo pode ser feito em templates PHP ao passar um terceiro argumento ao método `generate()`:

```
<a href="<?php echo $view['router']->generate('_welcome', array(), true) ?>">Home</a>
```

Vinculando os Assets

Templates podem frequentemente referir a imagens, Javascript, folhas de estilo e outros recursos. Claro você poderia fazer um hardcode do atalho desses assets (ex: `/images/logo.png`), mas Symfony2 providencia uma opção mais dinâmica via função `assets` do Twig:

- *Twig*

```
    <img src="{{ asset('images/logo.png') }}" alt="Symfony!" />

    <link href="{{ asset('css/blog.css') }}" rel="stylesheet" type="text/css" />
```

- *PHP*

```
    <img src="<?php echo $view['assets']->getUrl('images/logo.png') ?>" alt="Symfony!" />

    <link href="<?php echo $view['assets']->getUrl('css/blog.css') ?>" rel="stylesheet" type="text/c
```

O principal propósito da função `asset` é tornar sua aplicação mais portátil. Se sua aplicação reside na raiz do seu host (ex: http://example.com), então os atalhos interpretados deveriam ser `/images/logo.png`. Mas se sua aplicação reside em um sub-diretório (ex: http://example.com/my_app), cada caminho do asset deveria interpretar com o diretório (e.g. `/my_app/images/logo.png`). A função `asset` toma conta disto ao determinar como sua aplicação está sendo usada e gerando os atalhos de acordo com o correto.

Adicionalmente, se você usar função asset, Symfony pode automaticamente anexar uma string de consulta para asset, em detrimento de garantir que assets estáticos atualizados não serão armazenados quando distribuídos. Por exemplo, `/images/logo.png` poderia parecer como `/images/logo.png?v2`. Para mais informações, veja a opção de configuração ref-framework-assets-version.

Incluindo Folhas de Estilo e Javascript no Twig

Nenhum site seria completo sem incluir arquivos Javascript e folhas de estilo. Em Symfony, a inclusão desses assets é elegantemente manipulada ao tirar vantagem das heranças de template do Symfony.

Dica: Esta seção irá ensinar você a filosofia por trás disto, incluindo folha de estilo e asset Javascript em Symfony. Symfony também engloba outra biblioteca, chamada Assetic, que segue essa filosofia mas também permite você fazer mais coisas muito interessantes com esses assets. Para mais informações sobre usar Assetic veja Como usar o Assetic para o Gerenciamento de Assets.

Comece adicionando dois blocos a seu template base que irá abrigar seus assets: uma chamada `stylesheets` dentro da tag `head` e outra chamada `javascripts` justamente acima do fechamento da tag `body`. Esses blocos irão conter todas as folhas de estilo e Javascripts que você irá precisar através do seu site:

```
{# 'app/Resources/views/base.html.twig' #}
<html>
    <head>
        {# ... #}

        {% block stylesheets %}
            <link href="{{ asset('/css/main.css') }}" type="text/css" rel="stylesheet" />
        {% endblock %}
    </head>
    <body>
        {# ... #}

        {% block javascripts %}
            <script src="{{ asset('/js/main.js') }}" type="text/javascript"></script>
        {% endblock %}
    </body>
</html>
```

Isso é fácil o bastante ! Mas e se você precisar incluir uma folha de estilo ou Javascript de um template filho ? Por exemplo, suponha que você tenha uma página de contatos e você precise incluir uma folha de estilo `contact.css` *bem* naquela página. Dentro do template da página de contatos, faça o seguinte:

```
{# src/Acme/DemoBundle/Resources/views/Contact/contact.html.twig #}
{# extends '::base.html.twig' #}

{% block stylesheets %}
    {{ parent() }}

    <link href="{{ asset('/css/contact.css') }}" type="text/css" rel="stylesheet" />
{% endblock %}

{# ... #}
```

No template filho, você simplesmente sobrepõe o bloco `stylesheets` e coloca sua nova tag de folha de estilo dentro daquele bloco. Claro, desde que você queira adicionar ao conteúdo do bloco pai (e realmente não irá *substituí-lo), você deveria usar a função `parent()` do Twig function para incluir tudo do bloco `stylesheets` do template base.

Você pode também incluir assets localizados em seus arquivos de pacotes `Resources/public`. Você precisará executar o comando‘‘php app/console assets:install target [–symlink]‘‘ , que move (ou symlinks) arquivos dentro da localização correta. (target é sempre por padrão “web).

```
<link href="{{ asset('bundles/acmedemo/css/contact.css') }}" type="text/css" rel="stylesheet" />
```

O resultado final é uma página que inclui ambas as folhas de estilo `main.css` e `contact.css`.

Configurando e usando o Serviço `templating`

O coração do sistema de template em Symfony2 é o template `Engine`. Este objeto especial é responsável por manipular templates e retornar o conteúdo deles. Quando você manipula um template em um controller, por exemplo, você está na verdade usando o serviço do template engine. Por exemplo:

```
return $this->render('AcmeArticleBundle:Article:index.html.twig');
```

é equivalente a:

```
$engine = $this->container->get('templating');
$content = $engine->render('AcmeArticleBundle:Article:index.html.twig');
```

```
return $response = new Response($content);
```

O engine de template (ou "serviço") é pré-configurada para trabalhar automaticamente dentro de Symfony2. Ele pode, claro, ser configurado mais adiante no arquivo de configuração da aplicação:

- *YAML*

```
# app/config/config.yml
framework:
    # ...
    templating: { engines: ['twig'] }
```

- *XML*

```
<!-- app/config/config.xml -->
<framework:templating>
    <framework:engine id="twig" />
</framework:templating>
```

- *PHP*

```
// app/config/config.php
$container->loadFromExtension('framework', array(
    // ...
    'templating'      => array(
        'engines' => array('twig'),
    ),
));
```

Várias opções de configuração estão disponíveis e estão cobertos em `Configuration Appendix`.

Nota: O engine `twig` é obrigatório para usar o webprofiler (bem como outros pacotes de terceiros).

Sobrepondo Templates de Pacote

A comunidade Symfony2 orgulha-se de si própria em criar e manter pacotes de alta qualidade (veja Symfony2Bundles.org) para um grande número de funcionalidades diferentes. Uma vez que você use um pacote de terceiros, você irá certamente precisar sobrepor e personalizar um ou mais de seus templates.

Suponha que você incluiu o imaginário open-source `AcmeBlogBundle` em seu projeto (ex: no diretório `src/Acme/BlogBundle`). E enquanto você estiver realmente feliz com tudo, você quer sobrepor a página de "lista" do blog para personalizar a marcação especificamente para sua aplicação. Ao se aprofundar no controller `Blog` eo `AcmeBlogBundle`, você encontrará o seguinte:

```
public function indexAction()
{
    $blogs = // some logic to retrieve the blogs

    $this->render('AcmeBlogBundle:Blog:index.html.twig', array('blogs' => $blogs));
}
```

Quando `AcmeBlogBundle:Blog:index.html.twig` é manipulado, Symfony2 realmente observa duas diferentes localizações para o template:

1. `app/Resources/AcmeBlogBundle/views/Blog/index.html.twig`
2. `src/Acme/BlogBundle/Resources/views/Blog/index.html.twig`

Para sobrepor o template de pacote, só copie o template `index.html.twig` do pacote para `app/Resources/AcmeBlogBundle/views/Blog/index.html.twig` (o diretório `app/Resources/AcmeBlogBundle` não existirão, então você precisará criá-lo). Você está livre agora para personalizar o template.

Esta lógica também se aplica a templates de pacote base. Suponha também que cada template em `AcmeBlogBundle` herda de um template base chamado `AcmeBlogBundle::layout.html.twig`. Justo como antes, Symfony2 irá observar os seguintes dois lugares para o template:

1. `app/Resources/AcmeBlogBundle/views/layout.html.twig`
2. `src/Acme/BlogBundle/Resources/views/layout.html.twig`

Uma vez novamente, para sobrepor o template, apenas copie ele para `app/Resources/AcmeBlogBundle/views/layout.html.twig`. Você agora está livre para personalizar esta cópia como você quiser.

Se você voltar um passo atrás, verá que Symfony2 sempre começa a observar no diretório `app/Resources/{BUNDLE_NAME}/views/` por um template. Se o template não existe aqui, ele continua checando dentro do diretório `Resources/views` do próprio pacote. Isso significa que todos os templates do pacote podem ser sobrepostos ao colocá-los no sub-diretório correto `app/Resources`.

Sobrepondo Templates Centrais

Como o framework Symfony é um pacote por si só, templates centrais podem ser sobrepostos da mesma forma. Por exemplo, o núcleo `TwigBundle` contém um númeto de diferentes templates "exception" e "error" que podem ser sobrepostas ao copiar cada uma do diretório `Resources/views/Exception` do `TwigBundle` para, você adivinhou, o diretório `app/Resources/TwigBundle/views/Exception`.

Herança de Três Níveis

Um modo comum de usar herança é usar uma aproximação em três níveis. Este método trabalha perfeitamente com três tipos diferentes de templates que nós certamente cobrimos:

- Criar um arquivo `app/Resources/views/base.html.twig` que contém o layout principal para sua aplicação (como nos exemplos anteriores). Internamente, este template é chamado `::base.html.twig`;
- Cria um template para cada "seção" do seu site. Por exemplo, um `AcmeBlogBundle`, teria um template chamado `AcmeBlogBundle::layout.html.twig` que contém somente elementos específicos para a sessão no blog:

```
{# src/Acme/BlogBundle/Resources/views/layout.html.twig #}
{% extends '::base.html.twig' %}

{% block body %}
    <h1>Blog Application</h1>

    {% block content %}{% endblock %}
{% endblock %}
```

- Criar templates individuais para cada página e fazer cada um estender a template de sessão apropriada. Por exemplo, a página "index" deveria ser chamada de algo próximo a `AcmeBlogBundle:Blog:index.html.twig` e listar os blogs de posts reais.

```
{# src/Acme/BlogBundle/Resources/views/Blog/index.html.twig #}
{% extends 'AcmeBlogBundle::layout.html.twig' %}
```

```
{% block content %}
    {% for entry in blog_entries %}
        <h2>{{ entry.title }}</h2>
        <p>{{ entry.body }}</p>
    {% endfor %}
{% endblock %}
```

Perceba que este template estende a template de sessão - (`AcmeBlogBundle::layout.html.twig`) que por sua vez estende o layout de aplicação base (`::base.html.twig`). Isso é o modelo comum de herança de três níveis.

Quando construir sua aplicação, você pode escolher seguir esse método ou simplesmente tornar cada template de página estender a template de aplicação base diretamente (ex: `{% extends '::base.html.twig' %}`). O modelo de três templates é o método de melhor prática usado por vendor bundles então aquele template base para um pacote pode ser facilmente sobreposto para propriamente estender seu layout base de aplicação.

Saída para escape

Quando gerar HTML de um template, sempre há um risco que uma variável de template pode gerar HTML involutário ou codigo do lado cliente perigoso. O resultado é que o conteúdo dinâmico poderia quebrar o HTML de uma página de resultados ou permitir um usuário maldoso realizar um ataque Cross Site Scripting (XSS). Considere esse exemplo clássico:

- *Twig*

```
Hello {{ name }}
```

- *PHP*

```
Hello <?php echo $name ?>
```

Imagine que o usuário entre o seguinte código como o nome dele/dela:

```
<script>alert('hello!')</script>
```

Sem qualquer outra saída de escape, o resultado da template irá causar uma caixa de alerta em JavaScript para saltar na tela:

```
Hello <script>alert('hello!')</script>
```

E enquanto isso parece inofensivo, se um usuário pode chegar tão longe, o mesmo usuário deveria também ser capaz de escrever Javascript que realiza ações maliciosas dentro de uma área segura de um usuário legítimo e desconhecido.

A resposta para o problema é saída para escape. Sem a saída para escape ativa, o mesmo template irá manipular inofensivamente, e literalmente imprimir a tag `script` na tela:

```
Hello &lt;script&gt;alert(&#39;helloe&#39;)&lt;/script&gt;
```

Os sistemas de templating Twig e PHP aproximam-se do problema de formas diferentes. Se você está usando Twig, saída para escape é ativado por padrão e você está protegido. Em PHP, saída para escape não é automático, significando que você precisará manualmente fazer o escape quando necessário.

Saída para escape em Twig

Se você está usando templates Twig, então saída para escape é ativado por padrão. Isto significa que você está protegido externamente de consequencias acidentais por código submetido por usuário. Por padrão, a saída para escape assume que o conteúdo está sendo escapado pela saída HTML.

Em alguns casos, você precisará desabilitar saída para escape quando você está manipulando uma variável que é confiável e contém marcação que não poderia ter escape. Suponha que usuários administrativos são capazes de escrever artigos que contenham código HTML. Por padrão, Twig irá escapar o corpo do artigo. Para fazê-lo normalamente, adicione o filtro `raw`: `{{ article.body | raw }}`.

Você pode também desabilitar saída para escape dentro de uma área `{% block %}` ou para um template inteiro. Para mais informações, veja Output Escaping na documentação do Twig.

Saída para escape em PHP

Saída para escape não é automática quando usamos templates PHP. Isso significa que a menos que você escolha escapar uma variável explicitamente, você não está protegido. Para usar saída para escape use o método de view `escape()`:

```
Hello <?php echo $view->escape($name) ?>
```

Por padrão, o método `escape()` assume que a variável está sendo manipulada dentro de um contexto HTML (e assim a variável escapa e está segura para o HTML). O segundo argumento deixa você mudar o contexto. Por exemplo, para gerar algo em uma string Javascript, use o contexto `js` :

```
var myMsg = 'Hello <?php echo $view->escape($name, 'js') ?>';
```

Debugging

Novo na versão 2.0.9: Esta funcionalidade está disponível no Twig `1.5.x`, e foi adicionada primeiramente no Symfony 2.0.9.

Ao utilizar o PHP, você pode usar o `var_dump()` se precisa encontrar rapidamente o valor de uma variável passada. Isso é útil, por exemplo, dentro de seu controlador. O mesmo pode ser conseguido ao usar o Twig com a extensão de depuração. Esta extensão precisa ser ativada na configuração:

- *YAML*

```
# app/config/config.yml
services:
    acme_hello.twig.extension.debug:
        class:        Twig_Extension_Debug
        tags:
            - { name: 'twig.extension' }
```

- *XML*

```
<!-- app/config/config.xml -->
<services>
    <service id="acme_hello.twig.extension.debug" class="Twig_Extension_Debug">
        <tag name="twig.extension" />
    </service>
</services>
```

- *PHP*

```
// app/config/config.php
use Symfony\Component\DependencyInjection\Definition;

$definition = new Definition('Twig_Extension_Debug');
$definition->addTag('twig.extension');
$container->setDefinition('acme_hello.twig.extension.debug', $definition);
```

O dump dos parâmetros do template pode ser feito usando a função `dump`:

```
{# src/Acme/ArticleBundle/Resources/views/Article/recentList.html.twig #}
{{ dump(articles) }}

{% for article in articles %}
    <a href="/article/{{ article.slug }}">
        {{ article.title }}
    </a>
{% endfor %}
```

O dump das variáveis somente será realizado se a definição `debug` do Twig (no `config.yml`) for `true`. Por padrão, isto signifca que será feito o dump das variáveis no ambiente `dev` mas não no `prod`.

Verificação de Sintaxe

Novo na versão 2.1: O comando `twig:lint` foi adicionado no Symfony 2.1

Você pode verificar erros de sintaxe nos templates do Twig usando o comando de console `twig:lint`:

```
# You can check by filename:
$ php app/console twig:lint src/Acme/ArticleBundle/Resources/views/Article/recentList.html.twig

# or by directory:
$ php app/console twig:lint src/Acme/ArticleBundle/Resources/views

# or using the bundle name:
$ php app/console twig:lint @AcmeArticleBundle
```

Formatos de Template

Templates são uma forma genérica de modificar conteúdo em *qualquer* formato. E enquanto em muitos casos você irá usar templates para modificar conteúdo HTML, um template pode tão fácil como certo gerar JavaScript, CSS, XML ou qualquer outro formato que você possa sonhar.

Por exemplo, o mesmo "recurso" é sempre modificado em diversos formatos diferentes. Para modificar uma página inicial de um artigo XML, simplesmente inclua o formato no nome do template:

- *nome do template XML*: `AcmeArticleBundle:Article:index.xml.twig`
- *nome do arquivo do template XML*: `index.xml.twig`

Na realidade, isso é nada mais que uma convenção de nomeação e o template não é realmente modificado de forma diferente ao baseado no formato dele.

Em muitos casos, você pode querer permitir um controller unitário para modificar múltiplos formatos diferentes baseado no "formato de requisição". Por aquela razão, um padrão comum é fazer o seguinte:

```
public function indexAction()
{
    $format = $this->getRequest()->getRequestFormat();

    return $this->render('AcmeBlogBundle:Blog:index.'.$format.'.twig');
}
```

O `getRequestFormat` no objeto `Request` padroniza para `html`, mas pode retornar qualquer outro formato baseado no formato solicitado pelo usuário. O formato solicitado é frequentemente mais gerenciado pelo roteamento, onde uma rota pode ser configurada para que `/contact` configure o formato requisitado `html` enquanto `/contact.xml` configure o formato para `xml`. Para mais informações, veja *Advanced Example in the Routing chapter*.

Para criar links que incluam o parâmetro de formato, inclua uma chave `_format` no detalhe do parâmetro:

- *Twig*

```
<a href="{{ path('article_show', {'id': 123, '_format': 'pdf'}) }}">
    PDF Version
</a>
```

- *PHP*

```
<a href="<?php echo $view['router']->generate('article_show', array('id' => 123, '_format' => 'p
    PDF Version
</a>
```

Considerações finais

O engine de template em Symfony é uma ferramenta poderosa que pode ser usada cada momento que você precisa para gerar conteúdo de apresentação em HTML, XML ou qualquer outro formato. E apesar de tempaltes serem um jeito comum de gerar conteúdo em um controller, o uso deles não são obrigatórios. O objeto `Response` object retornado por um controller pode ser criado com ou sem o uso de um template:

```
// creates a Response object whose content is the rendered template
$response = $this->render('AcmeArticleBundle:Article:index.html.twig');

// creates a Response object whose content is simple text
$response = new Response('response content');
```

Engine de template do Symfony é muito flexível e dois editores de template diferentes estão disponíveis por padrão: os tradicionais templates do *PHP* e os polidos e poderosos templates do *Twig* . Ambos suportam uma hierarquia de template e vêm empacotados com um conjunto rico de funções helper capazes de realizar as tarefas mais comuns.

No geral, o tópico de template poderia ser pensado como uma ferramenta poderosa que está à sua disposição. Em alguns casos, você pode não precisar modificar um template, e em Symfony2, isso é absolutamente legal.

Aprenda mais do Cookbook

- `/cookbook/templating/PHP`
- Como personalizar as páginas de erro

2.1.7 Bancos de Dados e Doctrine

Temos que dizer, uma das tarefas mais comuns e desafiadoras em qualquer aplicação envolve persistir e ler informações de um banco de dados. Felizmente o Symfony vem integrado com o Doctrine, uma biblioteca cujo único objetivo é fornecer poderosas ferramentas que tornem isso fácil. Nesse capítulo você aprenderá a filosofia básica por trás do Doctrine e verá quão fácil pode ser trabalhar com um banco de dados.

Nota: O Doctrine é totalmente desacoplado do Symfony, e seu uso é opcional. Esse capítulo é totalmente sobre o Doctrine ORM, que visa permitir fazer mapeamento de objetos para um banco de dados relacional (como o *MySQL*, *PostgreSQL* ou o *Microsoft SQL*). É fácil usar consultas SQL puras se você preferir, isso é explicado na entrada do cookbook "Como usar a Camada DBAL do Doctrine".

Você também pode persistir dados no MongoDB usando a biblioteca Doctrine ODM. Para mais informações, leia a documentação "`/bundles/DoctrineMongoDBBundle/index`".

Um Exemplo Simples: Um Produto

O jeito mais fácil de entender como o Doctrine trabalha é vendo-o em ação. Nessa seção, você configurará seu banco de dados, criará um objeto `Product`, fará sua persistência no banco e depois irá retorná-lo.

Codifique seguindo o exemplo

Se quiser seguir o exemplo deste capítulo, crie um `AcmeStoreBundle` via:

```
php app/console generate:bundle --namespace=Acme/StoreBundle
```

Configurando o Banco de Dados

Antes de começar realmente, você precisa configurar a informação de conexão do seu banco. Por convenção, essa informação geralmente é configurada no arquivo `app/config/parameters.yml`:

```
# app/config/parameters.yml
parameters:
    database_driver:   pdo_mysql
    database_host:     localhost
    database_name:     test_project
    database_user:     root
    database_password: password
```

Nota: Definir a configuração pelo `parameters.yml` é apenas uma convenção. Os parâmetros definidos naquele arquivo são referenciados pelo arquivo de configuração principal na configuração do Doctrine:

```
doctrine:
    dbal:
        driver:   %database_driver%
        host:     %database_host%
        dbname:   %database_name%
        user:     %database_user%
        password: %database_password%
```

Colocando a informação do banco de dados em um arquivo separado, você pode manter de forma fácil diferentes versões em cada um dos servidores. Você pode também guardar facilmente a configuração de banco (ou qualquer outra informação delicada) fora do seu projeto, por exemplo dentro do seu diretório de configuração do Apache. Para mais informações, de uma olhada em Como definir Parâmetros Externos no Container de Serviços.

Agora que o Doctrine sabe sobre seu banco, pode deixar que ele faça a criação dele para você:

```
php app/console doctrine:database:create
```

Criando uma Classe Entidade

Suponha que você esteja criando uma aplicação onde os produtos precisam ser mostrados. Antes mesmo de pensar sobre o Doctrine ou banco de dados, você já sabe que irá precisar de um objeto `Product` para representar esses produtos. Crie essa classe dentro do diretório `Entity` no seu bundle `AcmeStoreBundle`:

```
// src/Acme/StoreBundle/Entity/Product.php
namespace Acme\StoreBundle\Entity;
```

```
class Product
{
    protected $name;

    protected $price;

    protected $description;
}
```

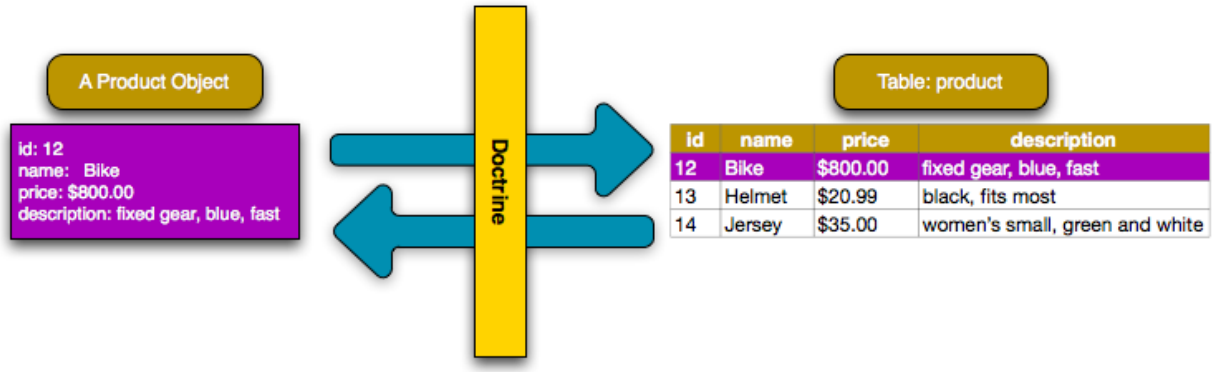
A classe - frequentemente chamada de "entidade", que significa *uma classe básica para guardar dados* - é simples e ajuda a cumprir o requisito de negócio referente aos produtos na sua aplicação. Essa classe ainda não pode ser persistida no banco de dados - ela é apenas uma classe PHP simples.

Dica: Depois que você aprender os conceitos por trás do Doctrine, você pode deixá-lo criar essa classe entidade para você:

```
php app/console doctrine:generate:entity --entity="AcmeStoreBundle:Product" --fields="name:string(25
```

Adicionando Informações de Mapeamento

O Doctrine permite que você trabalhe de uma forma muito mais interessante com banco de dados do que apenas buscar registros de uma tabela baseada em colunas para um array. Em vez disso, o Doctrine permite que você persista *objetos* inteiros no banco e recupere objetos inteiros do banco de dados. Isso funciona mapeando uma classe PHP com uma tabela do banco, e as propriedades dessa classe com as colunas da tabela:



Para o Doctrine ser capaz disso, você tem apenas que criar "metadados", em outras palavras a configuração que diz ao Doctrine exatamente como a classe `Product` e suas propriedades devem ser *mapeadas* com o banco de dados. Esses metadados podem ser especificados em vários diferentes formatos incluindo YAML, XML ou diretamente dentro da classe `Product` por meio de annotations:

Nota: Um bundle só pode aceitar um formato para definição de metadados. Por exemplo, não é possível misturar definições em YAML com definições por annotations nas classes entidade.

- *Annotations*

```
    // src/Acme/StoreBundle/Entity/Product.php
    namespace Acme\StoreBundle\Entity;
```

```php
use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity
 * @ORM\Table(name="product")
 */
class Product
{
    /**
     * @ORM\Id
     * @ORM\Column(type="integer")
     * @ORM\GeneratedValue(strategy="AUTO")
     */
    protected $id;

    /**
     * @ORM\Column(type="string", length=100)
     */
    protected $name;

    /**
     * @ORM\Column(type="decimal", scale=2)
     */
    protected $price;

    /**
     * @ORM\Column(type="text")
     */
    protected $description;
}
```

- *YAML*

```yaml
# src/Acme/StoreBundle/Resources/config/doctrine/Product.orm.yml
Acme\StoreBundle\Entity\Product:
    type: entity
    table: product
    id:
        id:
            type: integer
            generator: { strategy: AUTO }
    fields:
        name:
            type: string
            length: 100
        price:
            type: decimal
            scale: 2
        description:
            type: text
```

- *XML*

```xml
<!-- src/Acme/StoreBundle/Resources/config/doctrine/Product.orm.xml -->
<doctrine-mapping xmlns="http://doctrine-project.org/schemas/orm/doctrine-mapping"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://doctrine-project.org/schemas/orm/doctrine-mapping
                  http://doctrine-project.org/schemas/orm/doctrine-mapping.xsd">
```

```
        <entity name="Acme\StoreBundle\Entity\Product" table="product">
            <id name="id" type="integer" column="id">
                <generator strategy="AUTO" />
            </id>
            <field name="name" column="name" type="string" length="100" />
            <field name="price" column="price" type="decimal" scale="2" />
            <field name="description" column="description" type="text" />
        </entity>
    </doctrine-mapping>
```

Dica: O nome da tabela é opcional e, se omitido, será determinado automaticamente baseado no nome da classe entidade.

O Doctrine permite que você escolha entre uma grande variedade de diferentes tipos de campo, cada um com suas opções específicas. Para informações sobre os tipos de campos disponíveis, dê uma olhada na seção *Referência dos Tipos de Campos do Doctrine*.

Veja também:

Você também pode conferir a Documentação Básica sobre Mapeamento do Doctrine para todos os detalhes sobre o tema. Se você usar annotations, irá precisar prefixar todas elas com `ORM\` (i.e. `ORM\Column(..)`), o que não é citado na documentação do Doctrine. Você também irá precisar incluir o comando `use Doctrine\ORM\Mapping as ORM;`, que *importa* o prefixo `ORM` das annotations.

Cuidado: Tenha cuidado para que o nome da sua classe e suas propriedades não estão mapeadas com o nome de um comando SQL protegido (como `group``ou ``user`). Por exemplo, se o nome da sua classe entidade é `Group` então, por padrão, o nome da sua tabela será `group`, que causará um erro de SQL em alguns dos bancos de dados. Dê uma olhada na Documentação sobre os nomes de comandos SQL reservados para ver como escapar adequadamente esses nomes. Alternativamente, se você pode escolher livremente seu esquema de banco de dados, simplesmente mapeie para um nome de tabela ou nome de coluna diferente. Veja a documentação do Doctrine sobre Classes persistentes e Mapeamento de propriedades

Nota: Quando usar outra biblioteca ou programa (i.e. Doxygen) que usa annotations você dever colocar a annotation `@IgnoreAnnotation` na classe para indicar que annotations o Symfony deve ignorar.

Por exemplo, para prevenir que a annotation `@fn` gere uma exceção, inclua o seguinte:

```
/**
    • @IgnoreAnnotation("fn")
*/
class Product
```

Gerando os Getters e Setters

Apesar do Doctrine agora saber como persistir um objeto `Product` num banco de dados, a classe ainda não é realmente útil. Como `Product` é apenas uma classe PHP usual, você precisa criar os métodos getters e setters (i.e. `getName()`, `setName()` para acessar sua suas propriedades (até as propriedades `protected`). Felizmente o Doctrine pode fazer isso por você executando:

```
php app/console doctrine:generate:entities Acme/StoreBundle/Entity/Product
```

Esse comando garante que todos os getters e setters estão criados na classe `Product`. Ele é um comando seguro - você pode executá-lo muitas e muitas vezes: ele apenas gera getters e setters que ainda não existem (i.e. ele não altera os models já existentes).

Cuidado: O comando `doctrine:generate:entities` gera um backup do `Product.php` original chamado de ‘‘Product.php~‘. Em alguns casos, a presença desse arquivo pode causar um erro “Cannot redeclare class‘. É seguro removê-lo.

Você pode gerar todas as entidades que são conhecidas por um bundle (i.e. cada classe PHP com a informação de mapeamento do Doctrine) ou de um namespace inteiro.

```
php app/console doctrine:generate:entities AcmeStoreBundle
php app/console doctrine:generate:entities Acme
```

Nota: O Doctrine não se importa se as suas propriedades são `protected` ou `private`, ou se você não tem um método getter ou setter. Os getters e setters são gerados aqui apenas porque você irá precisar deles para interagir com o seu objeto PHP.

Criando as Tabelas/Esquema do Banco de Dados

Agora você tem uma classe utilizável `Product` com informação de mapeamento assim o Doctrine sabe exatamente como fazer a persistência dela. É claro, você ainda não tem a tabela correspondente `product` no seu banco de dados. Felizmente, o Doctrine pode criar automaticamente todas as tabelas necessárias no banco para cada uma das entidades conhecidas da sua aplicação. Para isso, execute:

```
php app/console doctrine:schema:update --force
```

Dica: Na verdade, esse comando é extremamente poderoso. Ele compara o que o banco de dados *deveria* se parecer (baseado na informação de mapeamento das suas entidades) com o que ele *realmente* se parece, e gera os comandos SQL necessários para *atualizar* o banco para o que ele deveria ser. Em outras palavras, se você adicionar uma nova propriedade com metadados de mapeamento na classe `Product``e executar esse comando novamente, ele irá criar a instrução ''alter table'' para adicionar as novas colunas na tabela ``product` existente.

Uma maneira ainda melhor de se aproveitar dessa funcionalidade é por meio das `migrations`, que lhe permitem criar essas instruções SQL e guardá-las em classes migration que podem ser rodadas de forma sistemática no seu servidor de produção para que se possa acompanhar e migrar o schema do seu banco de dados de uma forma mais segura e confiável.

Seu banco de dados agora tem uma tabela `product` totalmente funcional com as colunas correspondendo com os metadados que foram especificados.

Persistindo Objetos no Banco de Dados

Agora que você tem uma entidade `Product` mapeada e a tabela correspondente `product`, já está pronto para persistir os dados no banco. De dentro de um controller, isso é bem simples. Inclua o seguinte método no `DefaultController` do bundle:

```php
// src/Acme/StoreBundle/Controller/DefaultController.php
use Acme\StoreBundle\Entity\Product;
use Symfony\Component\HttpFoundation\Response;
// ...

public function createAction()
{
    $product = new Product();
    $product->setName('A Foo Bar');
    $product->setPrice('19.99');
    $product->setDescription('Lorem ipsum dolor');

    $em = $this->getDoctrine()->getManager();
    $em->persist($product);
    $em->flush();

    return new Response('Created product id '.$product->getId());
}
```

Nota: Se você estiver seguindo o exemplo na prática, precisará criar a rota que aponta para essa action se quiser vê-la funcionando.

Vamos caminhar pelo exemplo:

- **linhas 8-11** Nessa parte você instancia o objeto `$product` como qualquer outro objeto PHP normal;
- **linha 13** Essa linha recuperar o objeto *entity manager* do Doctrine, que é o responsável por lidar com o processo de persistir e retornar objetos do e para o banco de dados;
- **linha 14** O método `persist()` diz ao Doctrine para ''gerenciar" o objeto `$product`. Isso não gera (ainda) um comando real no banco de dados.
- **linha 15** Quando o método `flush()` é chamado, o Doctrine verifica em todos os objetos que ele gerencia para ver se eles necessitam ser persistidos no banco. Nesse exemplo, o objeto `$product` ainda não foi persistido, por isso o entity manager executa um comando `INSERT` e um registro é criado na tabela `product`.

Nota: Na verdade, como o Doctrine conhece todas as entidades gerenciadas, quando você chama o método `flush()`, ele calcula um changeset geral e executa o comando ou os comandos mais eficientes possíveis. Por exemplo, se você vai persistir um total de 100 objetos `Product` e em seguida chamar o método `flush()`, o Doctrine irá criar um *único* prepared statment e reutilizá-lo para cada uma das inserções. Esse padrão é chamado de *Unit of Work*, e é utilizado porque é rápido e eficiente.

Na hora de criar ou atualizar objetos, o fluxo de trabalho é quase o mesmo. Na próxima seção, você verá como o Doctrine é inteligente o suficiente para rodar uma instrução `UPDATE` de forma automática se o registro já existir no banco.

Dica: O Doctrine fornece uma biblioteca que permite a você carregar programaticamente dados de teste no seu projeto (i.e. "fixture data"). Para mais informações, veja `/bundles/DoctrineFixturesBundle/index`.

Trazendo Objetos do Banco de Dados

Trazer um objeto a partir do banco é ainda mais fácil. Por exemplo, suponha que você tenha configurado uma rota para mostrar um `Product` específico baseado no seu valor `id`:

```
public function showAction($id)
{
    $product = $this->getDoctrine()
        ->getRepository('AcmeStoreBundle:Product')
        ->find($id);

    if (!$product) {
        throw $this->createNotFoundException('No product found for id '.$id);
    }

    // faz algo, como passar o objeto $product para um template
}
```

Quando você busca um tipo de objeto em particular, você sempre usa o que chamamos de "repositório". Você pode pensar num repositório como uma classe PHP cuja única função é auxiliar a trazer entidades de uma determinada classe. Você pode acessar o objeto repositório por uma classe entidade dessa forma:

```
$repository = $this->getDoctrine()
    ->getRepository('AcmeStoreBundle:Product');
```

Nota: A string `AcmeStoreBundle:Product` é um atalho que você pode usar em qualquer lugar no Doctrine em vez do nome completo da classe entidade (i.e `Acme\StoreBundle\Entity\Product`). Desde que sua entidade esteja sob o namespace `Entity` do seu bundle, isso vai funcionar.

Uma vez que você tiver seu repositório, terá acesso a todos os tipos de métodos úteis:

```
// Busca pela chave primária (geralmente "id")
$product = $repository->find($id);

// nomes de métodos dinâmicos para busca baseados no valor de uma coluna
$product = $repository->findOneById($id);
$product = $repository->findOneByName('foo');

// busca *todos* os produtos
$products = $repository->findAll();

// busca um grupo de produtos baseada numa valor arbitrário de coluna
$products = $repository->findByPrice(19.99);
```

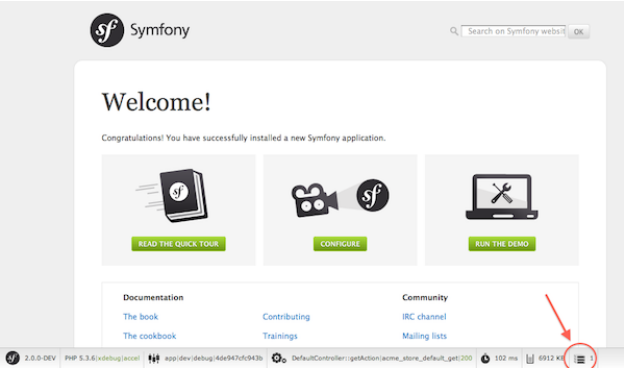
Nota: Naturalmente, você pode também pode rodar consultas complexas, vamos aprender mais sobre isso na seção *Consultando Objetos*.

Você também pode se aproveitar dos métodos bem úteis `findBy` e `findOneBy` para retornar facilmente objetos baseando-se em múltiplas condições:

```
// busca por um produto que corresponda a um nome e um preço
$product = $repository->findOneBy(array('name' => 'foo', 'price' => 19.99));

// busca por todos os produtos correspondentes a um nome, ordenados por
// preço
```

```
$product = $repository->findBy(
    array('name' => 'foo'),
    array('price' => 'ASC')
);
```

Dica: Quando você renderiza uma página, você pode ver quantas buscas foram feitas no canto inferior direito da web debug toolbar.



Se você clicar no ícone, irá abrir o profiler, mostrando a você as consultas exatas que foram feitas.

Atualizando um Objeto

Depois que você trouxe um objeto do Doctrine, a atualização é fácil. Suponha que você tenha uma rota que mapeia o id de um produto para uma action de atualização em um controller:

```
public function updateAction($id)
{
    $em = $this->getDoctrine()->getManager();
    $product = $em->getRepository('AcmeStoreBundle:Product')->find($id);

    if (!$product) {
        throw $this->createNotFoundException('No product found for id '.$id);
    }

    $product->setName('New product name!');
    $em->flush();

    return $this->redirect($this->generateUrl('homepage'));
}
```

Atualizar um objeto envolve apenas três passos:

1. retornar um objeto do Doctrine;
2. modificar o objeto;
3. chamar `flush()` no entity manager

Observe que não é necessário chamar `$em->persist($product)`. Chamar novamente esse método apenas diz ao Doctrine para gerenciar ou "ficar de olho" no objeto `$product`. Nesse caso, como o objeto `$product` foi trazido do Doctrine, ele já está sendo gerenciado.

Excluindo um Objeto

Apagar um objeto é muito semelhante, mas requer um chamada ao método `remove()` do entity manager:

```
$em->remove($product);
$em->flush();
```

Como você podia esperar, o método `remove()` notifica o Doctrine que você quer remover uma determinada entidade do banco. A consulta real `DELETE`, no entanto, não é executada de verdade até que o método `flush()` seja chamado.

Consultando Objetos

Você já viu como o repositório objeto permite que você execute consultas básicas sem nenhum esforço:

```
$repository->find($id);

$repository->findOneByName('Foo');
```

É claro, o Doctrine também permite que se escreva consulta mais complexas usando o Doctrine Query Language (DQL). O DQL é similar ao SQL exceto que você deve imaginar que você está consultando um ou mais objetos de uma classe entidade (i.e. `Product`) em vez de consultar linhas em uma tabela (i.e. `product`).

Quando estiver consultando no Doctrine, você tem duas opções: escrever consultas Doctrine puras ou usar o Doctrine's Query Builder.

Consultando Objetos com DQL

Imagine que você queira buscar por produtos, mas retornar apenas produtos que custem menos que `19,99`, ordenados do mais barato para o mais caro. De um controller, faça o seguinte:

```
$em = $this->getDoctrine()->getManager();
$query = $em->createQuery(
    'SELECT p FROM AcmeStoreBundle:Product p WHERE p.price > :price ORDER BY p.price ASC'
)->setParameter('price', '19.99');

$products = $query->getResult();
```

Se você se sentir confortável com o SQL, então o DQL deve ser bem natural. A grande diferença é que você precisa pensar em termos de "objetos" em vez de linhas no banco de dados. Por esse motivo, você faz um "select" *from* `AcmeStoreBundle:Product` e dá para ele o alias `p`.

O método `getResult()` retorna um array de resultados. Se você estiver buscando por apenas um objeto, você pode usar em vez disso o método `getSingleResult()`:

```
$product = $query->getSingleResult();
```

Cuidado: O método `getSingleResult()` gera uma exceção `Doctrine\ORM\NoResultException` se nenhum resultado for retornado e uma `Doctrine\ORM\NonUniqueResultException` se *mais* de um resultado for retornado. Se você usar esse método, você vai precisar envolvê-lo em um bloco try-catch e garantir que apenas um resultado é retornado (se estiver buscando algo que possa de alguma forma retornar mais de um resultado):

```
$query = $em->createQuery('SELECT ....')
    ->setMaxResults(1);

try {
    $product = $query->getSingleResult();
} catch (\Doctrine\Orm\NoResultException $e) {
    $product = null;
}
// ...
```

A sintaxe DQL é incrivelmente poderosa, permitindo que você faça junções entre entidades facilmente (o tópico de *relacionamentos* será coberto posteriormente), grupos etc. Para mais informações, veja a documentação oficial do Doctrine Query Language.

Configurando parâmetros

Tome nota do método `setParameter()`. Quando trabalhar com o Doctrine, é sempre uma boa ideia configurar os valores externos como `placeholders`, o que foi feito na consulta acima:

```
... WHERE p.price > :price ...
```

Você pode definir o valor do placeholder `price``chamando o método ``setParameter()`:

```
->setParameter('price', '19.99')
```

Usar parâmetros em vez de colocar os valores diretamente no texto da consulta é feito para prevenir ataques de SQL injection e deve ser feito *sempre*. Se você estiver usando múltiplos parâmetros, você pode definir seus valores de uma vez só usando o método `setParameters()`:

```
->setParameters(array(
    'price' => '19.99',
    'name'  => 'Foo',
))
```

Usando o Doctrine's Query Builder

Em vez de escrever diretamente suas consultas, você pode alternativamente usar o `QueryBuilder` do Doctrine para fazer o mesmo serviço usando uma bela interface orientada a objetos. Se você utilizar uma IDE, pode também se beneficiar do auto-complete à medida que você digita o nome dos métodos. A partir de um controller:

```
$repository = $this->getDoctrine()
    ->getRepository('AcmeStoreBundle:Product');

$query = $repository->createQueryBuilder('p')
    ->where('p.price > :price')
    ->setParameter('price', '19.99')
    ->orderBy('p.price', 'ASC')
    ->getQuery();
```

```
$products = $query->getResult();
```

O objeto `QueryBuilder` contém todos os métodos necessários para criar sua consulta. Ao chamar o método `getQuery()`, o query builder retorna um objeto ``Query` normal, que é o mesmo objeto que você criou diretamente na seção anterior.

Para mais informações, consulte a documentação do Query Builder do Doctrine.

Classes Repositório Personalizadas

Nas seções anteriores, você começou a construir e usar consultas mais complexas de dentro de um controller. De modo a isolar, testar e reutilizar essas consultas, é uma boa ideia criar uma classe repositório personalizada para sua entidade e adicionar métodos com sua lógica de consultas lá dentro.

Para fazer isso, adicione o nome da classe repositório na sua definição de mapeamento.

- *Annotations*

```
// src/Acme/StoreBundle/Entity/Product.php
namespace Acme\StoreBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity(repositoryClass="Acme\StoreBundle\Repository\ProductRepository")
 */
class Product
{
    //...
}
```

- *YAML*

```
# src/Acme/StoreBundle/Resources/config/doctrine/Product.orm.yml
Acme\StoreBundle\Entity\Product:
    type: entity
    repositoryClass: Acme\StoreBundle\Repository\ProductRepository
    # ...
```

- *XML*

```
<!-- src/Acme/StoreBundle/Resources/config/doctrine/Product.orm.xml -->
<!-- ... -->
<doctrine-mapping>

    <entity name="Acme\StoreBundle\Entity\Product"
            repository-class="Acme\StoreBundle\Repository\ProductRepository">
            <!-- ... -->
    </entity>
</doctrine-mapping>
```

O Doctrine pode gerar para você a classe repositório usando o mesmo comando utilizado anteriormente para criar os métodos getters e setters que estavam faltando:

```
php app/console doctrine:generate:entities Acme
```

Em seguida, adicione um novo método - `findAllOrderedByName()` - para sua recém-gerada classe repositório. Esse método irá buscar por todas as entidades `Product`, ordenadas alfabeticamente.

```
// src/Acme/StoreBundle/Repository/ProductRepository.php
namespace Acme\StoreBundle\Repository;

use Doctrine\ORM\EntityRepository;

class ProductRepository extends EntityRepository
{
    public function findAllOrderedByName()
    {
        return $this->getEntityManager()
            ->createQuery('SELECT p FROM AcmeStoreBundle:Product p ORDER BY p.name ASC')
            ->getResult();
    }
}
```

Dica: O entity manager pode ser acessado via `$this->getEntityManager()` de dentro do repositório.

Você pode usar esse novo método da mesma forma que os métodos padrões "find" do repositório:

```
$em = $this->getDoctrine()->getManager();
$products = $em->getRepository('AcmeStoreBundle:Product')
            ->findAllOrderedByName();
```

Nota: Quando estiver usando uma classe repositório personalizada, você continua tendo acesso aos métodos padrões finder com `find()` e `findAll()`.

Relacionamentos/Associações de Entidades

Suponha que todos os produtos na sua aplicação pertençam exatamente a uma "categoria". Nesse caso, você precisa de um objeto `Category` e de uma forma de relacionar um objeto `Produto` com um objeto `Category`. Comece criando uma entidade `Category`. Como você sabe que irá eventualmente precisar de fazer a persistência da classe através do Doctrine, você pode deixá-lo criar a classe por você.

```
php app/console doctrine:generate:entity --entity="AcmeStoreBundle:Category" --fields="name:string(25
```

Esse comando gera a entidade `Category` para você, com um campo `id`, um campo `name` e as funções getters e setters relacionadas.

Metadado para Mapeamento de Relacionamentos

Para relacionar as entidades `Category` e `Product`, comece criando a propriedade `products` na classe `Category`:

```
// src/Acme/StoreBundle/Entity/Category.php
// ...
use Doctrine\Common\Collections\ArrayCollection;

class Category
{
    // ...

    /**
```

```
     * @ORM\OneToMany(targetEntity="Product", mappedBy="category")
     */
    protected $products;

    public function __construct()
    {
        $this->products = new ArrayCollection();
    }
}
```

Primeiro, como o objeto `Category` irá se relacionar a vários objetos `Product`, uma propriedade array `products é adicionada para guardar esses objetos `Product`. Novamente, isso não é feito porque o Doctrine precisa dele, mas na verdade porque faz sentido dentro da aplicação guardar um array de objetos `Product`.

Nota: O código no método `__construct()` é importante porque o Doctrine requer que a propriedade `$products` seja um objeto `ArrayCollection`. Esse objeto se parece e age quase *exatamente* como um array, mas tem mais um pouco de flexibilidade embutida. Se isso te deixa desconfortável, não se preocupe. Apenas imagine que ele é um `array` e você estará em boas mãos.

Em seguida, como cada classe `Product` pode se relacionar exatamente com um objeto `Category`, você irá querer adicionar uma propriedade `$category` na classe `Product`:

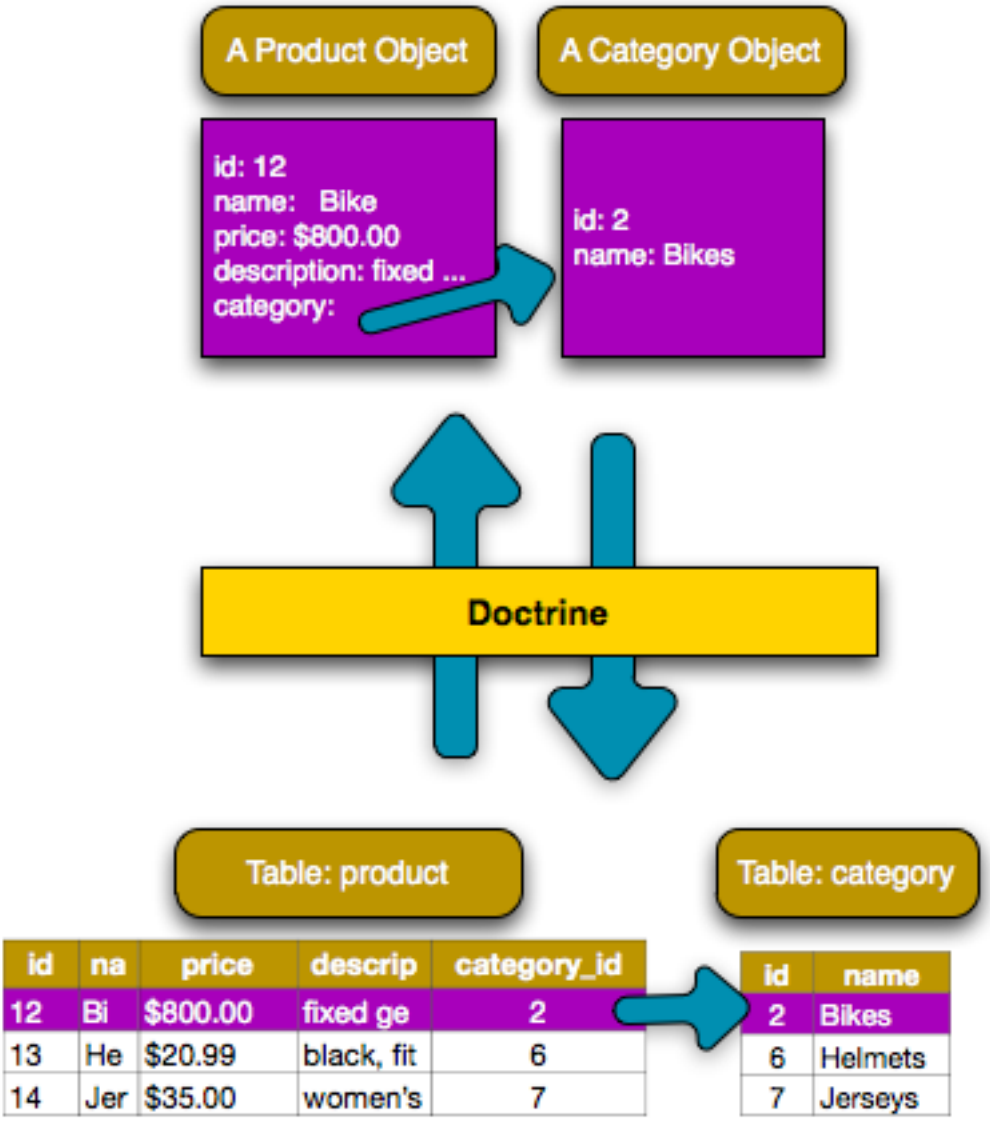
```
// src/Acme/StoreBundle/Entity/Product.php
// ...

class Product
{
    // ...

    /**
     * @ORM\ManyToOne(targetEntity="Category", inversedBy="products")
     * @ORM\JoinColumn(name="category_id", referencedColumnName="id")
     */
    protected $category;
}
```

Finalmente, agora que você adicionou um nova propriedade tanto na classe `Category` quanto na `Product`, diga ao Doctrine para gerar os métodos getters e setters que estão faltando para você:

```
php app/console doctrine:generate:entities Acme
```

Ignore o metadado do Doctrine por um instante. Agora você tem duas classes - `Category` e `Product` com um relacionamento natural um-para-muitos. A classe categoria contém um array de objetos `Product` e o objeto `Product` pode conter um objeto `Category`. Em outras palavras - você construiu suas classes de um jeito que faz sentido para as suas necessidades. O fato de que os dados precisam ser persistidos no banco é sempre secundário.

Agora, olhe o metadado acima da propriedade `$category` na classe `Product`. A informação aqui diz para o Doctrine que a classe relacionada é a `Category` e que ela deve guardar o `id` do registro categoria em um campo `category_id` que fica na tabela `product`. Em outras palavras, o objeto `Category` será guardado na propriedade `$category`, mas nos bastidores, o Doctrine irá persistir esse relacionamento guardando o valor do id da categoria na coluna `category_id` da tabela `product`.



O metadado acima da propriedade `$products` do objeto `Category` é menos importante, e simplesmente diz ao Doctrine para olhar a propriedade `Product.category` para descobrir como o relacionamento é mapeado.

Antes de continuar, tenha certeza de dizer ao Doctrine para adicionar uma nova tabela `category`, além de uma coluna `product.category_id` e uma nova chave estrangeira:

```
php app/console doctrine:schema:update --force
```

Nota: Esse comando deve ser usado apenas durante o desenvolvimento. Para um método mais robusto de atualização sistemática em um banco de dados de produção, leia sobre as `Doctrine migrations`.

Salvando as Entidades Relacionadas

Agora é o momento de ver o código em ação. Imagine que você está dentro de um controller:

```
// ...
use Acme\StoreBundle\Entity\Category;
use Acme\StoreBundle\Entity\Product;
use Symfony\Component\HttpFoundation\Response;
// ...

class DefaultController extends Controller
{
    public function createProductAction()
    {
        $category = new Category();
        $category->setName('Main Products');

        $product = new Product();
        $product->setName('Foo');
        $product->setPrice(19.99);
        // relaciona a categoria com esse produto
        $product->setCategory($category);

        $em = $this->getDoctrine()->getManager();
        $em->persist($category);
        $em->persist($product);
        $em->flush();

        return new Response(
            'Created product id: '.$product->getId().' and category id: '.$category->getId()
        );
    }
}
```

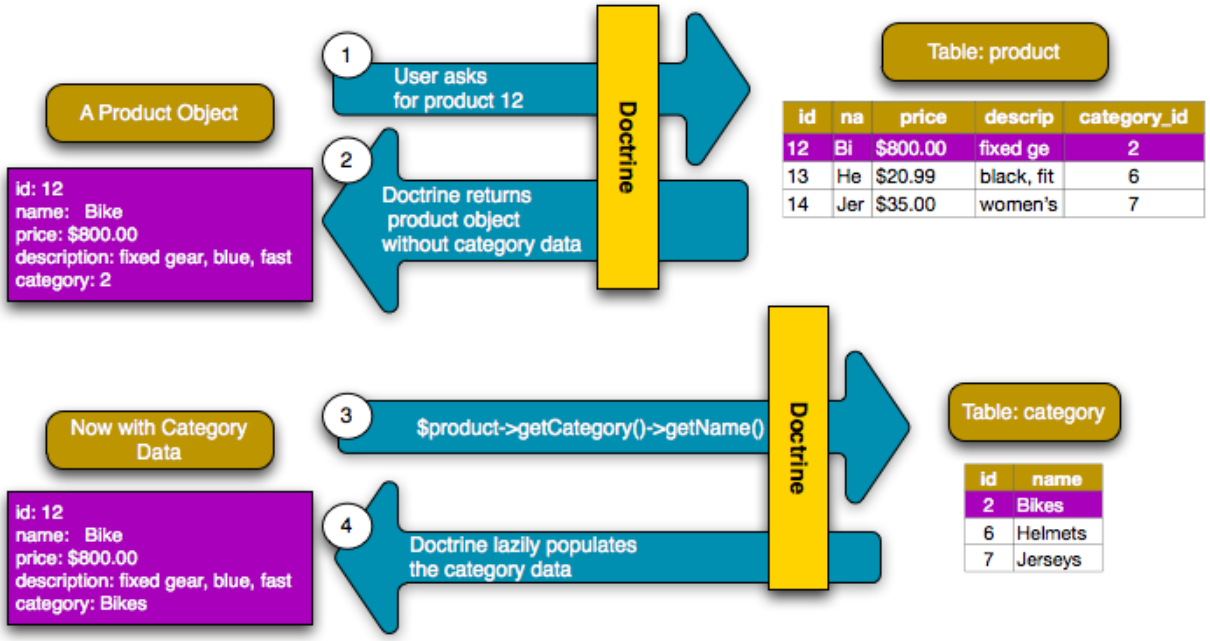
Agora, um registro único é adicionado para ambas tabelas `category` e `product`. A coluna `product.category_id` para o novo produto é definida como o que for definido como `id` na nova categoria. O Doctrine gerencia a persistência desse relacionamento para você.

Retornando Objetos Relacionados

Quando você precisa pegar objetos associados, seu fluxo de trabalho é parecido com o que foi feito anteriormente. Primeiro, consulte um objeto `$product` e então acesse seu o objeto `Category` relacionado:

```
public function showAction($id)
{
    $product = $this->getDoctrine()
        ->getRepository('AcmeStoreBundle:Product')
        ->find($id);

    $categoryName = $product->getCategory()->getName();

    // ...
}
```

Nesse exemplo, você primeiro busca por um objeto `Product` baseado no `id` do produto. Isso gera uma consulta *apenas* para os dados do produto e faz um hydrate do objeto `$product` com esses dados. Em seguida, quando você chamar `$product->getCategory()->getName()`, o Doctrine silenciosamente faz uma segunda consulta para buscar a `Category` que está relacionada com esse `Product`. Ele prepara o objeto `$category` e o retorna para você.



O que é importante é o fato de que você tem acesso fácil as categorias relacionadas com os produtos, mas os dados da categoria não são realmente retornados até que você peça pela categoria (i.e. sofre "lazy load").

Você também pode buscar na outra direção:

```
public function showProductAction($id)
{
    $category = $this->getDoctrine()
        ->getRepository('AcmeStoreBundle:Category')
        ->find($id);

    $products = $category->getProducts();

    // ...
}
```

Nesse caso, ocorre a mesma coisa: primeiro você busca por um único objeto `Category`, e então o Doctrine faz uma segunda busca para retornar os objetos `Product` relacionados, mas apenas se você pedir por eles (i.e. quando você chama `->getProducts()`). A variável `$products` é uma array de todos os objetos `Product` que estão relacionados com um dado objeto `Category` por meio do valor de seu campo `category_id`.

Relacionamentos e Classes Proxy

O "lazy loading" é possível porque, quando necessário, o Doctrine retorna um objeto "proxy" no lugar do objeto real. Olhe novamente o exemplo acima:

```
$product = $this->getDoctrine()
    ->getRepository('AcmeStoreBundle:Product')
    ->find($id);

$category = $product->getCategory();

// prints "Proxies\AcmeStoreBundleEntityCategoryProxy"
echo get_class($category);
```

Esse objeto proxy estende o verdadeiro objeto `Category`, e se parece e age exatamente como ele. A diferença é que, por usar um objeto proxy, o Doctrine pode retardar a busca pelos dados reais da `Category``até que você realmente precise daqueles dados (e.g. até que você chame ``$category->getName()`).
As classes proxy são criadas pelo Doctrine e armazenadas no diretório cache. E apesar de que você provavelmente nunca irá notar que o seu objeto `$category` é na verdade um objeto proxy, é importante manter isso em mente.
Na próxima seção, quando você retorna os dados do produto e categoria todos de uma vez (via um *join*), o Doctrine irá retornar o *verdadeiro* objeto "Category", uma vez que nada precisa ser carregado de modo "lazy load".

Juntando Registros Relacionados

Nos exemplos acima, duas consultas foram feitas - uma para o objeto original (e.g uma `Category`) e uma para os objetos relacionados (e.g. os objetos `Product`).

Dica: Lembre que você pode visualizar todas as consultas feitas durante uma requisição pela web debug toolbar.

É claro, se você souber antecipadamente que vai precisar acessar ambos os objetos, você pode evitar a segunda consulta através da emissão de um "join" na consulta original. Inclua o método seguinte na classe `ProductRepository`:

```
// src/Acme/StoreBundle/Repository/ProductRepository.php

public function findOneByIdJoinedToCategory($id)
{
    $query = $this->getEntityManager()
        ->createQuery('
            SELECT p, c FROM AcmeStoreBundle:Product p
            JOIN p.category c
            WHERE p.id = :id'
        )->setParameter('id', $id);

    try {
        return $query->getSingleResult();
    } catch (\Doctrine\ORM\NoResultException $e) {
        return null;
    }
}
```

Agora, você pode usar esse método no seu controller para buscar um objeto `Product` e sua `Category` relacionada com apenas um consulta:

```
public function showAction($id)
{
    $product = $this->getDoctrine()
        ->getRepository('AcmeStoreBundle:Product')
        ->findOneByIdJoinedToCategory($id);

    $category = $product->getCategory();

    // ...
}
```

Mais Informações sobre Associações

Essa seção foi uma introdução para um tipo comum de relacionamento de entidades, o um-para-muitos. Para detalhes mais avançados e exemplos de como usar outros tipos de relacionamentos (i.e. `um-para-um`, ``muitos-para-muitos`), verifique a Documentação sobre Mapeamento e Associações do Doctrine.

Nota: Se você estiver usando annotations, irá precisar prefixar todas elas com `ORM\` (e.g `ORM\OneToMany`), o que não está descrito na documentação do Doctrine. Você também precisará incluir a instrução `use Doctrine\ORM\Mapping as ORM;`, que faz a *importação* do prefixo `ORM` das annotations.

Configuração

O Doctrine é altamente configurável, embora você provavelmente não vai precisar se preocupar com a maioria de suas opções. Para saber mais sobre a configuração do Doctrine, veja a seção Doctrine do `reference manual`.

Lifecycle Callbacks

Às vezes, você precisa executar uma ação justamente antes ou depois de uma entidade ser inserida, atualizada ou apagada. Esses tipos de ações são conhecidas como "lifecycle" callbacks, pois elas são métodos callbacks que você precisa executar durante diferentes estágios do ciclo de vida de uma entidade (i.e. a entidade foi inserida, atualizada, apagada, etc.).

Se você estiver usando annotations para seus metadados, comece habilitando esses callbacks. Isso não é necessário se estiver utilizando YAML ou XML para seus mapeamentos:

```
/**
 * @ORM\Entity()
 * @ORM\HasLifecycleCallbacks()
 */
class Product
{
    // ...
}
```

Agora, você pode dizer ao Doctrine para executar um método em cada um dos eventos de ciclo de vida disponíveis. Por exemplo, suponha que você queira definir uma coluna `created` do tipo data para a data atual, apenas quando for a primeira persistência da entidade (i.e. inserção):

- *Annotations*

```
/**
 * @ORM\prePersist
 */
public function setCreatedValue()
{
    $this->created = new \DateTime();
}
```

- *YAML*

```
# src/Acme/StoreBundle/Resources/config/doctrine/Product.orm.yml
Acme\StoreBundle\Entity\Product:
    type: entity
    # ...
    lifecycleCallbacks:
        prePersist: [ setCreatedValue ]
```

- *XML*

```
<!-- src/Acme/StoreBundle/Resources/config/doctrine/Product.orm.xml -->
<!-- ... -->
<doctrine-mapping>

    <entity name="Acme\StoreBundle\Entity\Product">
            <!-- ... -->
            <lifecycle-callbacks>
                <lifecycle-callback type="prePersist" method="setCreatedValue" />
            </lifecycle-callbacks>
    </entity>
</doctrine-mapping>
```

Nota: O exemplo acima presume que você tenha criado e mapeado uma propriedade `created` (que não foi mostrada aqui).

Agora, logo no momento anterior a entidade ser persistida pela primeira vez, o Doctrine irá automaticamente chamar esse método e o campo `created` será preenchido com a data atual.

Isso pode ser repetido para qualquer um dos outros eventos de ciclo de vida, que incluem:

- `preRemove`
- `postRemove`
- `prePersist`
- `postPersist`
- `preUpdate`
- `postUpdate`
- `postLoad`
- `loadClassMetadata`

Para mais informações sobre o que esses eventos significam e sobre os lifecycle callbacks em geral, veja a Documentação sobre Lifecycle Events do Doctrine.

Lifecycle Callbacks e Event Listeners

Observe que o método `setCreatedValue()` não recebe nenhum argumento. Esse é o comportamento usual dos lifecycle callbacks e é intencional: eles devem ser métodos simples que estão preocupados com as transformações internas dos dados na entidade (e.g. preencher um campo created/updated ou gerar um valor slug).
Se você precisar fazer algo mais pesado - como rotinas de log ou mandar um e-mail - você deve registrar uma classe externa como um event listener ou subscriber e dar para ele acesso aos recursos que precisar. Para mais informações, veja Como Registrar Ouvintes e Assinantes de Eventos.

Extensões do Doctrine: Timestampable, Sluggable, etc.

O Doctrine é bastante flexível, e um grande número de extensões de terceiros está disponível o que permirte que você execute facilmente tarefas repetitivas e comuns nas suas entidades. Isso inclui coisas como *Sluggable*, *Timestampable*, *Loggable*, *Translatable* e *Tree*.

Para mais informações sobre como encontrar e usar essas extensões, veja o artigo no cookbook sobre using common Doctrine extensions.

Referência dos Tipos de Campos do Doctrine

O Doctrine já vem com um grande número de tipos de campo disponível. Cada um deles mapeia um tipo de dados do PHP para um tipo de coluna específico em qualquer banco de dados que você estiver utilizando. Os seguintes tipos são suportados no Doctrine:

- **Strings**
  - `string` (usado para strings curtas)
  - `text` (usado para strings longas)
- **Números**
  - `integer`
  - `smallint`
  - `bigint`
  - `decimal`
  - `float`
- **Datas e Horários** (usa um objeto DateTime para esses campos no PHP)
  - `date`
  - `time`
  - `datetime`
- **Outros Tipos**
  - `boolean`
  - `object` (serializado e armazenado em um campo `CLOB`)
  - `array` (serializado e guardado em um campo `CLOB`)

Para mais informações, veja a Documentação sobre Tipos de Mapeamento do Doctrine.

Opções de Campo

Cada campo pode ter um conjunto de opções aplicado sobre ele. As opções disponíveis incluem `type` (o padrão é `string`), `name`, `lenght`, `unique` e `nullable`. Olhe alguns exemplos de annotations:

```
/**
 * Um campo string com tamanho 255 que não pode ser nulo
 * (segue os valores padrões para "type", "length" e *nullable* options)
 *
 * @ORM\Column()
 */
protected $name;

/**
 * Um campo string com tamanho 150 persistido na coluna "email_adress"
 * e com um índice único
 *
 * @ORM\Column(name="email_address", unique="true", length="150")
 */
protected $email;
```

Nota: Existem mais algumas opções que não estão listadas aqui. Para mais detalhes, veja a Documentação sobre Mapeamento de Propriedades do Doctrine.

Comandos de Console

A integração com o Doctrine2 ORM fornece vários comandos de console no namespace `doctrine`. Para ver a lista de comandos, você pode executar o console sem nenhum argumento:

```
php app/console
```

A lista dos comandos disponíveis será mostrada, muitos dos quais começam com o prefixo `doctrine`. Você pode encontrar mais informações sobre qualquer um desses comandos (e qualquer comando do Symfony) rodando o comando `help`. Por exemplo, para pegar detalhes sobre o comando `doctrine:database:create`, execute:

```
php app/console help doctrine:database:create
```

Alguns comandos interessantes e notáveis incluem:

- `doctrine:ensure-production-settings` - verifica se o ambiente atual está configurado de forma eficiente para produção. Deve ser sempre executado no ambiente `prod`:

```
    php app/console doctrine:ensure-production-settings --env=prod
```

- `doctrine:mapping:import` - permite ao Doctrine fazer introspecção de um banco de dados existente e criar a informação de mapeamento. Para mais informações veja Como gerar Entidades de uma base de dados existente.
- `doctrine:mapping:info` - diz para você todas as entidades que o Doctrine tem conhecimento e se existe ou não algum erro básico com o mapeamento.
- `doctrine:query:dql` and `doctrine:query:sql` - permite que você execute consultas DQL ou SQL diretamente na linha de comando.

Nota: Para poder carregar data fixtures para seu banco de dados, você precisa ter o bundle `DoctrineFixturesBundle` instalado. Para aprender como fazer isso, leia a entrada

"/bundles/DoctrineFixturesBundle/index" da documentação.

Sumário

Com o Doctrine, você pode se focar nos seus objetos e como eles podem ser úteis na sua aplicação, deixando a preocupação com a persistência de banco de dados em segundo plano. Isso porque o Doctrine permite que você use qualquer objeto PHP para guardar seus dados e se baseia nos metadados de mapeamento para mapear os dados de um objetos para um tabela específica no banco.

E apesar do Doctrine girar em torno de um conceito simples, ele é incrivelmente poderoso, permitindo que você crie consultas complexas e faça subscrição em eventos que permitem a você executar ações diferentes à medida que os objetos vão passando pelo seu ciclo de vida de persistência.

Para mais informações sobre o Doctrine, veja a seção *Doctrine* do cookbook, que inclui os seguintes artigos:

- /bundles/DoctrineFixturesBundle/index
- Como usar as extens?es do Doctrine: Timestampable, Sluggable, Translatable, etc.

2.1.8 Testes

Sempre que você escrever uma nova linha de código, você também adiciona potenciais novos bugs. Para construir aplicações melhores e mais confiáveis, você deve testar seu código usando testes funcionais e unitários.

O Framework de testes PHPUnit

O Symfony2 se integra com uma biblioteca independente - chamada PHPUnit - para dar a você um rico framework de testes. Esse capitulo não vai abranger o PHPUnit propriamente dito, mas ele tem a sua excelente documentação documentation.

Nota: O Symfony2 funciona com o PHPUnit 3.5.11 ou posterior, embora a versão 3.6.4 é necessária para testar o código do núcleo do Symfony.

Cada teste - quer seja teste unitário ou teste funcional - é uma classe PHP que deve residir no sub-diretório *Tests/* de seus bundles. Se você seguir essa regra, você pode executar todos os testes da sua aplicação com o seguinte comando:

```
# espefifique o diretório de configuração na linha de comando
$ phpunit -c app/
```

A opção `-c` diz para o PHPUnit procurar no diretório `app/` por um arquivo de configuração. Se você está curioso sobre as opções do PHPUnit, dê uma olhada no arquivo `app/phpunit.xml.dist`.

Dica: O Code coverage pode ser gerado com a opção `--coverage-html`.

Testes Unitários

Um teste unitário é geralmente um teste de uma classe PHP especifica. Se você quer testar o comportamento global da sua aplicação, veja a seção sobre *Testes Funcionais*.

Escrever testes unitários no Symfony2 não é nada diferente do que escrever um teste unitário padrão do PHPUnit. Vamos supor que, por exemplo, você tem uma classe *incrivelmente* simples chamada `Calculator` no diretório `Utility/` do seu bundle:

```
// src/Acme/DemoBundle/Utility/Calculator.php
namespace Acme\DemoBundle\Utility;

class Calculator
{
    public function add($a, $b)
    {
        return $a + $b;
    }
}
```

Para testar isso, crie um arquivo chamado `CalculatorTest` no diretório `Tests/Utility` do seu bundle:

```
// src/Acme/DemoBundle/Tests/Utility/CalculatorTest.php
namespace Acme\DemoBundle\Tests\Utility;

use Acme\DemoBundle\Utility\Calculator;

class CalculatorTest extends \PHPUnit_Framework_TestCase
{
    public function testAdd()
    {
        $calc = new Calculator();
        $result = $calc->add(30, 12);

        // assert that our calculator added the numbers correctly!
        $this->assertEquals(42, $result);
    }
}
```

Nota: Por convenção, o sub-diretório `Tests/` deve replicar o diretório do seu bundle. Então, se você estiver testando uma classe no diretório `Utility/` do seu bundle, coloque o teste no diretório `Tests/Utility/`.

Assim como na rua aplicação verdadeira - o autoloading é automaticamente habilitado via o arquivo `bootstrap.php.cache` (como configurado por padrão no arquivo `phpunit.xml.dist`).

Executar os testes para um determinado arquivo ou diretório também é muito fácil:

```
# executa todos os testes no diretório Utility
$ phpunit -c app src/Acme/DemoBundle/Tests/Utility/

# executa os testes para a classe Article
$ phpunit -c app src/Acme/DemoBundle/Tests/Utility/CalculatorTest.php

# executa todos os testes para todo o Bundle
$ phpunit -c app src/Acme/DemoBundle/
```

Testes Funcionais

Testes funcionais verificam a integração das diferentes camadas de uma aplicação (do roteamento as views). Eles não são diferentes dos testes unitários levando em consideração o PHPUnit, mas eles tem um fluxo bem especifico:

- Fazer uma requisição;

- Testar a resposta;
- Clicar em um link ou submeter um formulário;
- Testar a resposta;
- Repetir a operação.

Seu Primeiro Teste Funcional

Testes funcionais são arquivos PHP simples que estão tipicamente no diretório `Tests/Controller` do seu bundle. Se você quer testar as páginas controladas pela sua classe `DemoController`, inicie criando um novo arquivo `DemoControllerTest.php` que extende a classe especial `WebTestCase`.

Por exemplo, o Symfony2 Standard Edition fornece um teste funcional simples para o `DemoController` (DemoControllerTest) descrito assim:

```
// src/Acme/DemoBundle/Tests/Controller/DemoControllerTest.php
namespace Acme\DemoBundle\Tests\Controller;

use Symfony\Bundle\FrameworkBundle\Test\WebTestCase;

class DemoControllerTest extends WebTestCase
{
    public function testIndex()
    {
        $client = static::createClient();

        $crawler = $client->request('GET', '/demo/hello/Fabien');

        $this->assertTrue($crawler->filter('html:contains("Hello Fabien")')->count() > 0);
    }
}
```

Dica: Para executar seus testes funcionais, a classe `WebTestCase` class inicializa o kernel da sua aplicação. Na maioria dos casos, isso acontece automaticamente. Entretando, se o seu kernel está em um diretório diferente do padrão, você vai precisar modificar seu arquivo `phpunit.xml.dist` para alterar a variável de ambiente `KERNEL_DIR` para o diretório do seu kernel:

```
<phpunit
    <!-- ... -->
    <php>
        <server name="KERNEL_DIR" value="/path/to/your/app/" />
    </php>
    <!-- ... -->
</phpunit>
```

O método `createClient()` retorna um cliente, que é como um navegador que você vai usar para navegar no seu site:

```
$crawler = $client->request('GET', '/demo/hello/Fabien');
```

O método `request()` (veja *mais sobre o método request*) retorna um objeto `Crawler` que pode ser usado para selecionar um elemento na Response, clicar em links, e submeter formulários.

Dica: O Crawler só funciona se a resposta é um documento XML ou HTML. Para pegar a resposta bruta, use `$client->getResponse()->getContent()`.

Clique em um link primeiramente selecionando-o com o Crawler usando uma expressão XPath ou um seletor CSS, então use o Client para clicar nele. Por exemplo, o segunte código acha todos os links com o texto `Greet`, então seleciona o segundo, e então clica nele:

```
$link = $crawler->filter('a:contains("Greet")')->eq(1)->link();

$crawler = $client->click($link);
```

Submeter um formulário é muito parecido, selecione um botão do formulário, opcionalmente sobrescreva alguns valores do formulário, então submeta-o:

```
$form = $crawler->selectButton('submit')->form();

// pega alguns valores
$form['name'] = 'Lucas';
$form['form_name[subject]'] = 'Hey there!';

// submete o formulário
$crawler = $client->submit($form);
```

Dica: O formulário também pode manipular uploads e tem métodos para preencher diferentes tipos de campos (ex. `select()` e `tick()`). Para mais detalhers, veja a seção **'Forms'_** abaixo.

Agora que você pode facilmente navegar pela sua aplicação, use as afirmações para testar que ela realmente faz o que você espera que ela faça. Use o Crawler para fazer afirmações no DOM:

```
// Afirma que a resposta casa com um seletor informado
$this->assertTrue($crawler->filter('h1')->count() > 0);
```

Ou, teste contra o conteúdo do Response diretamente se você só quer afirmar que o conteudo contém algum texto ou se o Response não é um documento XML/HTML:

```
$this->assertRegExp('/Hello Fabien/', $client->getResponse()->getContent());
```

Mais sobre o método `request()`:

A assinatura completa do método `request()` é:

```
request(
    $method,
    $uri,
    array $parameters = array(),
    array $files = array(),
    array $server = array(),
    $content = null,
    $changeHistory = true
)
```

O array `server` são valores brutos que você espera encontrar normalmente na variável superglobal do PHP $_SERVER. Por exemplo, para setar os cabeçalhos HTTP *Content-Type* e *Referer*, você passará o seguinte:

```
$client->request(
    'GET',
    '/demo/hello/Fabien',
    array(),
    array(),
    array(
        'CONTENT_TYPE' => 'application/json',
        'HTTP_REFERER' => '/foo/bar',
    )
);
```

Trabalhando com o Teste Client

O teste Client simula um cliente HTTP como um navegador e faz requisições na sua aplicação Symfony2:

```
$crawler = $client->request('GET', '/hello/Fabien');
```

O método `request()` pega o método HTTP e a URL como argumentos e retorna uma instancia de `Crawler`.

Utilize o Crawler para encontrar elementos DOM no Response. Esses elementos podem então ser usados para clicar em links e submeter formulários:

```
$link = $crawler->selectLink('Go elsewhere...')->link();
$crawler = $client->click($link);

$form = $crawler->selectButton('validate')->form();
$crawler = $client->submit($form, array('name' => 'Fabien'));
```

Os métodos `click()` e `submit()` retornam um objeto `Crawler`. Esses métodos são a melhor maneira de navegar na sua aplicação por tomarem conta de várias coisas para você, como detectar o método HTTP de um formulário e dar para você uma ótima API para upload de arquivos.

Dica: Você vai aprende rmais sobre os objetos `Link` e `Form` na seção *Crawler* abaixo.

O método `request` pode também ser usado para simular submissões de formulários diretamente ou fazer requisições mais complexas:

```
// Submeter diretamente um formuário (mas utilizando o Crawler é mais fácil!)
$client->request('POST', '/submit', array('name' => 'Fabien'));
```

```
// Submissão de formulário com um upload de arquivo
use Symfony\Component\HttpFoundation\File\UploadedFile;

$photo = new UploadedFile(
    '/path/to/photo.jpg',
    'photo.jpg',
    'image/jpeg',
    123
);
// ou
$photo = array(
    'tmp_name' => '/path/to/photo.jpg',
    'name' => 'photo.jpg',
    'type' => 'image/jpeg',
    'size' => 123,
    'error' => UPLOAD_ERR_OK
);
$client->request(
    'POST',
    '/submit',
    array('name' => 'Fabien'),
    array('photo' => $photo)
);

// Executa uma requisição de DELETE e passa os cabeçalhos HTTP
$client->request(
    'DELETE',
    '/post/12',
    array(),
    array(),
    array('PHP_AUTH_USER' => 'username', 'PHP_AUTH_PW' => 'pa$$word')
);
```

Por último mas não menos importante, você pode forçar cara requisição para ser executada em seu próprio processo PHP para evitar qualquer efeito colateral quando estiver trabalhando com vários clientes no mesmo script:

```
$client->insulate();
```

Navegando

O Cliente suporta muitas operação que podem ser realizadas em um navegador real:

```
$client->back();
$client->forward();
$client->reload();

// Limpa todos os cookies e histórico
$client->restart();
```

Acessando Objetos Internos

Se você usa o cliente para testar sua aplicação, você pode querer acessar os objetos internos do cliente:

```
$history   = $client->getHistory();
$cookieJar = $client->getCookieJar();
```

Você também pode pegar os objetos relacionados a requisição mais recente:

```
$request  = $client->getRequest();
$response = $client->getResponse();
$crawler  = $client->getCrawler();
```

Se as suas requisição não são isoladas, você pode também acessar o `Container` e o `Kernel`:

```
$container = $client->getContainer();
$kernel    = $client->getKernel();
```

Acessando o Container

É altamente recomendado que um teste funcional teste somente o Response. Mas em circunstancias extremamente raras, você pode querer acessar algum objeto interno para escrever afirmações. Nestes casos, você pode acessar o dependency injection container:

```
$container = $client->getContainer();
```

Esteja ciente que isso não funciona se você isolar o cliente ou se você usar uma camada HTTP. Para ver a lista de serviços disponíves na sua aplicação, utilize a task `container:debug`.

Dica: Se a informação que você precisa verificar está disponível no profiler, uso-o então

Acessando dados do Profiler

En cada requisição, o profiler do Symfony coleta e guarda uma grande quantidade de dados sobre a manipulação interna de cada request. Por exemplo, o profiler pode ser usado para verificar se uma determinada página executa menos consultas no banco quando estiver carregando.

Para acessar o Profiler da última requisição, faço o seguinte:

```
$profile = $client->getProfile();
```

Para detalhes especificos de como usar o profiler dentro de um teste, seja o artigo `/cookbook/testing/profiling` do cookbook.

Redirecionamento

Quando uma requisição retornar uma redirecionamento como resposta, o cliente automaticamente segue o redirecionamento. Se você quer examinar o Response antes do redirecionamento use o método `followRedirects()`:

```
$client->followRedirects(false);
```

Quando o cliente não segue os redirecionamentos, você pode forçar o redirecionamento com o método `followRedirect()`:

```
$crawler = $client->followRedirect();
```

O Crawler

Uma instancia do Crawler é retornada cada vez que você faz uma requisição com o Client. Ele permite que você examinar documentos HTML, selecionar nós, encontrar links e formulários.

Examinando

Como o jQuery, o Crawler tem metodos para examinar o DOM de um documento HTML/XML. Por exemplo, isso encontra todos os elementos `input[type=submit]`, seleciona o último da página, e então seleciona o elemento imediatamente acima dele:

```
$newCrawler = $crawler->filter('input[type=submit]')
    ->last()
    ->parents()
    ->first()
;
```

Muitos outros métodos também estão disponíveis:

Metodos	Descrição
`filter('h1.title')`	Nós que casam com o seletor CSS
`filterXpath('h1')`	Nós que casam com a expressão XPath
`eq(1)`	Nó para a posição especifica
`first()`	Primeiro nó
`last()`	Último nó
`siblings()`	Irmãos
`nextAll()`	Todos os irmãos posteriores
`previousAll()`	Todos os irmãos anteriores
`parents()`	Nós de um nivel superior
`children()`	Filhos
`reduce($lambda)`	Nós que a função não retorne false

Como cada um desses métodos retorna uma nova instância de `Crawler`, você pode restringir os nós selecionados encadeando a chamada de métodos:

```
$crawler
    ->filter('h1')
    ->reduce(function ($node, $i)
    {
        if (!$node->getAttribute('class')) {
            return false;
        }
    })
    ->first();
```

Dica: Utilize a função `count()` para pegar o número de nós armazenados no Crawler: `count($crawler)`

Extraindo Informações

O Crawler pode extrair informações dos nós:

```
// Retornar o valor do atributo para o primeiro nó
$crawler->attr('class');

// Retorna o valor do nó para o primeiro nó
$crawler->text();

// Extrai um array de atributos para todos os nós (_text retorna o valor do nó)
// retorna um array para cara elemento no crawler, cara um com o valor e href
```

```
$info = $crawler->extract(array('_text', 'href'));

// Executa a lambda para cada nó e retorna um array de resultados
$data = $crawler->each(function ($node, $i)
{
    return $node->attr('href');
});
```

Links

Para selecionar links, você pode usar os métodos acima ou o conveniente atalho `selectLink()`:

```
$crawler->selectLink('Click here');
```

Isso seleciona todos os links que contém o texto, ou imagens que o atributo `alt` contém o determinado texto. Como outros métodos de filtragem, esse retorna outro objeto `Crawler`.

Uma vez selecionado um link, você pode ter acesso a um objeto especial `Link`, que tem métodos especificos muito úties para links (como `getMethod()` e `getUri()`). Para clicar no link, use o método do Client `click()` e passe um objeto do tipo `Link`:

```
$link = $crawler->selectLink('Click here')->link();

$client->click($link);
```

Formulários

Assim como nos links, você seleciona o form com o método `selectButton()`:

```
$buttonCrawlerNode = $crawler->selectButton('submit');
```

Nota: Note que selecionamos os botões do formulário e não os forms, pois o form pode ter vários botões; se você usar a API para examinar, tenha em mente que você deve procurar por um botão.

O método `selectButton()` pode selecionar tags `button` e submit tags `input`. Ele usa diversas partes diferentes do botão para encontrá-los:

- O atributo `value`;
- O atributo `id` ou `alt` para imagens;
- O valor do atributo `id` ou `name` para tags `button`.

Uma vez que você tenha o Crawler representanto um botão, chame o método `form()` para pegar a instancia de `Form` do form que envolve o nó do botão:

```
$form = $buttonCrawlerNode->form();
```

Quando chamar o método `form()`, você pode também passar uma array com valores dos campos para sobreescrever os valores padrões:

```
$form = $buttonCrawlerNode->form(array(
    'name'              => 'Fabien',
    'my_form[subject]'  => 'Symfony rocks!',
));
```

E se você quiser simular algum método HTTP especifico para o form, passe-o como um segundo argumento:

```
$form = $crawler->form(array(), 'DELETE');
```

O Client pode submeter instancias de `Form`:

```
$client->submit($form);
```

Os valores dos campos também posem ser passsados como um segundo argumento do método `submit()`:

```
$client->submit($form, array(
    'name'              => 'Fabien',
    'my_form[subject]'  => 'Symfony rocks!',
));
```

Para situações mais complexas, use a instancia de `Form` como um array para setar o valor de cada campo individualmente:

```
// Muda o valor do campo
$form['name'] = 'Fabien';
$form['my_form[subject]'] = 'Symfony rocks!';
```

Também existe uma API para manipular os valores do campo de acordo com o seu tipo:

```
// Seleciona um option ou um radio
$form['country']->select('France');

// Marca um checkbox
$form['like_symfony']->tick();

// Faz o upload de um arquivo
$form['photo']->upload('/path/to/lucas.jpg');
```

Dica: Você pode pegar os valores que serão submetidos chamando o método `getValues()` no objeto `Form`. Os arquivos do upload estão disponiveis em um array separado retornado por `getFiles()`. Os métodos `getPhpValues()` e `getPhpFiles()` também retorna valores submetidos, mas no formato PHP (ele converte as chaves para a notação de colchetes - ex. `my_form[subject]` - para PHP arrays).

Configuração de Testes

O Client usado pelos testes funcionais cria um Kernel que roda em um ambiente especial chamado `test`. Uma vez que o Symfony carrega o `app/config/config_test.yml` no ambiente `test`, você pode ajustar qualquer configuração de sua aplicação especificamente para testes.

Por exemplo, por padrão, o swiftmailer é configurado para *não* enviar realmente os e-mails no ambiente `test`. Você pode ver isso na opção de configuração `swiftmailer`:

- *YAML*

```
# app/config/config_test.yml
# ...

swiftmailer:
    disable_delivery: true
```

- *XML*

```
    <!-- app/config/config_test.xml -->
    <container>
        <!-- ... -->

        <swiftmailer:config disable-delivery="true" />
    </container>
```

- *PHP*

```
    // app/config/config_test.php
    // ...

    $container->loadFromExtension('swiftmailer', array(
        'disable_delivery' => true
    ));
```

Você também pode usar um ambiente completamente diferente, ou sobrescrever o modo de debug (`true`) passando cada um como uma opção para o método `createClient()`:

```
$client = static::createClient(array(
    'environment' => 'my_test_env',
    'debug'       => false,
));
```

Se a sua aplicação se comporta de acordo com alguns cabeçalhos HTTP, passe eles como o segundo argumento de `createClient()`:

```
$client = static::createClient(array(), array(
    'HTTP_HOST'       => 'en.example.com',
    'HTTP_USER_AGENT' => 'MySuperBrowser/1.0',
));
```

Você também pode sobrescrever cabeçalhos HTTP numa base por requisições:

```
$client->request('GET', '/', array(), array(), array(
    'HTTP_HOST'       => 'en.example.com',
    'HTTP_USER_AGENT' => 'MySuperBrowser/1.0',
));
```

Dica: O cliente de testes está disponível como um serviço no container no ambiente `teste` (ou em qualquer lugar que a opção framework.test esteja habilitada). Isso significa que você pode sobrescrever o serviço inteiramente se você precisar.

Configuração do PHPUnit

Cada aplicação tem a sua própria configuração do PHPUnit, armazenada no arquivo `phpunit.xml.dist`. Você pode editar o arquivo para mudar os valores padrões ou criar um arquivo `phpunit.xml` ` para ajustar a configuração para sua máquina local.

Dica: Armazene o arquivo `phpunit.xml.dist` no seu repositório de códigos e ignore o arquivo `phpunit.xml`.

Por padrão, somente os testes armazenados nos bundles "standard" são rodados pelo comando `phpunit` (standard sendo os testes nos diretórios `src/*/Bundle/Tests` ou `src/*/Bundle/*Bundle/Tests`) Mas você pode

facilmente adicionar mais diretórios. Por exemplo, a seguinte configuração adiciona os testes de um bundle de terceiros instalado:

```
<!-- hello/phpunit.xml.dist -->
<testsuites>
    <testsuite name="Project Test Suite">
        <directory>../src/*/*Bundle/Tests</directory>
        <directory>../src/Acme/Bundle/*Bundle/Tests</directory>
    </testsuite>
</testsuites>
```

Para incluir outros diretórios no code coverage, edite também a sessção `<filter>`:

```
<filter>
    <whitelist>
        <directory>../src</directory>
        <exclude>
            <directory>../src/*/*Bundle/Resources</directory>
            <directory>../src/*/*Bundle/Tests</directory>
            <directory>../src/Acme/Bundle/*Bundle/Resources</directory>
            <directory>../src/Acme/Bundle/*Bundle/Tests</directory>
        </exclude>
    </whitelist>
</filter>
```

Aprenda mais no Cookbook

- `/cookbook/testing/http_authentication`
- `/cookbook/testing/insulating_clients`
- `/cookbook/testing/profiling`

2.1.9 Validação

Validação é uma tarefa muito comum em aplicações web. Dado inserido em formulário precisa ser validado. Dado também precisa ser revalidado antes de ser escrito num banco de dados ou passado a um serviço web.

Symfony2 vem acompanhado com um componente Validator que torna essa tarefa fácil e transparente. Esse componente é baseado na especificação **'JSR303 Bean Validation'_**. O quê ? Uma especificação Java no PHP? Você ouviu corretamente, mas não é tão ruim quanto parece. Vamos olhar como isso pode ser usado no PHP.

As bases da validação

A melhor forma de entender validação é vê-la em ação. Para começar, suponha que você criou um bom e velho objeto PHP que você precisa usar em algum lugar da sua aplicação:

```
// src/Acme/BlogBundle/Entity/Author.php
namespace Acme\BlogBundle\Entity;

class Author
{
    public $name;
}
```

Até agora, essa é somente uma classe comum que serve para alguns propósitos dentro de sua aplicação. O objetivo da validação é avisar você se um dado de um objeto é válido ou não. Para esse trabalho, você irá configura uma lista de regras (chamada *constraints*) em que o objeto deve seguir em ordem para ser validado. Essas regras podem ser especificadas por um número de diferentes formatos (YAML, XML, annotations, ou PHP).

Por exemplo, para garantir que a propriedade `$name` não é vazia, adicione o seguinte:

- *YAML*

```
# src/Acme/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\Author:
    properties:
        name:
            - NotBlank: ~
```

- *Annotations*

```
// src/Acme/BlogBundle/Entity/Author.php
use Symfony\Component\Validator\Constraints as Assert;

class Author
{
    /**
     * @Assert\NotBlank()
     */
    public $name;
}
```

- *XML*

```
<!-- src/Acme/BlogBundle/Resources/config/validation.xml -->
<?xml version="1.0" encoding="UTF-8" ?>
<constraint-mapping xmlns="http://symfony.com/schema/dic/constraint-mapping"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/dic/constraint-mapping http://symfony.com/sche

    <class name="Acme\BlogBundle\Entity\Author">
        <property name="name">
            <constraint name="NotBlank" />
        </property>
    </class>
</constraint-mapping>
```

- *PHP*

```
// src/Acme/BlogBundle/Entity/Author.php

use Symfony\Component\Validator\Mapping\ClassMetadata;
use Symfony\Component\Validator\Constraints\NotBlank;

class Author
{
    public $name;

    public static function loadValidatorMetadata(ClassMetadata $metadata)
    {
        $metadata->addPropertyConstraint('name', new NotBlank());
    }
}
```

Dica:

Propriedades protected e private podem também ser validadas, bem como os métodos

"getter" (veja *validator-constraint-targets*

Usando o serviço `validator`

Próximo passo, para realmente validar um objeto''Author'', use o método `validate` no serviço `validator` (classe `Validator`). A tarefa do `validator` é fácil: ler as restrições (i.e. regras) de uma classe e verificar se o dado no objeto satisfaz ou não aquelas restrições. Se a validação falhar, retorna um array de erros. Observe esse simples exemplo de dentro do controller:

```
use Symfony\Component\HttpFoundation\Response;
use Acme\BlogBundle\Entity\Author;
// ...

public function indexAction()
{
    $author = new Author();
    // ... do something to the $author object

    $validator = $this->get('validator');
    $errors = $validator->validate($author);

    if (count($errors) > 0) {
        return new Response(print_r($errors, true));
    } else {
        return new Response('The author is valid! Yes!');
    }
}
```

Se a propriedade `$name` é vazia, você verá a seguinte mensagem de erro:

```
Acme\BlogBundle\Author.name:
    This value should not be blank
```

Se você inserir um valor na propriedade `name`, aparecerá a feliz mensagem de sucesso.

Dica: A maior parte do tempo, você não irá interagir diretamente com o serviço `validator` ou precisará se preocupar sobre imprimir os erros. A maior parte do tempo, você irá usar a validação indiretamente quando lidar com dados enviados do formulário. Para mais informações, veja: ref:*book-validation-forms*.

Você também poderia passar o conjunto de erros em um template.

```
if (count($errors) > 0) {
    return $this->render('AcmeBlogBundle:Author:validate.html.twig', array(
        'errors' => $errors,
    ));
} else {
    // ...
}
```

Dentro do template, você pode gerar a lista de erros exatamente necessária:

- *Twig*

```
{# src/Acme/BlogBundle/Resources/views/Author/validate.html.twig #}

<h3>The author has the following errors</h3>
<ul>
{% for error in errors %}
    <li>{{ error.message }}</li>
{% endfor %}
</ul>
```

- *PHP*

```
<!-- src/Acme/BlogBundle/Resources/views/Author/validate.html.php -->

<h3>The author has the following errors</h3>
<ul>
<?php foreach ($errors as $error): ?>
    <li><?php echo $error->getMessage() ?></li>
<?php endforeach; ?>
</ul>
```

Nota: Cada erro de validação (chamado de "constraint violation"), é representado por um objeto `ConstraintViolation`.

Validação e formulários

O serviço `validator` pode ser usado a qualquer momento para validar qualquer objeto. Na realidade, entretanto, você irá trabalhar frequentemente com o `validator` indiretamente enquanto trabalhar com formulário. A biblioteca Symfony's form usa o serviço `validator` internamente para validar o objeto oculto após os valores terem sido enviados e fixados. As violações de restrição no objeto são convertidas em objetos `FieldError` que podem ser facilmente exibidos com seu formulário. O tipico fluxo de envio do formulário parece o seguinte dentro do controller:

```
use Acme\BlogBundle\Entity\Author;
use Acme\BlogBundle\Form\AuthorType;
use Symfony\Component\HttpFoundation\Request;
// ...

public function updateAction(Request $request)
{
    $author = new Acme\BlogBundle\Entity\Author();
    $form = $this->createForm(new AuthorType(), $author);

    if ($request->isMethod('POST')) {
        $form->bind($request);

        if ($form->isValid()) {
            // the validation passed, do something with the $author object

            $this->redirect($this->generateUrl('...'));
        }
    }

    return $this->render('BlogBundle:Author:form.html.twig', array(
        'form' => $form->createView(),
```

```
    ));
}
```

Nota: Esse exemplo usa uma classe de formulários `AuthorType` , que não é mostrada aqui.

Para mais informações, veja: doc:*Forms</book/forms>* chapter.

Configuração

O validador do Symfony2 é abilitado por padrão, mas você deve abilitar explicitamente anotações se você usar o método de anotação para especificar suas restrições:

- *YAML*

```
# app/config/config.yml
framework:
    validation: { enable_annotations: true }
```

- *XML*

```
<!-- app/config/config.xml -->
<framework:config>
    <framework:validation enable_annotations="true" />
</framework:config>
```

- *PHP*

```
// app/config/config.php
$container->loadFromExtension('framework', array('validation' => array(
    'enable_annotations' => true,
)));
```

Restrições

O `validator` é designado para validar objtos perante *restrições* (i.e. regras). Em ordem para validar um objeto, simplesmente mapeie uma ou mais restrições para aquela classe e então passe para o serviço `validator`.

Por trás dos bastidores, uma restrição é simplesmente um objeto PHP que faz uma sentença afirmativa. Na vida real, uma restrição poderia ser: "O bolo não deve queimar". No Symfony2, restrições são similares: elas são afirmações que uma condição é verdadeira. Dado um valor, a restriçao irá indicar a você se o valor adere ou não às regras da restrição.

Restições Suportadas

Symfony2 engloba um grande número de restrições mais frequentemente usadas:

Você também pode criar sua própria restrição personalizada. Esse tópico é coberto no artigo do cookbook "Como criar uma Constraint de Validação Personalizada" .

Configuração de restrições

Algumas restrições, como `NotBlank`, são simples como as outras, como a restrição `Choice` , tem várias opções de configuração disponíveis. Suponha que a classe‘‘Author‘‘ tenha outra propriedade, `gender` que possa ser configurado como “male” ou “female”:

- *YAML*

```
# src/Acme/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\Author:
    properties:
        gender:
            - Choice: { choices: [male, female], message: Choose a valid gender. }
```

- *Annotations*

```
// src/Acme/BlogBundle/Entity/Author.php
use Symfony\Component\Validator\Constraints as Assert;

class Author
{
    /**
     * @Assert\Choice(
     *     choices = { "male", "female" },
     *     message = "Choose a valid gender."
     * )
     */
    public $gender;
}
```

- *XML*

```
<!-- src/Acme/BlogBundle/Resources/config/validation.xml -->
<?xml version="1.0" encoding="UTF-8" ?>
<constraint-mapping xmlns="http://symfony.com/schema/dic/constraint-mapping"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/dic/constraint-mapping http://symfony.com/sche

    <class name="Acme\BlogBundle\Entity\Author">
        <property name="gender">
            <constraint name="Choice">
                <option name="choices">
                    <value>male</value>
                    <value>female</value>
                </option>
                <option name="message">Choose a valid gender.</option>
            </constraint>
        </property>
    </class>
</constraint-mapping>
```

- *PHP*

```
// src/Acme/BlogBundle/Entity/Author.php
use Symfony\Component\Validator\Mapping\ClassMetadata;
use Symfony\Component\Validator\Constraints\NotBlank;

class Author
{
```

```
        public $gender;

        public static function loadValidatorMetadata(ClassMetadata $metadata)
        {
            $metadata->addPropertyConstraint('gender', new Choice(array(
                'choices' => array('male', 'female'),
                'message' => 'Choose a valid gender.',
            )));
        }
    }
```

A opção de uma restrição pode sempre ser passada como um array. Algumas restrições, entretanto, também permitem a você passar o valor de uma opção "*default*" no lugar do array. No cado da restrição `Choice` , as opções `choices` podem ser especificadas dessa forma.

- *YAML*

```
    # src/Acme/BlogBundle/Resources/config/validation.yml
    Acme\BlogBundle\Entity\Author:
        properties:
            gender:
                - Choice: [male, female]
```

- *Annotations*

```
    // src/Acme/BlogBundle/Entity/Author.php
    use Symfony\Component\Validator\Constraints as Assert;

    class Author
    {
        /**
         * @Assert\Choice({"male", "female"})
         */
        protected $gender;
    }
```

- *XML*

```
    <!-- src/Acme/BlogBundle/Resources/config/validation.xml -->
    <?xml version="1.0" encoding="UTF-8" ?>
    <constraint-mapping xmlns="http://symfony.com/schema/dic/constraint-mapping"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://symfony.com/schema/dic/constraint-mapping http://symfony.com/sche

        <class name="Acme\BlogBundle\Entity\Author">
            <property name="gender">
                <constraint name="Choice">
                    <value>male</value>
                    <value>female</value>
                </constraint>
            </property>
        </class>
    </constraint-mapping>
```

- *PHP*

```
    // src/Acme/BlogBundle/Entity/Author.php
    use Symfony\Component\Validator\Mapping\ClassMetadata;
    use Symfony\Component\Validator\Constraints\Choice;
```

```
class Author
{
    protected $gender;

    public static function loadValidatorMetadata(ClassMetadata $metadata)
    {
        $metadata->addPropertyConstraint('gender', new Choice(array('male', 'female')));
    }
}
```

Isso significa simplesmente fazer a configuração da opção mais comum de uma restrição mais curta e rápida.

Se você está incerto de como especificar uma opção, ou verifique a documentação da API para a restrição ou faça de forma segura sempre passando um array de opções (o primeiro método mostrado acima).

Escopos da restrição

Restrições podem ser aplicadas a uma propriedade de classe (e.g. `name`) ou um método getter público (e.g. `getFullName`). O primeiro é mais comum e fácil de usar, mas o segundo permite você especificar regras de validação mais complexas.

Propriedades

Validar as propriedades de uma classe é a técnica de validação mais básica.Symfony2 permite a você validar propriedades private, protected ou public. A próxima listagem mostra a você como configurar a propriedade `$firstName` da classe `Author` para ter ao menos 3 caracteres.

- *YAML*

```
# src/Acme/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\Author:
    properties:
        firstName:
            - NotBlank: ~
            - MinLength: 3
```

- *Annotations*

```
// Acme/BlogBundle/Entity/Author.php
use Symfony\Component\Validator\Constraints as Assert;

class Author
{
    /**
     * @Assert\NotBlank()
     * @Assert\MinLength(3)
     */
    private $firstName;
}
```

- *XML*

```
<!-- src/Acme/BlogBundle/Resources/config/validation.xml -->
<class name="Acme\BlogBundle\Entity\Author">
    <property name="firstName">
        <constraint name="NotBlank" />
        <constraint name="MinLength">3</constraint>
```

```
        </property>
    </class>
```

- *PHP*

```
    // src/Acme/BlogBundle/Entity/Author.php
    use Symfony\Component\Validator\Mapping\ClassMetadata;
    use Symfony\Component\Validator\Constraints\NotBlank;
    use Symfony\Component\Validator\Constraints\MinLength;

    class Author
    {
        private $firstName;

        public static function loadValidatorMetadata(ClassMetadata $metadata)
        {
            $metadata->addPropertyConstraint('firstName', new NotBlank());
            $metadata->addPropertyConstraint('firstName', new MinLength(3));
        }
    }
```

Getters

Restrições podem também ser aplicadas no método de retorno de um valor.Symfony2 permite a você adicionar uma restrição para qualquer método public cujo nome comec com "get" ou "is". Nesse guia, ambos os tipos de métodos são referidos como "getters".

O benefício dessa técnica é que permite a você validar seu objeto dinamicamente. Por exemplo, suponha que você queira ter certeza que um campo de senha não coincida com o primeiro nome do usuário (por motivos de segurança). Você pode fazer isso criando um método `isPasswordLegal`, e então afirmando que esse método deva retornar ''true'':

- *YAML*

```
    # src/Acme/BlogBundle/Resources/config/validation.yml
    Acme\BlogBundle\Entity\Author:
        getters:
            passwordLegal:
                - "True": { message: "The password cannot match your first name" }
```

- *Annotations*

```
    // src/Acme/BlogBundle/Entity/Author.php
    use Symfony\Component\Validator\Constraints as Assert;

    class Author
    {
        /**
         * @Assert\True(message = "The password cannot match your first name")
         */
        public function isPasswordLegal()
        {
            // return true or false
        }
    }
```

- *XML*

```
<!-- src/Acme/BlogBundle/Resources/config/validation.xml -->
<class name="Acme\BlogBundle\Entity\Author">
    <getter property="passwordLegal">
        <constraint name="True">
            <option name="message">The password cannot match your first name</option>
        </constraint>
    </getter>
</class>
```

- *PHP*

```
// src/Acme/BlogBundle/Entity/Author.php
use Symfony\Component\Validator\Mapping\ClassMetadata;
use Symfony\Component\Validator\Constraints\True;

class Author
{
    public static function loadValidatorMetadata(ClassMetadata $metadata)
    {
        $metadata->addGetterConstraint('passwordLegal', new True(array(
            'message' => 'The password cannot match your first name',
        )));
    }
}
```

Agora, crie o método `isPasswordLegal()` , e inclua a lógica que você precisa:

```
public function isPasswordLegal()
{
    return ($this->firstName != $this->password);
}
```

Nota: Com uma visão apurada, você irá perceber que o prefixo do getter (“get” ou “is”) é omitido no mapeamento. Isso permite você mover a restrição para uma propriedade com o mesmo nome mais tarde (ou vice-versa) sem mudar sua lógica de validação.

Classes

Algumas restrições aplicam para a classe inteira ser validada. Por exemplo, a restrição `Callback` é uma restrição genérica que é aplicada para a própria classe. Quando a classe é validada, métodos especificados por aquela restrição são simplesmente executadas então cada um pode prover uma validação mais personalizada.

Grupos de validação

Até agora, você foi capaz de adicionar restrições a uma classe e perguntar se aquela classe passa ou não por todas as restrições definidas. Em alguns casos, entretanto, você precisará validar um objeto a somente *algumas* das restrições naqula classe. Para fazer isso, você pode organizar cada restrição dentro de um ou mais “grupos de validação”, e então aplicar validação a apenas um grupo de restrições.

Por exemplo, suponha que você tenha uma classe `User` , que é usada tanto quando um usuário registra e quando um usuário atualiza sua informações de contato posteriormente:

- *YAML*

```yaml
# src/Acme/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\User:
    properties:
        email:
            - Email: { groups: [registration] }
        password:
            - NotBlank: { groups: [registration] }
            - MinLength: { limit: 7, groups: [registration] }
        city:
            - MinLength: 2
```

- *Annotations*

```php
// src/Acme/BlogBundle/Entity/User.php
namespace Acme\BlogBundle\Entity;

use Symfony\Component\Security\Core\User\UserInterface;
use Symfony\Component\Validator\Constraints as Assert;

class User implements UserInterface
{
    /**
    * @Assert\Email(groups={"registration"})
    */
    private $email;

    /**
    * @Assert\NotBlank(groups={"registration"})
    * @Assert\MinLength(limit=7, groups={"registration"})
    */
    private $password;

    /**
    * @Assert\MinLength(2)
    */
    private $city;
}
```

- *XML*

```xml
<!-- src/Acme/BlogBundle/Resources/config/validation.xml -->
<class name="Acme\BlogBundle\Entity\User">
    <property name="email">
        <constraint name="Email">
            <option name="groups">
                <value>registration</value>
            </option>
        </constraint>
    </property>
    <property name="password">
        <constraint name="NotBlank">
            <option name="groups">
                <value>registration</value>
            </option>
        </constraint>
        <constraint name="MinLength">
            <option name="limit">7</option>
            <option name="groups">
                <value>registration</value>
```

```
                </option>
            </constraint>
        </property>
        <property name="city">
            <constraint name="MinLength">7</constraint>
        </property>
    </class>
```

- *PHP*

```
    // src/Acme/BlogBundle/Entity/User.php
    namespace Acme\BlogBundle\Entity;

    use Symfony\Component\Validator\Mapping\ClassMetadata;
    use Symfony\Component\Validator\Constraints\Email;
    use Symfony\Component\Validator\Constraints\NotBlank;
    use Symfony\Component\Validator\Constraints\MinLength;

    class User
    {
        public static function loadValidatorMetadata(ClassMetadata $metadata)
        {
            $metadata->addPropertyConstraint('email', new Email(array(
                'groups' => array('registration')
            )));

            $metadata->addPropertyConstraint('password', new NotBlank(array(
                'groups' => array('registration')
            )));
            $metadata->addPropertyConstraint('password', new MinLength(array(
                'limit'  => 7,
                'groups' => array('registration')
            )));

            $metadata->addPropertyConstraint('city', new MinLength(3));
        }
    }
```

Com essa configuração, existem dois grupos de validação:

- `Default` - contém as restrições não atribuidas a qualquer outro grupo;
- `registration` - Contém as restrições somente nos campos `email` e `password`.

Para avisar o validador a usar um grupo específico, passe um ou mais nomes de trupos como um segundo argumento para o método‘‘validate()‘‘

$errors = $validator->validate($author, array(‘registration’));

Claro, você irá frequentemente trabalhar com validação indiretamnte por meio da biblioteca do formulário. Para informações em como usar grupos de validação dentro de formulários, veja *Grupos de Validação*.

Validando Valores e Arrays

Até agora, você viu como pode validar objetos inteiros. Mas às vezes, você somente quer validar um valor simples - como verificar se uma string é um endereço de e-mail válido. Isso é realmente muito fácil de fazer. De dentro do controller, parece com isso:

// add this to the top of your class use SymfonyComponentValidatorConstraintsEmail;

```
public function addEmailAction($email) {
    $emailConstraint = new Email(); // all constraint "options" can be set this way
    $emailConstraint->message = 'Invalid email address';
    // use the validator to validate the value $errorList = $this->get('validator')-
    >validateValue($email, $emailConstraint);
    if (count($errorList) == 0) { // this IS a valid email address, do something
    } else { // this is not a valid email address $errorMessage = $errorList[0]->getMessage()
        // do somethign with the error
    }
    // ...
}
```

Ao chamar `validateValue` no validador, você pode passar um valor bruto e o objeto de restrição que você com o qual você quer validar aquele valor. Uma lista completa de restrições disponíveis - bem como o nome inteiro da classe para cada restrição - está disponível em constraints reference section .

O método `validateValule` retorna um objeto `ConstraintViolationList` , que age como um array de erros. Cada erro na coleção é um objeto : class:*Symfony\Component\Validator\ConstraintViolation* , que contém a mensagem de erro no método *getMessage* dele.

Considerações Finais

O Symfony2 `validator` é uma ferramenta poderosa que pode ser multiplicada para garantir que o dado de qualquer objeto seja "válido". O poder por trás da validação rside em "restrições", que seão regras que você pode aplicar a propriedades ou métodos getter de seus objetos. E enquanto você irá usar mais frequentemente usar a validação do framework indiretamente quando usar formulários, lembre que isso pode ser usado em qualquer lugar para validar qualquer objeto.

Aprenda mais do Cookbook

- Como criar uma Constraint de Validação Personalizada

2.1.10 Formulários

Lidar com formulários HTML é uma das mais comuns - e desafiadoras - tarefas para um desenvolvedor web. O Symfony2 integra um componente de formulário que torna fácil a tarefa de lidar com formulários. Neste capítulo, você vai construir um formulário complexo a partir do zero, aprendendo as características mais importantes da biblioteca de formulários ao longo do caminho.

Nota: O componente de formulário do Symfony é uma biblioteca independente que pode ser utilizada fora de projetos Symfony2. Para mais informações, consulte o Componente de Formulário do Symfony2 no Github.

Criando um formulário simples

Suponha que você está construindo uma aplicação simples de lista de tarefas que precisará exibir "tarefas". Devido aos seus usuários terem que editar e criar tarefas, você precisará construir um formulário. Mas, antes de começar, primeiro vamos focar na classe genérica `Task` que representa e armazena os dados de uma única tarefa:

```php
// src/Acme/TaskBundle/Entity/Task.php
namespace Acme\TaskBundle\Entity;

class Task
{
    protected $task;

    protected $dueDate;

    public function getTask()
    {
        return $this->task;
    }
    public function setTask($task)
    {
        $this->task = $task;
    }

    public function getDueDate()
    {
        return $this->dueDate;
    }
    public function setDueDate(\DateTime $dueDate = null)
    {
        $this->dueDate = $dueDate;
    }
}
```

Nota: Se você está codificando junto com este exemplo, crie o `AcmeTaskBundle` primeiro, executando o seguinte comando (e aceite todas as opções padrão):

```
php app/console generate:bundle --namespace=Acme/TaskBundle
```

Essa classe é um "antigo objeto PHP simples", porque, até agora, não tem nada a ver com Symfony ou qualquer outra biblioteca. É simplesmente um objeto PHP normal que, diretamente resolve um problema no interior da *sua* aplicação (ou seja, a necessidade de representar uma tarefa na sua aplicação). Claro, até o final deste capítulo, você será capaz de enviar dados para uma instância `Task` (através de um formulário HTML), validar os seus dados, e persisti-los para o banco de dados.

Construindo o Formulário

Agora que você já criou a classe `Task`, o próximo passo é criar e renderizar o formulário HTML real. No Symfony2, isto é feito através da construção de um objeto de formulário e, em seguida, renderizando em um template. Por ora, tudo isso pode ser feito dentro de um controlador:

```
// src/Acme/TaskBundle/Controller/DefaultController.php
namespace Acme\TaskBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Acme\TaskBundle\Entity\Task;
use Symfony\Component\HttpFoundation\Request;

class DefaultController extends Controller
{
    public function newAction(Request $request)
```

```
    {
        // create a task and give it some dummy data for this example
        $task = new Task();
        $task->setTask('Write a blog post');
        $task->setDueDate(new \DateTime('tomorrow'));

        $form = $this->createFormBuilder($task)
            ->add('task', 'text')
            ->add('dueDate', 'date')
            ->getForm();

        return $this->render('AcmeTaskBundle:Default:new.html.twig', array(
            'form' => $form->createView(),
        ));
    }
}
```

Dica: Este exemplo mostra como construir o seu formulário diretamente no controlador. Mais tarde, na seção "*Criando classes de formulário*", você aprenderá como construir o seu formulário em uma classe independente, que é o recomendado pois torna o seu formulário reutilizável.

A criação de um formulário requer relativamente pouco código porque os objetos de formulário do Symfony2 são construídos com um "construtor de formulários". A finalidade do construtor de formulários é permitir que você escreva "receitas" simples de formulários, e ele fazer todo o trabalho pesado, de, realmente, construir o formulário.

Neste exemplo, você acrescentou dois campos ao seu formulário - `task` e `dueDate` - que correspondem as propriedades `task` e `dueDate` da classe `Task`. Você também atribuiu a cada um deles um "type" (exemplo: `text`, `date`), que, entre outras coisas, determina qual(ais) tag(s) HTML de formulário serão renderizadas para esse campo.

O Symfony2 vem com muitos tipos embutidos, que serão discutidos em breve (veja *Tipos de campos integrados (Built-in)*).

Renderizando o Formulário

Agora que o formulário foi criado, o próximo passo é renderizá-lo. Isto é feito passando um objeto "view" especial para o seu template (note o `$form->createView()` no controlador acima) e usando um conjunto de funções helper para o formulário:

- *Twig*

```
{# src/Acme/TaskBundle/Resources/views/Default/new.html.twig #}

<form action="{{ path('task_new') }}" method="post" {{ form_enctype(form) }}>
    {{ form_widget(form) }}

    <input type="submit" />
</form>
```

- *PHP*

```
<!-- src/Acme/TaskBundle/Resources/views/Default/new.html.php -->

<form action="<?php echo $view['router']->generate('task_new') ?>" method="post" <?php echo $vie
    <?php echo $view['form']->widget($form) ?>
```

```
        <input type="submit" />
    </form>
```

Nota: Este exemplo assume que você criou uma rota chamada `task_new` que aponta para o controlador `AcmeTaskBundle:Default:new` o qual foi criado anteriormente.

É isso! Ao imprimir o `form_widget(form)`, cada campo do formulário é renderizado, juntamente com uma label e uma mensagem de erro (se houver). Fácil assim, embora não muito flexível (ainda). Normalmente, você desejará renderizar cada campo do formulário individualmente, pois poderá controlar como será a aparência do formulário. Você aprenderá como fazer isso na seção "*Renderizando um formulário em um Template*".

Antes de prosseguirmos, observe como o campo input `task` renderizado tem o valor da propriedade `task` do objeto `$task` (Ex. "Write a blog post"). Este é o primeiro trabalho de um formulário: pegar os dados de um objeto e traduzi-lo em um formato que seja adequado para ser renderizado em um formulário HTML.

Dica: O sistema de formulários é inteligente o suficiente para acessar o valor da propriedade protegida `task` através dos métodos `getTask()` e `setTask()` na classe `Task`. A menos que a propriedade seja pública, ela *deve* ter um método "getter" e "setter" para que o componente de formulário possa obter e definir os dados na propriedade. Para uma propriedade Boolean, você pode usar um método "isser" (por exemplo, `isPublished()`) em vez de um getter (Ex. `getPublished()`).

Manipulando o envio de formulários

O segundo trabalho de um formulário é traduzir os dados enviados pelo usuário de volta as propriedades de um objeto. Para que isso aconteça, os dados enviados pelo usuário devem ser vinculados (bound) ao formulário. Adicione as seguintes funcionalidades no seu controlador:

```
// ...

public function newAction(Request $request)
{
    // just setup a fresh $task object (remove the dummy data)
    $task = new Task();

    $form = $this->createFormBuilder($task)
        ->add('task', 'text')
        ->add('dueDate', 'date')
        ->getForm();

    if ($request->isMethod('POST')) {
        $form->bind($request);
```

```
        if ($form->isValid()) {
            // perform some action, such as saving the task to the database

            return $this->redirect($this->generateUrl('task_success'));
        }
    }

    // ...
}
```

Novo na versão 2.1: O método `bind` tornou-se mais flexível no Symfony 2.1. Ele aceita agora os dados brutos do cliente (como antes) ou um objeto Request do Symfony. Ele é preferido ao invés do método obsoleto `bindRequest`.

Agora, quando enviar o formulário, o controlador vincula (bind) ao formulário os dados enviados, que traduz os dados de volta as propriedades `task` e `dueDate` do objeto `$task`. Isso tudo acontece através do método `bind()`.

Nota: Assim que o `bind()` é chamado, os dados enviados são transferidos imediatamente para o objeto implícito. Isso acontece independentemente dos dados implícitos serem realmente válidos.

Este controlador segue um padrão comum para a manipulação de formulários, e possui três caminhos possíveis:

1. Inicialmente quando se carrega a página em um navegador, o método de solicitação (request) é `GET` e o formulário é simplesmente criado e renderizado;
2. Quando o usuário envia o formulário (Ex., o método é `POST`) mas os dados não são válidos (a validação será discutida na próxima seção), o formulário é vinculado (bound) e então processado, desta vez exibindo todos os erros de validação;
3. Quando o usuário envia o formulário com dados válidos, o formulário é vinculado (bound) e você tem a oportunidade de executar algumas ações usando o objeto `$task` (por exemplo, persisti-lo para o banco de dados) antes de redirecionar o usuário para outra página (por exemplo, uma página de "obrigado" ou "sucesso").

Nota: Redirecionar o usuário após o envio bem sucedido do formulário impede que ele, ao clicar em "atualizar", reenvie os dados do formulário.

Validação do formulário

Na seção anterior, você aprendeu como um formulário pode ser enviado com dados válidos ou inválidos. No Symfony2, a validação é aplicada ao objeto implícito (Ex., `Task`). Em outras palavras, a questão não é se o "formulário" é válido, mas se o objeto `$task` é válido após a aplicação dos dados enviados pelo formulário. A chamada `$form->isValid()` é um atalho que pergunta ao objeto `$task` se ele possui ou não dados válidos.

A validação é feita adicionando um conjunto de regras (chamadas *constraints*) à uma classe. Para ver isso em ação, adicione *constraints* de validação para que o campo `task` não deve ser vazio e o campo `dueDate` não deve ser vazio e deve ser um objeto DateTime válido.

- *YAML*

```
    # Acme/TaskBundle/Resources/config/validation.yml
    Acme\TaskBundle\Entity\Task:
        properties:
            task:
                - NotBlank: ~
            dueDate:
```

```
            - NotBlank: ~
            - Type: \DateTime
```

- *Annotations*

```
// Acme/TaskBundle/Entity/Task.php
use Symfony\Component\Validator\Constraints as Assert;

class Task
{
    /**
     * @Assert\NotBlank()
     */
    public $task;

    /**
     * @Assert\NotBlank()
     * @Assert\Type("\DateTime")
     */
    protected $dueDate;
}
```

- *XML*

```
<!-- Acme/TaskBundle/Resources/config/validation.xml -->
<class name="Acme\TaskBundle\Entity\Task">
    <property name="task">
        <constraint name="NotBlank" />
    </property>
    <property name="dueDate">
        <constraint name="NotBlank" />
        <constraint name="Type">
            <value>\DateTime</value>
        </constraint>
    </property>
</class>
```

- *PHP*

```
// Acme/TaskBundle/Entity/Task.php
use Symfony\Component\Validator\Mapping\ClassMetadata;
use Symfony\Component\Validator\Constraints\NotBlank;
use Symfony\Component\Validator\Constraints\Type;

class Task
{
    // ...

    public static function loadValidatorMetadata(ClassMetadata $metadata)
    {
        $metadata->addPropertyConstraint('task', new NotBlank());

        $metadata->addPropertyConstraint('dueDate', new NotBlank());
        $metadata->addPropertyConstraint('dueDate', new Type('\DateTime'));
    }
}
```

É isso! Se você reenviar o formulário com dados inválidos, verá os erros correspondentes exibidos com o formulário.

Validação HTML5

Com o HTML5, muitos navegadores podem, nativamente, impor certas *constraints* de validação no lado do cliente. A validação mais comum é ativada renderizando um atributo `required` em campos que são obrigatórios. Para navegadores que suportam HTML5, isso irá resultar em uma mensagem nativa do navegador sendo exibida se o usuário tentar enviar o formulário com o campo em branco.
Os formulários gerados podem aproveitar ao máximo esta nova funcionalidade, adicionando atributos HTML que disparam a validação. A validação ao lado do cliente, entretanto, pode ser desativada ao adicionar o atributo `novalidate` na tag `form` ou `formnovalidate` na tag submit. Isto é especialmente útil quando você quiser testar suas *constraints* de validação ao lado do servidor, mas estão sendo impedidas pelo seu navegador, por exemplo, ao enviar campos em branco.

A validação é um recurso muito poderoso do Symfony2 e tem seu próprio capítulo dedicado.

Grupos de Validação

Dica: Se você não estiver usando *grupos de validação*, então, você pode pular esta seção.

Se o seu objeto aproveita a *grupos de validação*, você precisa especificar qual(ais) grupo(s) de validação seu formulário deve usar:

```
$form = $this->createFormBuilder($users, array(
    'validation_groups' => array('registration'),
))->add(...)
;
```

Se você está criando *classes de formulário* (uma boa prática), então você precisa adicionar o seguinte ao método `setDefaultOptions()`:

```
use Symfony\Component\OptionsResolver\OptionsResolverInterface;

public function setDefaultOptions(OptionsResolverInterface $resolver)
{
    $resolver->setDefaults(array(
        'validation_groups' => array('registration')
    ));
}
```

Em ambos os casos, *apenas* o grupo de validação `registration` será usado para validar o objeto implícito.

Grupos com base nos dados submetidos

Novo na versão 2.1: A capacidade de especificar um callback ou Closure no `validation_groups` é novo na versão 2.1

Se você precisar de alguma lógica avançada para determinar os grupos de validação (por exemplo, com base nos dados submetidos), você pode definir a opção `validation_groups` para um `array callback` ou uma `Closure`:

```
use Symfony\Component\OptionsResolver\OptionsResolverInterface;

public function setDefaultOptions(OptionsResolverInterface $resolver)
{
    $resolver->setDefaults(array(
        'validation_groups' => array('Acme\\AcmeBundle\\Entity\\Client', 'determineValidationGroups')
```

```
    ));
}
```

Isso irá chamar o método estático `determineValidationGroups()` na classe `Client` após o formulário ser vinculado (bound), mas antes da validação ser executada. O objeto do formulário é passado como um argumento para esse método (veja o exemplo seguinte). Você também pode definir toda a lógica inline usando uma Closure:

```
use Symfony\Component\Form\FormInterface;
use Symfony\Component\OptionsResolver\OptionsResolverInterface;

public function setDefaultOptions(OptionsResolverInterface $resolver)
{
    $resolver->setDefaults(array(
        'validation_groups' => function(FormInterface $form) {
            $data = $form->getData();
            if (Entity\Client::TYPE_PERSON == $data->getType()) {
                return array('person')
            } else {
                return array('company');
            }
        },
    ));
}
```

Tipos de campos integrados (Built-in)

O Symfony vem, por padrão, com um grande grupo de tipos de campos que cobrem todos os os campos comuns de formulário e tipos de dados que você vai encontrar:

Você também pode criar os seus próprios tipos de campo personalizados. Este tópico é abordado no artigo "Como Criar um Tipo de Campo de Formulário Personalizado" do cookbook.

Opções dos tipos de campos

Cada tipo de campo possui um número de opções que podem ser usadas para configurá-lo. Por exemplo, o campo `dueDate` é atualmente processado como 3 select boxes. No entanto, o `campo date` pode ser configurado para ser renderizado como uma caixa de texto simples (onde o usuário deve digitar a data como uma string na caixa):

```
->add('dueDate', 'date', array('widget' => 'single_text'))
```

Cada tipo de campo tem um número de opções diferentes que podem ser passadas à ele. Muitas delas são específicas para o tipo de campo e os detalhes podem ser encontrados na documentação de cada tipo.

A opção `required`

A opção mais comum é a opção `required`, que pode ser aplicada à qualquer campo. Por padrão, a opção `required` é definida como `true`, o que significa que os navegadores prontos para o HTML5 aplicarão a validação ao lado do cliente se o campo for deixado em branco. Se você não deseja esse comportamento, defina a opção `required` em seu campo para `false` ou *desabilite a validação HTML5*.
Além disso, note que a configuração da opção `required` para `true` **não** resultará em validação aplicada ao lado do servidor. Em outras palavras, se um usuário enviar um valor em branco para o campo (ou usar um navegador antigo ou web service, por exemplo), ela será aceita como um valor válido, a menos que você utilize a constraint de validação do Symfony `NotBlank` ou `NotNull`.
Em outras palavras, a opção `required` é "agradável", mas a validação verdadeira ao lado do servidor *sempre* deverá ser usada.

Adivinhando o tipo do campo

Agora que você adicionou metadados de validação na classe `Task`, o Symfony já sabe um pouco sobre os seus campos. Se você permitir, o Symfony pode "adivinhar" o tipo do seu campo e configurá-lo para você. Neste exemplo, o Symfony pode adivinhar a partir das regras de validação que o campo `task` é um campo `texto` normal e o campo `dueDate` é um campo `data`:

```
public function newAction()
{
    $task = new Task();

    $form = $this->createFormBuilder($task)
        ->add('task')
        ->add('dueDate', null, array('widget' => 'single_text'))
        ->getForm();
}
```

A "adivinhação" é ativada quando você omitir o segundo argumento do método `add()` (ou se você passar `null` para ele). Se você passar um array de opções como o terceiro argumento (feito para o `dueDate` acima), estas opções são aplicadas ao campo adivinhado.

Cuidado: Se o formulário usa um grupo de validação específico, o adivinhador do tipo de campo ainda vai considerar *todas* as *constraints* de validação quando estiver adivinhando os seus tipos de campos (incluindo as *constraints* que não fazem parte dos grupos de validação sendo utilizados).

Adivinhando as opções dos tipos de campos

Além de adivinhar o "tipo" para um campo, o Symfony também pode tentar adivinhar os valores corretos de uma série de opções do campo.

Dica: Quando essas opções são definidas, o campo será renderizado com atributos HTML especiais que fornecem para a validação HTML5 ao lado do cliente. Entretanto, ele não gera as *constraints* equivalentes ao lado do servidor (Ex. `Assert\MaxLength`). E, embora você precisará adicionar manualmente a validação ao lado do servidor, essas opções de tipo de campo podem, então, ser adivinhadas a partir dessa informação.

- `required`: A opção `required` pode ser adivinhada com base nas regras de validação (ou seja, o campo é `NotBlank` ou `NotNull`) ou metadados do Doctrine (ou seja, é o campo é `nullable`). Isto é muito útil, pois a sua validação ao lado do cliente irá corresponder automaticamente as suas regras de validação.

- `min_length`: Se o campo é uma espécie de campo de texto, então, a opção `min_length` pode ser adivinhada a partir das *constraints* de validação (se o `MinLength` ou `Min` é usado) ou a partir dos metadados do Doctrine (através do tamanho do campo).
- `max_length`: Semelhante ao `min_length`, o tamanho máximo também pode ser adivinhado.

Nota: Estas opções de campo são adivinhadas *apenas* se você estiver usando o Symfony para adivinhar o tipo de campo (ou seja, omitir ou passar `null` como o segundo argumento para o `add()`).

Se você desejar modificar um dos valores adivinhados, você pode sobrescrevê-lo passando a opção no array de opções do campo:

```
->add('task', null, array('min_length' => 4))
```

Renderizando um formulário em um Template

Até agora, você viu como um formulário inteiro pode ser renderizado com apenas uma linha de código. Claro, você geralmente precisará de muito mais flexibilidade quando estiver renderizando:

- *Twig*

```
{# src/Acme/TaskBundle/Resources/views/Default/new.html.twig #}

<form action="{{ path('task_new') }}" method="post" {{ form_enctype(form) }}>
    {{ form_errors(form) }}

    {{ form_row(form.task) }}
    {{ form_row(form.dueDate) }}

    {{ form_rest(form) }}

    <input type="submit" />
</form>
```

- *PHP*

```
<!-- // src/Acme/TaskBundle/Resources/views/Default/newAction.html.php -->

<form action="<?php echo $view['router']->generate('task_new') ?>" method="post" <?php echo $vie
    <?php echo $view['form']->errors($form) ?>

    <?php echo $view['form']->row($form['task']) ?>
    <?php echo $view['form']->row($form['dueDate']) ?>

    <?php echo $view['form']->rest($form) ?>

    <input type="submit" />
</form>
```

Vamos dar uma olhada em cada parte:

- `form_enctype(form)` - Se pelo menos um campo for um campo para upload de arquivo, ele irá renderizar o `enctype="multipart/form-data"` obrigatório;
- `form_errors(form)` - Renderiza quaisquer erros globais para todo o formulário (erros específicos de campos são exibidos ao lado de cada campo);

- `form_row(form.dueDate)` - Renderiza a label, qualquer erro, e o widget HTML do formulário para o campo informado (Ex. `dueDate`), por padrão, um elemento `div`;
- `form_rest(form)` - Renderiza quaisquer campos que ainda não tenham sido renderizados. Geralmente é uma boa idéia fazer uma chamada deste helper na parte inferior de cada formulário (no caso de você ter esquecido algum campo ou não quer se preocupar em renderizar manualmente os campos ocultos). Este helper também é útil para aproveitar a *Proteção CSRF* automática.

A maioria do trabalho é feito pelo helper `form_row`, que renderiza a label, os erros e widgets HTML do formulário para cada campo dentro de uma tag `div` por padrão. Na seção *Tematizando os formulários*, você aprenderá como a saída do `form_row` pode ser personalizada em muitos níveis diferentes.

Dica: Você pode acessar os dados atuais do seu formulário via `form.vars.value`:

- *Twig*

```
{{ form.vars.value.task }}
```

- *PHP*

```
<?php echo $view['form']->get('value')->getTask() ?>
```

Renderizando cada campo manualmente

O helper `form_row` é ótimo porque você pode renderizar rapidamente cada campo de seu formulário (e também é possível personalizar a marcação utilizada para a "linha"). Mas, como a vida nem sempre é tão simples, você também pode renderizar cada campo inteiramente à mão. O produto final do que segue é o mesmo de quando você usou o helper `form_row`:

- *Twig*

```
{{ form_errors(form) }}

<div>
    {{ form_label(form.task) }}
    {{ form_errors(form.task) }}
    {{ form_widget(form.task) }}
</div>

<div>
    {{ form_label(form.dueDate) }}
    {{ form_errors(form.dueDate) }}
    {{ form_widget(form.dueDate) }}
</div>

{{ form_rest(form) }}
```

- *PHP*

```
<?php echo $view['form']->errors($form) ?>

<div>
    <?php echo $view['form']->label($form['task']) ?>
    <?php echo $view['form']->errors($form['task']) ?>
    <?php echo $view['form']->widget($form['task']) ?>
</div>
```

```
    <div>
        <?php echo $view['form']->label($form['dueDate']) ?>
        <?php echo $view['form']->errors($form['dueDate']) ?>
        <?php echo $view['form']->widget($form['dueDate']) ?>
    </div>

    <?php echo $view['form']->rest($form) ?>
```

Se a label auto-gerada para um campo não estiver correta, você pode especificá-la explicitamente:

- *Twig*

```
    {{ form_label(form.task, 'Task Description') }}
```

- *PHP*

```
    <?php echo $view['form']->label($form['task'], 'Task Description') ?>
```

Finalmente, alguns tipos de campos tem opções de renderização adicionais que podem ser passadas para o widget. Estas opções estão documentadas com cada tipo, mas uma opção em comum é o `attr`, que permite modificar atributos no elemento do formulário. O seguinte código adiciona a classe `task_field` para o campo texto de entrada renderizado:

- *Twig*

```
    {{ form_widget(form.task, { 'attr': {'class': 'task_field'} }) }}
```

- *PHP*

```
    <?php echo $view['form']->widget($form['task'], array(
        'attr' => array('class' => 'task_field'),
    )) ?>
```

Referência de funções dos templates Twig

Se você está usando o Twig, uma referência completa das funções de renderização do formulário está disponível no `manual de referência`. Leia ele para saber tudo sobre os helpers disponíveis e as opções que podem ser usadas com cada um.

Criando classes de formulário

Como você viu, um formulário pode ser criado e usado diretamente em um controlador. No entanto, uma prática melhor é construir o formulário separadamente, em uma classe PHP independente, que poderá, então, ser reutilizada em qualquer lugar na sua aplicação. Crie uma nova classe que vai abrigar a lógica da construção do formulário de tarefas:

```
// src/Acme/TaskBundle/Form/Type/TaskType.php

namespace Acme\TaskBundle\Form\Type;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;

class TaskType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
```

```
        $builder->add('task');
        $builder->add('dueDate', null, array('widget' => 'single_text'));
    }

    public function getName()
    {
        return 'task';
    }
}
```

Esta nova classe contém todas as orientações necessárias para criar o formulário de tarefas (Note que o método `getName()` deve retornar um identificador exclusivo para esse "tipo" do formulário). Ele pode ser usado para construir rapidamente um objeto de formulário no controlador:

```
// src/Acme/TaskBundle/Controller/DefaultController.php

// add this new use statement at the top of the class
use Acme\TaskBundle\Form\Type\TaskType;

public function newAction()
{
    $task = // ...
    $form = $this->createForm(new TaskType(), $task);

    // ...
}
```

Colocando a lógica do formulário em sua própria classe significa que o formulário pode ser facilmente reutilizado em outros lugares no seu projeto. Esta é a melhor forma de criar formulários, mas, a decisão final depende de você.

Setando o `data_class`

Todo formulário precisa saber o nome da classe que contém os dados implícitos (Ex. `Acme\TaskBundle\Entity\Task`). Normalmente, ele é apenas adivinhado com base no objeto passado no segundo argumento para o `createForm` (Ex. `$task`). Mais tarde, quando você iniciar nos formulários embutidos, isto não será suficiente. Então, embora nem sempre necessário, é geralmente uma boa idéia especificar explicitamente a opção `data_class` adicionando o seguinte à sua classe type de formulário:

```
use Symfony\Component\OptionsResolver\OptionsResolverInterface;

public function setDefaultOptions(OptionsResolverInterface $resolver)
{
    $resolver->setDefaults(array(
        'data_class' => 'Acme\TaskBundle\Entity\Task',
    ));
}
```

Dica: Ao mapear formulários para objetos, todos os campos são mapeados. Qualquer campo do formulário que não existe no objeto mapeado irá fazer com que uma exceção seja gerada.

Nos casos em que você precisa de campos extras na formulário (por exemplo: um checkbox "você concorda com os termos") que não será mapeado para o objeto implícito, você precisa definir a opção property_path como `false`:

```
use Symfony\Component\Form\FormBuilderInterface;

public function buildForm(FormBuilderInterface $builder, array $options)
```

```
{
    $builder->add('task');
    $builder->add('dueDate', null, array('property_path' => false));
}
```

Além disso, se houver quaiquer campos do formulário que não estão incluídos nos dados submetidos, esses campos serão definidos explicitamente como `null`.

Os dados do campo podem ser acessados em um controlador com:

```
$form->get('dueDate')->getData();
```

Formulários e o Doctrine

O objetivo de um formulário é traduzir os dados de um objeto (Ex. `Task`) para um formulário HTML e, em seguida, traduzir os dados enviados pelo usuário de volta ao objeto original. Como tal, o tópico da persistência do objeto `Task` no banco de dados é totalmente não relacionado ao tópico de formulários. Mas, se você configurou a classe `Task` para ser persistida através do Doctrine (ou seja, você adicionou *metadados de mapeamento* à ele), então, a persistência após a submissão do formulário pode ser feita quando o formulário é válido:

```
if ($form->isValid()) {
    $em = $this->getDoctrine()->getManager();
    $em->persist($task);
    $em->flush();

    return $this->redirect($this->generateUrl('task_success'));
}
```

Se, por algum motivo, você não tem acesso ao seu objeto `$task` original, você pode buscá-lo a partir do formulário:

```
$task = $form->getData();
```

Para mais informações, consulte o capítulo Doctrine ORM.

A chave para entender é que, quando o formulário é vinculado (bound), os dados submetidos são transferidos imediatamente para o objeto implícito. Se você quiser persistir esses dados, basta persistir o objeto em si (que já contém os dados submetidos).

Formulários embutidos

Muitas vezes, você desejará criar um formulário que vai incluir campos de vários objetos diferentes. Por exemplo, um formulário de inscrição pode conter dados que pertencem a um objeto `User`, bem como, muitos objetos `Address`. Felizmente, isto é fácil e natural com o componente de formulário.

Embutindo um único objeto

Suponha que cada `Task` pertence a um simples objeto `Category`. Inicie, é claro, criando o objeto `Category`:

```
// src/Acme/TaskBundle/Entity/Category.php
namespace Acme\TaskBundle\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Category
```

```
{
    /**
     * @Assert\NotBlank()
     */
    public $name;
}
```

Em seguida, adicione uma nova propriedade `category` na classe `Task`:

```
// ...

class Task
{
    // ...

    /**
     * @Assert\Type(type="Acme\TaskBundle\Entity\Category")
     */
    protected $category;

    // ...

    public function getCategory()
    {
        return $this->category;
    }

    public function setCategory(Category $category = null)
    {
        $this->category = $category;
    }
}
```

Agora que a sua aplicação foi atualizada para refletir as novas exigências, crie uma classe de formulário para que o objeto `Category` possa ser modificado pelo usuário:

```
// src/Acme/TaskBundle/Form/Type/CategoryType.php
namespace Acme\TaskBundle\Form\Type;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolverInterface;

class CategoryType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder->add('name');
    }

    public function setDefaultOptions(OptionsResolverInterface $resolver)
    {
        $resolver->setDefaults(array(
            'data_class' => 'Acme\TaskBundle\Entity\Category',
        ));
    }

    public function getName()
    {
```

```
        return 'category';
    }
}
```

O objetivo final é permitir que a `Category` de uma `Task` possa ser modificada direitamente dentro do próprio formulário da tarefa. Para fazer isso, adicione um campo `category` ao objeto `TaskType` cujo tipo é uma instância da nova classe `CategoryType`:

```
use Symfony\Component\Form\FormBuilderInterface;

public function buildForm(FormBuilderInterface $builder, array $options)
{
    // ...

    $builder->add('category', new CategoryType());
}
```

Os campos do `CategoryType` podem agora ser renderizados juntamente com os campos da classe `TaskType`. Para ativar a validação no CategoryType, adicione a opção `cascade_validation`:

```
public function setDefaultOptions(OptionsResolverInterface $resolver)
{
    $resolver->setDefaults(array(
        'data_class' => 'Acme\TaskBundle\Entity\Category',
        'cascade_validation' => true,
    ));
}
```

Renderize os campos `Category` da mesma forma que os campos originais da `Task`:

- *Twig*

```
{# ... #}

<h3>Category</h3>
<div class="category">
    {{ form_row(form.category.name) }}
</div>

{{ form_rest(form) }}
{# ... #}
```

- *PHP*

```
<!-- ... -->

<h3>Category</h3>
<div class="category">
    <?php echo $view['form']->row($form['category']['name']) ?>
</div>

<?php echo $view['form']->rest($form) ?>
<!-- ... -->
```

Quando o usuário enviar o formulário, os dados submetidos para os campos `Category` são usados para construir uma instância de `Category`, que é então definida no campo `Category` da instância `Task`.

A instância `Category` é acessível naturalmente via `$task->getCategory()` e pode ser persistida no banco de dados ou usada como você precisar.

Embutindo uma coleção de formulários

Você também pode embutir uma coleção de formulários em um formulário (imagine um formulário `Category` com muitos sub-formulários `Product`). Isto é feito usando o tipo de campo `collection`.

Para mais informações consulte no "Como embutir uma Coleção de Formulários" e na collection referência dos tipos de campo.

Tematizando os formulários

Cada parte de como um formulário é renderizado pode ser personalizada. Você está livre para mudar como cada "linha" do formulário é renderizada, alterar a marcação usada para renderizar os erros, ou até mesmo, personalizar como uma tag ‘‘textarea‘ deve ser renderizada. Nada está fora dos limites, e é possível utilizar diferentes personalizações em diferentes lugares.

O Symfony utiliza templates para renderizar todas e cada uma das partes de um formulário, tais como tags `label`, tags `input`, mensagens de erro e tudo mais.

No Twig, cada "fragmento" do formulário é representado por um bloco Twig. Para personalizar qualquer parte de como um formulário é renderizado, você só precisa substituir o bloco apropriado.

No PHP, cada "fragmento" do formulário é renderizado por um arquivo de template individual. Para personalizar qualquer parte de como um formulário é renderizado, você só precisa sobrescrever o template já existente, criando um novo.

Para entender como isso funciona, vamos personalizar o fragmento `form_row` e adicionar um atributo class para o elemento `div` que envolve cada linha. Para fazer isso, crie um novo arquivo template que irá armazenar a marcação nova:

- *Twig*

```
{# src/Acme/TaskBundle/Resources/views/Form/fields.html.twig #}

{% block field_row %}
{% spaceless %}
    <div class="form_row">
        {{ form_label(form) }}
        {{ form_errors(form) }}
        {{ form_widget(form) }}
    </div>
{% endspaceless %}
{% endblock field_row %}
```

- *PHP*

```
<!-- src/Acme/TaskBundle/Resources/views/Form/field_row.html.php -->

<div class="form_row">
    <?php echo $view['form']->label($form, $label) ?>
    <?php echo $view['form']->errors($form) ?>
    <?php echo $view['form']->widget($form, $parameters) ?>
</div>
```

O fragmento `field_row` do formulário é utilizado para renderizar a maioria dos campos através da da função `form_row`. Para dizer ao componente de formulário para utilizar o seu novo fragmento `field_row` definido acima, adicione o seguinte no topo do template que renderiza o formulário:

- *Twig*

```
{# src/Acme/TaskBundle/Resources/views/Default/new.html.twig #}

{% form_theme form 'AcmeTaskBundle:Form:fields.html.twig' %}

<form ...>
```

- *PHP*

```
<!-- src/Acme/TaskBundle/Resources/views/Default/new.html.php -->

<?php $view['form']->setTheme($form, array('AcmeTaskBundle:Form')) ?>

<form ...>
```

A tag `form_theme` (no Twig) "importa" os fragmentos definidos no template informado e utiliza-os quando renderiza o formulário. Em outras palavras, quando a função `form_row` é chamada mais tarde neste template, ela usará o bloco `field_row` de seu tema personalizado (ao invés do bloco padrão `field_row` que vem com o Symfony).

Para personalizar qualquer parte de um formulário, você só precisa substituir o fragmento apropriado. Saber exatamente qual bloco ou arquivo deve-se substituir é o tema da próxima seção.

Novo na versão 2.1: Foi introduzida uma sintaxe alternativa do Twig para `form_theme` no 2.1. Ela aceita qualquer expressão Twig válida (a diferença mais notável está no uso de um array quando utilizar vários temas).

```
{# src/Acme/TaskBundle/Resources/views/Default/new.html.twig #}

{% form_theme form with 'AcmeTaskBundle:Form:fields.html.twig' %}

{% form_theme form with ['AcmeTaskBundle:Form:fields.html.twig', 'AcmeTaskBundle:Form:fields2.html.tw
```

Para uma discussão mais extensiva, consulte Como personalizar a Renderização de Formulários.

Nomeando os fragmentos do formulário

No Symfony, cada parte de um formulário que é renderizada - elementos de formulário HTML, erros, labels, etc - é definida em um tema base, que é uma coleção de blocos no Twig e uma coleção de arquivos de template no PHP.

No Twig, cada bloco necessário é definido em um único arquivo de template (form_div_layout.html.twig) que encontra-se no interior do Twig Bridge. Dentro desse arquivo, você pode ver todos os blocos necessários para renderizar um formulário e todo o tipo de campo padrão.

No PHP, os fragmentos são arquivos de template individuais. Por padrão, eles estão localizados no diretório *Resources/views/Form* do framework bundle (veja no GitHub).

Cada nome de fragmento segue o mesmo padrão básico e é dividido em duas partes, separadas por um único caractere de sublinhado (_). Alguns exemplos são:

- `field_row` - usado pelo `form_row` para renderizar a maioria dos campos;
- `textarea_widget` - usado pelo `form_widget` para renderizar um campo do tipo `textarea`;
- `field_errors` - usado pelo `form_errors` para renderizar os erros para um campo;

Cada fragmento segue o mesmo padrão básico: `type_part`. A porção `type` corresponde ao *tipo* do campo sendo renderizado (Ex. `textarea`, `checkbox`, `date`, etc) enquanto a porção `part` corresponde a *o que* está sendo renderizado (Ex., `label`, `widget`, `errors`, etc). Por padrão, existem 4 *partes* possíveis de um formulário que podem ser renderizadas:

`label`	(Ex. `field_label`)	renderiza label do campo
`widget`	(Ex. `field_widget`)	renderiza a representação HTML do campo
`errors`	(Ex. `field_errors`)	renderiza os errors do campo
`row`	(Ex. `field_row`)	renderiza a linha inteira do campo (label, widget e erros)

Nota: Na verdade, existem outras três *partes* - `rows`, `rest` e `enctype` - mas você raramente ou nunca vai precisar se preocupar em sobrescrevê-las

Ao conhecer o tipo do campo (Ex. ‘‘ textarea‘‘) e qual parte você deseja personalizar (Ex. `widget`), você pode construir o nome do fragmento que precisa ser sobrescrito (Ex. `textarea_widget`).

Herança dos fragmentos de template

Em alguns casos, o fragmento que você deseja personalizar parecerá estar faltando. Por exemplo, não existe um fragmento `textarea_errors` nos temas padrão fornecidos com o Symfony. Então, como são renderizados os erros de um campo textarea?

A resposta é: através do fragmento `field_errors`. Quando o Symfony renderiza os erros para um tipo textarea, ele procura primeiro por um fragmento `textarea_errors` antes de voltar para o fragmento `field_errors`. Cada tipo de campo tem um tipo *pai* (o tipo pai do `textarea` é `field`), e o Symfony usa o fragmento para o tipo pai se o fragmento base não existir.

Então, para substituir os erros para *apenas* os campos `textarea`, copie o fragmento `field_errors`, renomeie para `textarea_errors` e personalize-o. Para sobrescrever a renderização de erro padrão para *todos* os campos, copie e personalize diretamente o fragmento `field_errors`.

Dica: O tipo “pai” de cada tipo de campo está disponível na referência de tipos do formulário para cada tipo de campo.

Tematizando os formulários globalmente

No exemplo acima, você usou o helper `form_theme` (no Twig) para “importar” os fragmentos personalizados *somente* para este formulário. Você também pode dizer ao Symfony para importar as personalizações do formulário para todo o seu projeto.

Twig Para incluir automaticamente os blocos personalizados do template `fields.html.twig` criado anteriormente em *todos* os templates, modifique o seu arquivo de configuração da aplicação:

- *YAML*

```
# app/config/config.yml

twig:
    form:
        resources:
            - 'AcmeTaskBundle:Form:fields.html.twig'
    # ...
```

- *XML*

```
<!-- app/config/config.xml -->

<twig:config ...>
```

```
            <twig:form>
                <resource>AcmeTaskBundle:Form:fields.html.twig</resource>
            </twig:form>
            <!-- ... -->
    </twig:config>
```

- *PHP*

```
    // app/config/config.php

    $container->loadFromExtension('twig', array(
        'form' => array('resources' => array(
            'AcmeTaskBundle:Form:fields.html.twig',
         ))
        // ...
    ));
```

Quaisquer blocos dentro do template `fields.html.twig` agora são usados globalmente para definir a saída do formulário.

Personalizando toda a saída do formulário em um único arquivo com o Twig

No Twig, você também pode personalizar um bloco de formulário diretamente dentro do template onde a personalização é necessária:

```
{% extends '::base.html.twig' %}

{# import "_self" as the form theme #}
{% form_theme form _self %}

{# make the form fragment customization #}
{% block field_row %}
    {# custom field row output #}
{% endblock field_row %}

{% block content %}
    {# ... #}

    {{ form_row(form.task) }}
{% endblock %}
```

A tag `{% form_theme form _self %}` permite que blocos de formulário sejam personalizados diretamente dentro do template que usará essas personalizações. Utilize este método para fazer personalizações de saída do formulário rapidamente, que, somente serão necessárias em um único template.

PHP Para incluir automaticamente os templates personalizados do diretório `Acme/TaskBundle/Resources/views/Form` criado anteriormente em *todos* os templates, modifique o seu arquivo de configuração da aplicação:

- *YAML*

```
    # app/config/config.yml

    framework:
        templating:
            form:
                resources:
```

```
                - 'AcmeTaskBundle:Form'
    # ...
```

- *XML*

```
    <!-- app/config/config.xml -->

    <framework:config ...>
        <framework:templating>
            <framework:form>
                <resource>AcmeTaskBundle:Form</resource>
            </framework:form>
        </framework:templating>
        <!-- ... -->
    </framework:config>
```

- *PHP*

```
    // app/config/config.php

    $container->loadFromExtension('framework', array(
        'templating' => array('form' =>
            array('resources' => array(
                'AcmeTaskBundle:Form',
         )))
        // ...
    ));
```

Qualquer fragmento dentro do diretório `Acme/TaskBundle/Resources/views/Form` agora será usado globalmente para definir a saída do formulário.

Proteção CSRF

CSRF - ou Cross-site request forgery - é um método pelo qual um usuário mal-intencionado tenta fazer com que os seus usuários legítimos, sem saber, enviem dados que eles não pretendem enviar. Felizmente, os ataques CSRF podem ser prevenidos usando um token CSRF dentro do seu formulário.

A boa notícia é que o Symfony, por padrão, incorpora e valida os tokens CSRF automaticamente para você. Isso significa que você pode aproveitar a proteção CSRF sem precisar fazer nada. Na verdade, todo formulário neste capítulo aproveitou a proteção CSRF!

A proteção CSRF funciona adicionando um campo oculto ao seu formulário - chamado `_token` por padrão - que contém um valor que só você e seu usuário sabem. Isto garante que o usuário - e não alguma outra entidade - está enviando os dados. O Symfony automaticamente valida a presença e exatidão deste token.

O campo `_token` é um campo oculto e será automaticamente renderizado se você incluir a função `form_rest()` em seu template, que garante a saída de todos os campos não-renderizados.

O token CSRF pode ser personalizado formulário por formulário. Por exemplo:

```
use Symfony\Component\OptionsResolver\OptionsResolverInterface;

class TaskType extends AbstractType
{
    // ...

    public function setDefaultOptions(OptionsResolverInterface $resolver)
    {
        $resolver->setDefaults(array(
```

```
            'data_class'      => 'Acme\TaskBundle\Entity\Task',
            'csrf_protection' => true,
            'csrf_field_name' => '_token',
            // a unique key to help generate the secret token
            'intention'       => 'task_item',
        ));
    }

    // ...
}
```

Para desativar a proteção CSRF, defina a opção `csrf_protection` para false. As personalizações também podem ser feitas globalmente em seu projeto. Para mais informações veja a seção referência de configuração do formulário .

Nota: A opção `intention` é opcional, mas aumenta muito a segurança do token gerado, tornando-o diferente para cada formulário.

Utilizando um formulário sem uma classe

Na maioria dos casos, um formulário é vinculado a um objeto, e os campos do formulário obtêm e armazenam seus dados nas propriedades desse objeto. Isto foi exatamente o que você viu até agora neste capítulo com a classe *Task*.

Mas, às vezes, você pode desejar apenas utilizar um formulário sem uma classe, e receber um array dos dados submetidos. Isso é realmente muito fácil:

```
// Certifique-se que você importou o namespace Request acima da classe
use Symfony\Component\HttpFoundation\Request
// ...

public function contactAction(Request $request)
{
    $defaultData = array('message' => 'Type your message here');
    $form = $this->createFormBuilder($defaultData)
        ->add('name', 'text')
        ->add('email', 'email')
        ->add('message', 'textarea')
        ->getForm();

        if ($request->isMethod('POST')) {
            $form->bind($request);

            // data is an array with "name", "email", and "message" keys
            $data = $form->getData();
        }

    // ... render the form
}
```

Por padrão, um formulário assume que você deseja trabalhar com arrays de dados, em vez de um objeto. Há exatamente duas maneiras em que você pode mudar esse comportamento e amarrar o formulário à um objeto:

1. Passar um objeto ao criar o formulário (como o primeiro argumento para `createFormBuilder` ou o segundo argumento para `createForm`);
2. Declarar a opção `data_class` no seu formulário.

Se você *não* fizer qualquer uma destas, então o formulário irá retornar os dados como um array. Neste exemplo, uma vez que `$defaultData` não é um objeto (e não foi definida a opção `data_class`), o `$form->getData()` retorna um array.

Dica: Você também pode acessar os valores POST (neste caso, "name") diretamente através do objeto do pedido (`request`), desta forma:

```
$this->get('request')->request->get('name');
```

Esteja ciente, no entanto, que, na maioria dos casos, usar o método getData() é uma melhor escolha, já que retorna os dados (geralmente um objeto), após ele ser transformado pelo framework de formulário.

Adicionando a Validação

A peça que falta é a validação. Normalmente, quando você chama `$form->isValid()`, o objeto é validado através da leitura das *constraints* que você aplicou à classe. Mas, sem uma classe, como você pode adicionar *constraints* para os dados do seu formulário?

A resposta é configurar as *constraints* você mesmo, e passá-las para o seu formulário. A abordagem global é explicada um pouco mais no *validation chapter*, mas aqui está um pequeno exemplo:

```
// import the namespaces above your controller class
use Symfony\Component\Validator\Constraints\Email;
use Symfony\Component\Validator\Constraints\MinLength;
use Symfony\Component\Validator\Constraints\Collection;

$collectionConstraint = new Collection(array(
    'name' => new MinLength(5),
    'email' => new Email(array('message' => 'Invalid email address')),
));

// create a form, no default values, pass in the constraint option
$form = $this->createFormBuilder(null, array(
    'validation_constraint' => $collectionConstraint,
))->add('email', 'email')
    // ...
;
```

Agora, quando você chamar *$form->bind($request)*, a configuração de *constraints* aqui será executada em relação aos dados do seu formulário. Se você estiver usando uma classe de formulário, sobrescreva o método `setDefaultOptions` para especificar a opção:

```
namespace Acme\TaskBundle\Form\Type;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilder;
use Symfony\Component\OptionsResolver\OptionsResolverInterface;
use Symfony\Component\Validator\Constraints\Email;
use Symfony\Component\Validator\Constraints\MinLength;
use Symfony\Component\Validator\Constraints\Collection;

class ContactType extends AbstractType
{
    // ...

    public function setDefaultOptions(OptionsResolverInterface $resolver)
```

```
    {
        $collectionConstraint = new Collection(array(
            'name' => new MinLength(5),
            'email' => new Email(array('message' => 'Invalid email address')),
        ));

        $resolver->setDefaults(array(
            'validation_constraint' => $collectionConstraint
        ));
    }
}
```

Agora, você tem a flexibilidade de criar formulários - com validação - que retorna um array de dados, em vez de um objeto. Na maioria dos casos, é melhor - e certamente mais robusto - ligar (bind) o seu formulário à um objeto. Mas, para formulários simples, esta é uma excelente abordagem.

Considerações finais

Você já conhece todos os blocos de construção necessários para construir formulários complexos e funcionais para a sua aplicação. Ao construir formulários, tenha em mente que a primeira meta de um formulário é traduzir os dados de um objeto (`Task`) para um formulário HTML, para que o usuário possa modificar os dados. O segundo objetivo de um formulário é pegar os dados enviados pelo usuário e reaplicá-los ao objeto.

Ainda há muito mais para aprender sobre o mundo poderoso das formulários, tais como como lidar com uploads de arquivos com o Doctrine ou como criar um formulário onde um número dinâmico de sub-formulários podem ser adicionados (por exemplo, uma lista de tarefas onde você pode continuar a adicionar mais campos antes de enviar via Javascript). Veja estes tópicos no cookbook. Além disso, certifique-se de apoiar-se na documentação de referência de tipos de campo, que inclui exemplos de como usar cada tipo de campo e suas opções.

Aprenda mais no Cookbook

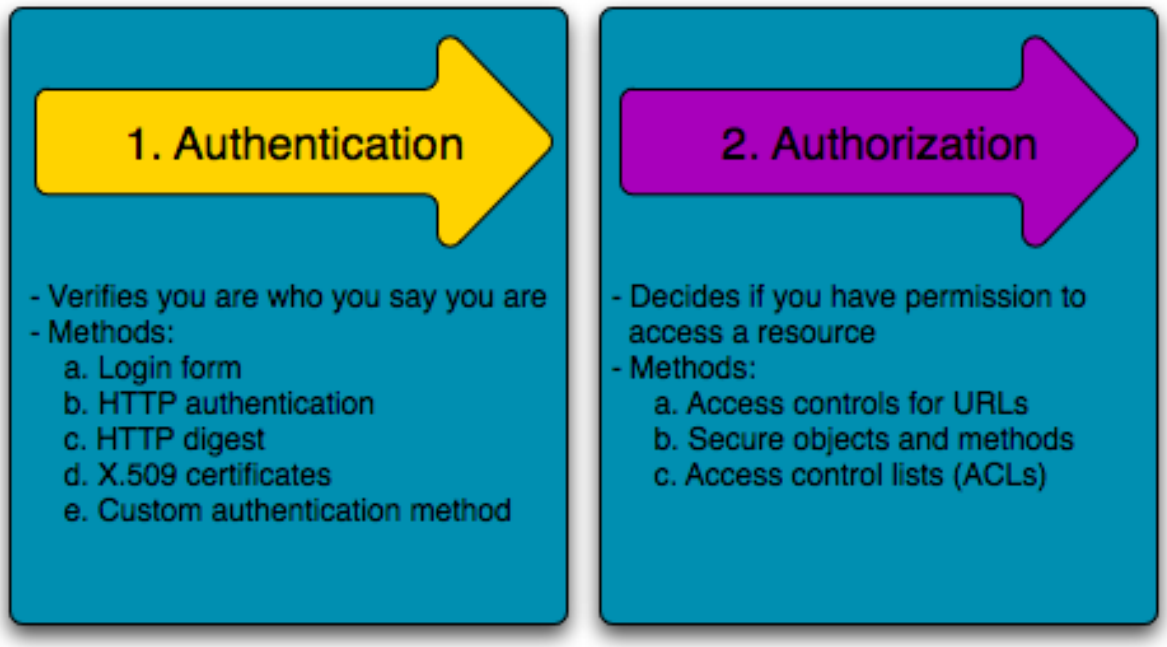
- Como Manipular o Upload de Arquivos com o Doctrine
- `File Field Reference`
- Creating Custom Field Types
- Como personalizar a Renderização de Formulários
- `/cookbook/form/dynamic_form_generation`
- Como usar os Transformadores de Dados

2.1.11 Segurança

Segurança é um processo em dois passos principais. Seu objetivo é evitar que um usuário tenha acesso a um recurso que ele não deveria ter.

No primeiro passo do processo, o sistema de segurança identifica quem o usuário é exigindo que o mesmo envie algum tipo de identificação. Este primeiro passo é chamado **autenticação** e signifcica que o sistema está tentando identificar que é o usuário.

Uma vez que o sistema sabe quem está acessando, o próximo passo é determinar se o usuário pode acessar determinado recurso. Este segundo passo é chamado de **autorização** e significa que o sistema irá checar se o usuário tem permissão para executar determinada ação.



Como a melhor maneira de aprender é com um exemplo, vamos para ele.

Nota: O componente de segurança do Symfony está disponível como uma biblioteca PHP podendo ser utilizada em qualquer projeto PHP.

Exemplo: Autenticação Básica HTTP

O componente de segurança pode ser configurado através da configuração de sua aplicação. Na verdade, a maioria dos esquemas comuns de segurança podem ser conseguidos apenas configurando adequadamente sua aplicação. A configuração a seguir diz ao Symfony para proteger qualquer URL que satisfaça `/admin/*` através da autenticação básica HTTP, solicitando do usuário credenciais (login/senha).

- *YAML*

```
# app/config/security.yml
security:
    firewalls:
        secured_area:
            pattern:    ^/
            anonymous: ~
            http_basic:
                realm: "Secured Demo Area"

    access_control:
        - { path: ^/admin, roles: ROLE_ADMIN }

    providers:
        in_memory:
            memory:
                users:
                    ryan:  { password: ryanpass, roles: 'ROLE_USER' }
```

```
                        admin: { password: kitten, roles: 'ROLE_ADMIN' }

    encoders:
        Symfony\Component\Security\Core\User\User: plaintext
```

- *XML*

```
<!-- app/config/security.xml -->
<srv:container xmlns="http://symfony.com/schema/dic/security"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:srv="http://symfony.com/schema/dic/services"
    xsi:schemaLocation="http://symfony.com/schema/dic/services http://symfony.com/schema/dic/ser

    <config>
        <firewall name="secured_area" pattern="^/">
            <anonymous />
            <http-basic realm="Secured Demo Area" />
        </firewall>

        <access-control>
            <rule path="^/admin" role="ROLE_ADMIN" />
        </access-control>

        <provider name="in_memory">
            <memory>
                <user name="ryan" password="ryanpass" roles="ROLE_USER" />
                <user name="admin" password="kitten" roles="ROLE_ADMIN" />
            </memory>
        </provider>

        <encoder class="Symfony\Component\Security\Core\User\User" algorithm="plaintext" />
    </config>
</srv:container>
```

- *PHP*

```
// app/config/security.php
$container->loadFromExtension('security', array(
    'firewalls' => array(
        'secured_area' => array(
            'pattern' => '^/',
            'anonymous' => array(),
            'http_basic' => array(
                'realm' => 'Secured Demo Area',
            ),
        ),
    ),
    'access_control' => array(
        array('path' => '^/admin', 'role' => 'ROLE_ADMIN'),
    ),
    'providers' => array(
        'in_memory' => array(
            'memory' => array(
                'users' => array(
                    'ryan' => array('password' => 'ryanpass', 'roles' => 'ROLE_USER'),
                    'admin' => array('password' => 'kitten', 'roles' => 'ROLE_ADMIN'),
                ),
            ),
        ),
```

```
        ),
        'encoders' => array(
            'Symfony\Component\Security\Core\User\User' => 'plaintext',
        ),
    ));
```

Dica: A distribuição padrão do Symfony coloca a configuração de segurança em um arquivo separado (e.g. `app/config/security.yml`). Se você não tem um arquivo separado para as configurações de segurança, pode colocar diretamente no arquivo de configuração principal (por exemplo, `app/config/config.yml`).

O resultado final desta configuração é um completo sistema de segurança funcional com as seguintes características:

- Há dois usuários no sistema (`ryan` e `admin`);
- Os usuários se autenticam através da janela de autenticação básica HTTP;
- Qualquer URL que comece com `/admin/*` será protegida e somente o usuário `admin` terá acesso;
- Todas URLs que *não* comecem com `/admin/*` são acessíveis a todos usuários (e ao usuário nunca serão solicitadas as credenciais de acesso).

Vamos dar uma olhada como funciona a segurança e como cada parte da configuração influencia no sistema.

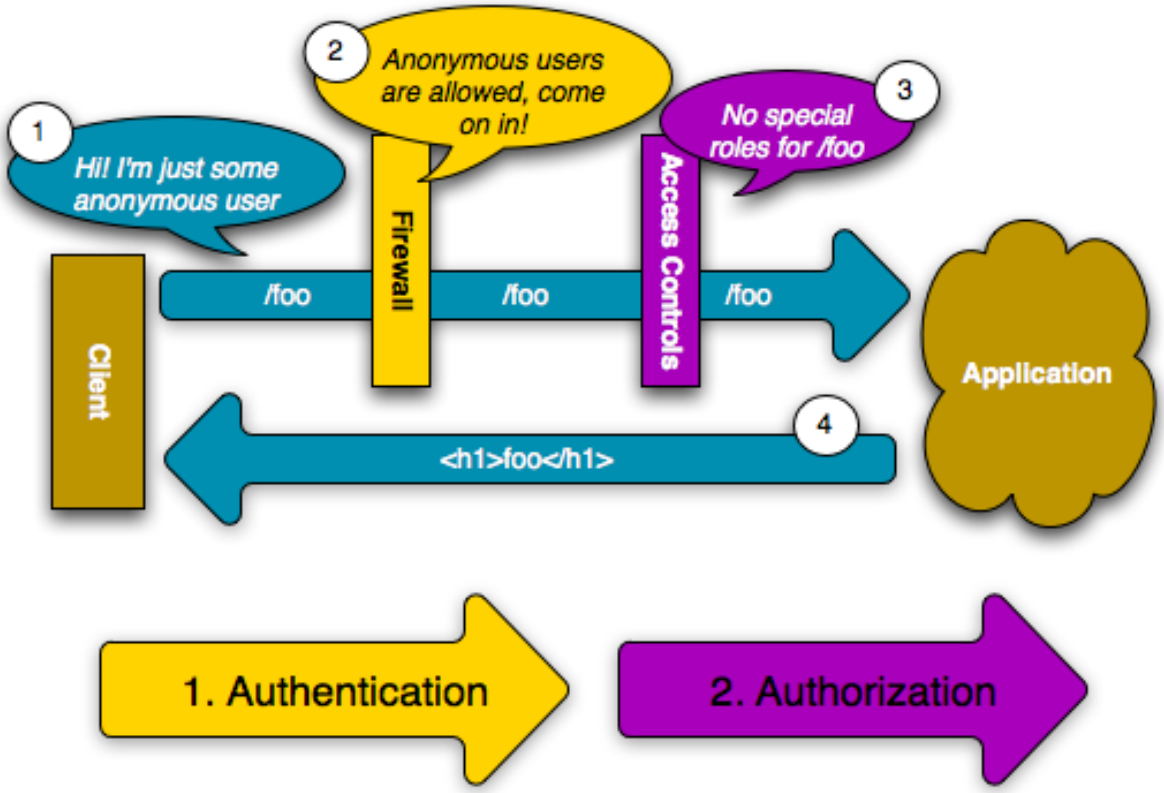
Como funciona a segurança: Autenticação e Autorização

O sistema de segurança do Symfony funciona determinando quem um usuário é (autenticação) e depois checando se o usuário tem acesso ao recurso específico ou URL solicitado.

Firewalls (Autenticação)

Quando um usuário requisita uma URL que está protegida por um firewall, o sistema de segurança é ativado. O trabalho do firewall é determinar se o usuário precisa ou não ser autenticado. Se ele precisar, envia a resposta de volta e inicia o processo de autenticação.

Um firewall será ativado quando a URL requisitada corresponda ao `padrão de caracteres` da expressão regular configurada na configuração de segurança. Neste exemplo, o `padrão de caracteres` (`^/`) corresponde a qualquer solicitação. O fato do firewall ser ativado *não* significa, porém, que a janela de autenticação básica HTTP (solicitando login e senha) será exibida para todas requisições. Por exemplo, qualquer usuário poderá acessar `/foo` sem que seja solicitada sua autenticação.

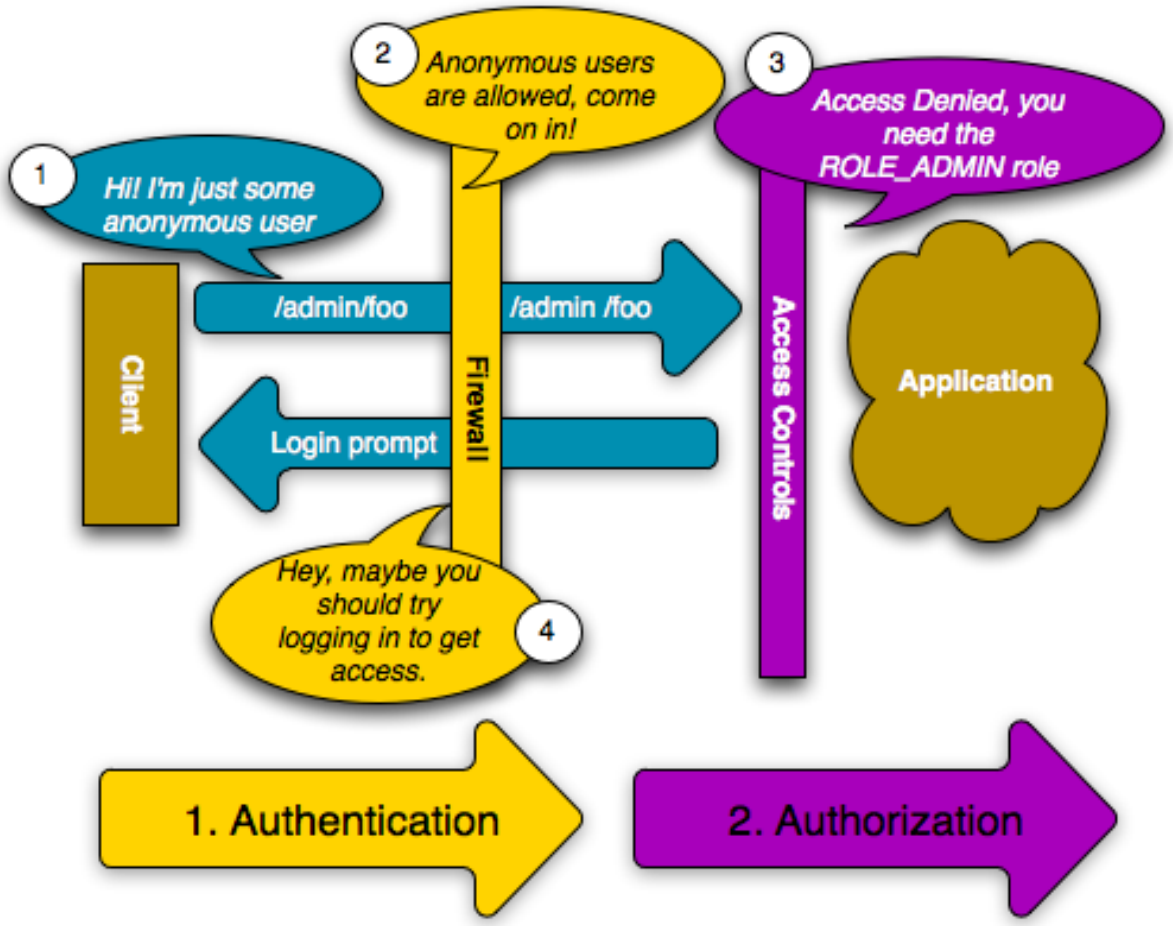


Isto funciona primeiramente por que o firewall permite *usuários anônimos* através do parâmetro `anonymous` da configuração. Em outras palavras, o firewall não exige que o usuário se autentique completamente. E por que nenhum `perfil` (`role`) é necessário para acessar `/foo` (na seção `access_control`), a solicitação pode ser realizada sem que o usuário sequer se identifique.

Se você remover a chave `anonymous`, o firewall *sempre* fará o usuário se identificar por completo imediatamente.

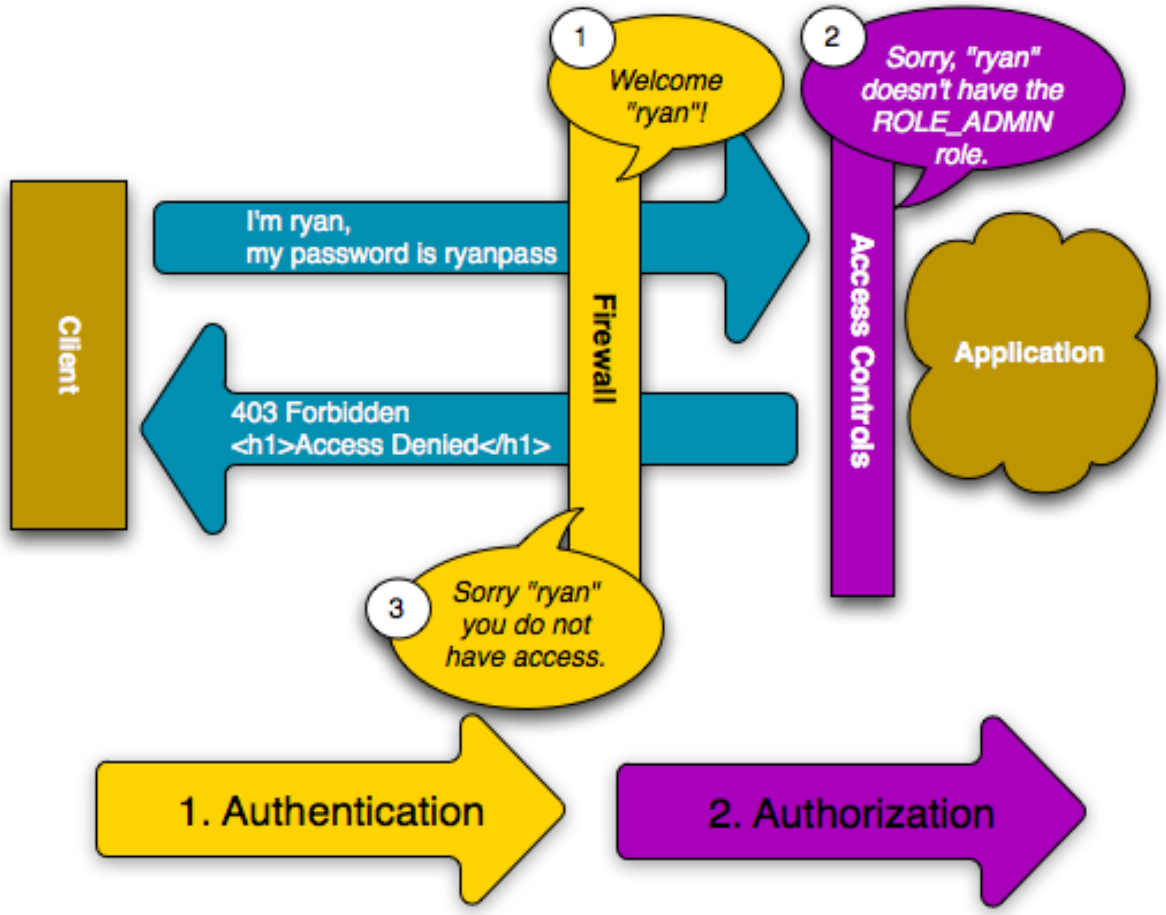
Controles de acesso (Autorização)

Se o usuário solicitar `/admin/foo`, porém, o processo toma um rumo diferente. Isto acontecerá por que a seção `access_control` da configuração indica que qualquer URL que se encaixe no padrão de caracteres `^/admin` (isto é, `/admin` ou qualquer coisa do tipo `/admin/*`) deve ser acessada somente por usuários com o perfil `ROLE_ADMIN`. Perfis são a base para a maioria das autorizações: o usuário pode acessar `/admin/foo` somente se tiver o perfil `ROLE_ADMIN`.



Como antes, o firewall não solicita credenciais de acesso. Assim que a camada de controle de acesso nega o acesso (por que o usuário não tem o perfil `ROLE_ADMIN`), porém, o firewall inicia o processo de autenticação. Este processo depende do mecanismo de autenticação que estiver utilizando. Por exemplo, se estiver utilizando o método de formulário de autenticação (`form login`), o usuário será redirecionado para a página de login. Se estiver utilizando o método básico de autenticação HTTP, o navegador recebe uma resposta do tipo HTTP 401 para que ao usuário seja exibida a janela de login/senha do navegador.

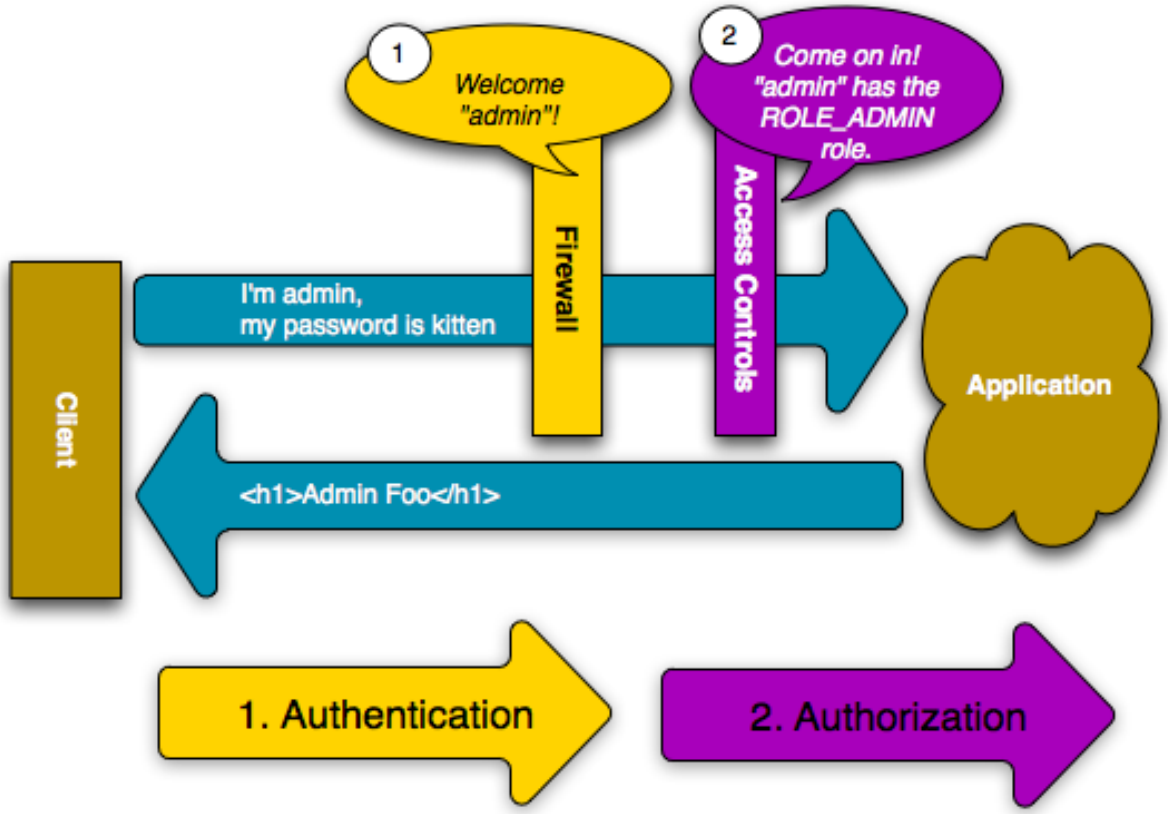
O usuário agora tem a oportunidade de digitar suas credenciais no aplicativo. Se as credenciais forem válidas, a requisição original será solicitada novamente.



No exemplo, o usuário `ryan` se autentica com sucesso pelo firewall. Como, porém, `ryan` não tem o perfil `ROLE_ADMIN`, ele ainda terá seu acesso negado ao recurso `/admin/foo`. Infelizmente, isto significa que o usuário verá uma mensagem indicando que o acesso foi negado.

Dica: Quando o Symfony nega acesso a um usuário, o usuário vê uma tela de erro e o navegador recebe uma resposta com o HTTP status code 403 (`Forbidden`). É possível personalizar a tela de erro de acesso negado seguindo as instruções em *Error Pages* do do texto do Symfony 2 - Passo-a-passo que ensina a personalizar a página de erro 403.

Finalmente, se o usuário `admin` requisitar `/admin/foo`, um processo similar entra em ação, mas neste caso, após a autenticação, a camada de controle de acesso permitirá que a requisição seja completada:



O fluxo de requisição quando um usuário solicita um recurso protegido é direto, mas muito flexível. Como verá mais tarde, a autenticação pode acontecer de diversas maneiras, incluindo formulário de login, certificado X.509, ou autenticação pelo Twitter. Independente do método de autenticação, o fluxo de requisiçao é sempre o mesmo:

1. Um usuário acessa um recurso protegido;
2. O aplicativo redireciona o usuário para o formulário de login;
3. O usuário envia suas credenciais (e.g. login/senha);
4. O firewall autentica o usuário;
5. O usuário autenticado é redirecionado para o recurso solicitado originalmente.

Nota: O processo *exato* na verdade depende um pouco do mecanismo de autenticação que estiver usando. Por exemplo, quando estiver utilizando formulário de login, o usuário envia suas credenciais para a URL que processa o formulário (por exemplo, `/login_check`) e depois é redirecionado de volta para a URL solicitada originalmente (por exemplo, `/admin/foo`). Se utilizar autenticação básica HTTP, porém, o usuário envia suas credenciais diretamente para a URL original (por exemplo, `/admin/foo`) e depois a página é retornada para o usuário na mesma requisição (isto significa que não há redirecionamentos).

Estes detalhes técnicos não devem ser relevantes no uso do sistema de segurança, mas é bom ter uma idéia a respeito.

Dica: Você aprenderá mais tarde como *qualquer coisa* pode ser protegida no Symfony2, incluindo controladores específicos, objetos, ou até métodos PHP.

Usando um formulário de login em HTML

Até agora, você viu como cobrir seu aplicativo depois do firewall e assim restringir o acesso de certas áreas a certos perfis. Utilizando a autenticação básica HTTP, é possível, sem esforços, submeter login/senha através da janela do navegador. O Symfony, porém, suporta de fábrica muitos outros mecanismos de autenticação. Para detalhes sobre todos eles, consulte `Referência Da Configuração De Segurança`.

Nesta seção, você aprimorará o processo permitindo que o usuário se autentique através de um formulário de login tradicional em HTML.

Primeiro habilite o formulário no seu firewall:

- *YAML*

```yaml
# app/config/security.yml
security:
    firewalls:
        secured_area:
            pattern:    ^/
            anonymous: ~
            form_login:
                login_path:  /login
                check_path:  /login_check
```

- *XML*

```xml
<!-- app/config/security.xml -->
<srv:container xmlns="http://symfony.com/schema/dic/security"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:srv="http://symfony.com/schema/dic/services"
    xsi:schemaLocation="http://symfony.com/schema/dic/services http://symfony.com/schema/dic/ser

    <config>
        <firewall name="secured_area" pattern="^/">
            <anonymous />
            <form-login login_path="/login" check_path="/login_check" />
        </firewall>
    </config>
</srv:container>
```

- *PHP*

```php
// app/config/security.php
$container->loadFromExtension('security', array(
    'firewalls' => array(
        'secured_area' => array(
            'pattern' => '^/',
            'anonymous' => array(),
            'form_login' => array(
                'login_path' => '/login',
                'check_path' => '/login_check',
            ),
        ),
    ),
));
```

Dica: Se não precisar de personlizar os valores de `login_path` ou `check_path` (os valores utilizados acima são os valores padrão), você pode encurtar seu configuração:

- *YAML*

```
form_login: ~
```

- *XML*

```
<form-login />
```

- *PHP*

```
'form_login' => array(),
```

Agora, quando o sistema de segurança inicia o processo de autenticação, ele redirecionará o usuário para o formulário de login (`/login` por padrão). É sua tarefa implementar o visual desse formulário. Primeiro, crie duas rotas: uma para a exibição do formulário de login (no caso, `/login`) e outra para processar a submissão do formulário (no caso, `/login_check`):

- *YAML*

```
# app/config/routing.yml
login:
    pattern:   /login
    defaults:  { _controller: AcmeSecurityBundle:Security:login }
login_check:
    pattern:   /login_check
```

- *XML*

```
<!-- app/config/routing.xml -->
<?xml version="1.0" encoding="UTF-8" ?>

<routes xmlns="http://symfony.com/schema/routing"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/routing http://symfony.com/schema/routing/rout

    <route id="login" pattern="/login">
        <default key="_controller">AcmeSecurityBundle:Security:login</default>
    </route>
    <route id="login_check" pattern="/login_check" />

</routes>
```

- *PHP*

```
// app/config/routing.php
use Symfony\Component\Routing\RouteCollection;
use Symfony\Component\Routing\Route;

$collection = new RouteCollection();
$collection->add('login', new Route('/login', array(
    '_controller' => 'AcmeDemoBundle:Security:login',
)));
$collection->add('login_check', new Route('/login_check', array()));

return $collection;
```

Nota: *Não* é preciso implementar o controller para a URL `/login_check` pois o firewall interceptará e processará o que foi submitido para essa URL. É opcional, porém útil, criar uma rota para que você possa gerar o link de submissão na template do formulário de login.

Novo na versão 2.1: Com o Symfony 2.1, você *deve* possuir rotas configuradas para suas URLs `login_path` (ex. `/login`), `check_path` (ex. `/login_check`) e `logout` (ex. `/logout` - veja **'Logging Out'_**).

Observe que o nome da rota `login` não é importante. O que importa é que a URL da rota corresponda o que foi colocado na configuração `login_path`, pois é para onde o sistema de segurança redirecionará os usuários que precisarem se autenticar.

O próximo passo é criar o controller que exibirá o formulário de login:

```
// src/Acme/SecurityBundle/Controller/Main;
namespace Acme\SecurityBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\Security\Core\SecurityContext;

class SecurityController extends Controller
{
    public function loginAction()
    {
        $request = $this->getRequest();
        $session = $request->getSession();

        // get the login error if there is one
        if ($request->attributes->has(SecurityContext::AUTHENTICATION_ERROR)) {
            $error = $request->attributes->get(SecurityContext::AUTHENTICATION_ERROR);
        } else {
            $error = $session->get(SecurityContext::AUTHENTICATION_ERROR);
        }

        return $this->render('AcmeSecurityBundle:Security:login.html.twig', array(
            // last username entered by the user
            'last_username' => $session->get(SecurityContext::LAST_USERNAME),
            'error'         => $error,
        ));
    }
}
```

Não se confunda com esse controller. Como verá, quando o usuário submete o formulário, o sistema de segurança automaticamente processar a submissão para você. Se o usuário entrou com login e/ou senha inválidos, este controller pega o erro ocorrido do sistema de segurança para poder exibir ao usuário.

Em outras palavras, seu trabalho é exibir o formulário de login e qualquer erro ocorrido durante a tentativa de autenticação, mas o sistema de segurança já toma conta de checar se as credenciais são válidas e de autenticar o usuário.

Finalmente crie a template correspondente:

- *Twig*

```
{# src/Acme/SecurityBundle/Resources/views/Security/login.html.twig #}
{% if error %}
    <div>{{ error.message }}</div>
{% endif %}

<form action="{{ path('login_check') }}" method="post">
    <label for="username">Username:</label>
    <input type="text" id="username" name="_username" value="{{ last_username }}" />

    <label for="password">Password:</label>
    <input type="password" id="password" name="_password" />
```

```
        {#
            If you want to control the URL the user is redirected to on success (more details below)
            <input type="hidden" name="_target_path" value="/account" />
        #}

        <input type="submit" name="login" />
    </form>
```

- *PHP*

```
    <?php // src/Acme/SecurityBundle/Resources/views/Security/login.html.php ?>
    <?php if ($error): ?>
        <div><?php echo $error->getMessage() ?></div>
    <?php endif; ?>

    <form action="<?php echo $view['router']->generate('login_check') ?>" method="post">
        <label for="username">Username:</label>
        <input type="text" id="username" name="_username" value="<?php echo $last_username ?>" />

        <label for="password">Password:</label>
        <input type="password" id="password" name="_password" />

        <!--
            If you want to control the URL the user is redirected to on success (more details below)
            <input type="hidden" name="_target_path" value="/account" />
        -->

        <input type="submit" name="login" />
    </form>
```

Dica: A variável `error` passada para a template é uma instância de `AuthenticationException`. Esta pode conter mais informações - ou até informações sensíveis - sobre a falha na autenticação, por isso use-a com sabedoria!

O formulário tem que atender alguns requisitos. Primeiro, ao submeter o formulário para `/login_check` (através da rota `login_check`), o sistema de segurança interceptará a submissão do formulário e o processará. Segundo, o sistema de segurança espera que os campos submetidos sejam chamados `_username` e `_password` (estes nomes podem ser configured).

E é isso! Quando submeter um formulário, o sistema de segurança irá automaticamente checar as credenciais do usuário e autenticá-lo ou enviar o ele de volta ao formulário de login para o erro ser exibido.

Vamos revisar o processo inteiro:

1. O usuário tenta acessar um recurso que está protegido;
2. O firewall inicia o processo de autenticação redirecionando o usuário para o formulário de login(`/login`);
3. A página `/login` produz o formulário de login através da rota e controlador criados neste exemplo;
4. O usuário submete o formulário de login para `/login_check`;
5. O sistema de segurança intercepta a solicitação, verifica as credenciais submetidas pelo usuário, autentica o mesmo se tiverem corretas ou envia de volta para o formulário de login caso contrário;

Por padrão, se as credenciais estiverem corretas, o usuário será redirecionado para a página que solicitou originalmente (e.g. `/admin/foo`). Se o usuário originalmente solicitar a página de login, ele será redirecionado para a página principal. Isto pode ser modificado se necessário, o que permitiria você redirecionar o usuário para um outra URL específica.

Para maiores detalhes sobre isso e como personalizar o processamento do formulário de login acesse Como personalizar o seu Formulário de Login.

Evite os erros comuns

Quando estiver configurando seu formulário de login, fique atendo aos seguintes erros comuns.

1. Crie as rotas corretas

Primeiro, tenha certeza que definiu as rotas `/login` e `/login_check` corretamente e que elas correspondem aos calores das configurações `login_path` e `check_path`. A configuração errada pode significar que você será redirecionado para a página de erro 404 ao invés da página de login ou a submissão do formulário de login não faça nada (você sempre vê o formulário sem sair dele).

2. Tenha certeza que a página de login não é protegida

Também tenha certeza que a página de login *não* precisa de qualquer perfil para ser vizualizada. Por exemplo, a seguinte configuração, que exige o perfil `ROLE_ADMIN` para todas as URLs (incluindo a URL `/login`), causará um redirecionamento circular:

- *YAML*

```
access_control:
    - { path: ^/, roles: ROLE_ADMIN }
```

- *XML*

```
<access-control>
    <rule path="^/" role="ROLE_ADMIN" />
</access-control>
```

- *PHP*

```
'access_control' => array(
    array('path' => '^/', 'role' => 'ROLE_ADMIN'),
),
```

Removendo o controle de acesso para a URL `/login` resolve o problema:

- *YAML*

```
access_control:
    - { path: ^/login, roles: IS_AUTHENTICATED_ANONYMOUSLY }
    - { path: ^/, roles: ROLE_ADMIN }
```

- *XML*

```
<access-control>
    <rule path="^/login" role="IS_AUTHENTICATED_ANONYMOUSLY" />
    <rule path="^/" role="ROLE_ADMIN" />
</access-control>
```

- *PHP*

```
'access_control' => array(
    array('path' => '^/login', 'role' => 'IS_AUTHENTICATED_ANONYMOUSLY'),
    array('path' => '^/', 'role' => 'ROLE_ADMIN'),
),
```

Além disso, se o seu firewall *não* permite usuários anônimos, você precisará criar um firewall especial para permitir usuários anônimos para a página de login:

- *YAML*

```
firewalls:
    login_firewall:
        pattern:    ^/login$
        anonymous:  ~
    secured_area:
        pattern:    ^/
        form_login: ~
```

- *XML*

```
<firewall name="login_firewall" pattern="^/login$">
    <anonymous />
</firewall>
<firewall name="secured_area" pattern="^/">
    <form_login />
</firewall>
```

Autorização

O primeiro passo na segurança é sempre a autenticação: o processo de verficar quem o usuário é. No Symfony, a autenticação pode ser feita de várias maneiras - via formulário de login, autenticação básica HTTP ou até mesmo pelo Facebook.

Uma vez que o usuário está autenticado, a autorização começa. Autorização fornece uma maneira padrão e poderosa de decidir se o usuário pode acessar algum recurso (uma URL, um objeto do modelo, um método...). Isto funciona com perfis atribuídos para cada usuário e exigindo perfis diferentes para diferentes recursos.

O processo de autorização tem dois lados diferentes:

1. O usuário tem um conjunto de perfis específico;
2. Um recurso requer um perfil específico para ser acessado.

Nesta seção, o foco será em como tornar seguros diferentes recursos (por exemplo URLs, chamadas a métodos, etc) com diferentes perfis. Mais tarde, você aprenderá mais como perfis são criados e atribuídos aos usuários.

Protegendo padrões de URLs

A maneira mais básica de proteger seu aplicativo é proteger um padrão de URL. Você já viu no primeiro exemplo deste capítulo que qualquer requisição que se encaixasse na expressão regular `^/admin` exigiria o perfil `ROLE_ADMIN`.

Você pode definir quantos padrões precisar. Cada um é uma expressão regular.

- *YAML*

```yaml
# app/config/config.yml
security:
    # ...
    access_control:
        - { path: ^/admin/users, roles: ROLE_SUPER_ADMIN }
        - { path: ^/admin, roles: ROLE_ADMIN }
```

- *XML*

```xml
<!-- app/config/config.xml -->
<config>
    <!-- ... -->
    <rule path="^/admin/users" role="ROLE_SUPER_ADMIN" />
    <rule path="^/admin" role="ROLE_ADMIN" />
</config>
```

- *PHP*

```php
// app/config/config.php
$container->loadFromExtension('security', array(
    // ...
    'access_control' => array(
        array('path' => '^/admin/users', 'role' => 'ROLE_SUPER_ADMIN'),
        array('path' => '^/admin', 'role' => 'ROLE_ADMIN'),
    ),
));
```

Dica: Iniciando o padrão com ^ garante que somente URLs *começando* com o padrão terá uma comparação positiva. Por exemplo, o padrão simples `/admin` (sem o ^) resultaria em uma comparação positiva para `/admin/foo`, mas também para URLs como `/foo/admin`.

Para cada requisição que chega, o Symfony2 tenta encontrar uma regra de acesso correspondente, com comparação positiva do padrão (a primeira que encontrar ganha). Se o usuário não estiver autenticado ainda, a autenticação é iniciada (isto é, o usuário tem a chance de fazer login). Se o usuário, porém, já *estiver* autenticado, mas não tiver o perfil exigido, uma exceção é disparada `AccessDeniedException` , que você pode tratar e transformar em uma apresentável página de "Acesso Negado" para o usuário. Veja Como personalizar as páginas de erro para mais informações.

Como o Symfony utiliza a primeira regra de acesso que der uma comparação positiva, uma URL como `/admin/users/new` corresponderá a primeira regra e exigirá somente o perfil `ROLE_SUPER_ADMIN`. Qualquer URL como `/admin/blog` corresponderá a segunda regra e exigirá o perfil `ROLE_ADMIN`.

Protegendo por IP

Algumas situações podem exigir que você restrinja o acesso de uma determinada rota com base no IP. Isto é particularmente relevante no caso de *Edge Side Includes* (ESI), por exemplo, que utiliza a rota com nome "_internal". Quanto ESI é utilizado, a rota _internal é requerida pelo gateway cache (gerente de cache) para possibilitar diferentes opções de caching para subseções dentro de uma determinada página. Esta rota vem com o prefixo ^/_internal por padrão na edição padrão (assumindo que você ativou estas linhas do seu arquivo de configuração de rotas - routing.yml).

Aqui está um exemplo de como poderia proteger esta rota de acesso externo:

- *YAML*

```
# app/config/security.yml
security:
    # ...
    access_control:
        - { path: ^/_internal, roles: IS_AUTHENTICATED_ANONYMOUSLY, ip: 127.0.0.1 }
```

- *XML*

```
<access-control>
    <rule path="^/_internal" role="IS_AUTHENTICATED_ANONYMOUSLY" ip="127.0.0.1" />
</access-control>
```

- *PHP*

```
'access_control' => array(
    array('path' => '^/_internal', 'role' => 'IS_AUTHENTICATED_ANONYMOUSLY', 'ip' => '127.0.0.1'
),
```

Protegendo por canal

Assim como a proteção por IP, exigir o uso de SSL é tão simples quanto adicionar uma nova entrar em access_control:

- *YAML*

```
# app/config/security.yml
security:
    # ...
    access_control:
        - { path: ^/cart/checkout, roles: IS_AUTHENTICATED_ANONYMOUSLY, requires_channel: https
```

- *XML*

```
<access-control>
    <rule path="^/cart/checkout" role="IS_AUTHENTICATED_ANONYMOUSLY" requires_channel="https" />
</access-control>
```

- *PHP*

```
    'access_control' => array(
        array('path' => '^/cart/checkout', 'role' => 'IS_AUTHENTICATED_ANONYMOUSLY', 'requires_chann
    ),
```

Protegendo um Controller

Proteger seu aplicativo baseado em padrões de URL é fácil, mas este método pode não ser específico o bastante em certos casos. Quando necessário, você pode ainda facilmente forçar autorização de dentro de um controller:

```
use Symfony\Component\Security\Core\Exception\AccessDeniedException;
// ...

public function helloAction($name)
{
    if (false === $this->get('security.context')->isGranted('ROLE_ADMIN')) {
        throw new AccessDeniedException();
    }

    // ...
}
```

Você pode ainda instalar e utilizar opcionalmente o `JMSSecurityExtraBundle`, que te permite proteger controllers através de anotações:

```
use JMS\SecurityExtraBundle\Annotation\Secure;

/**
 * @Secure(roles="ROLE_ADMIN")
 */
public function helloAction($name)
{
    // ...
}
```

Para mais informações, veja a documentação JMSSecurityExtraBundle . Se você a distribuição Standard do Symfony, este bundle está habilitado por padrão. Se não estiver, você pode facilmente baixar e instalá-lo.

Protegendo outros serviços

De fato, qualquer coisa pode ser protegida em Symfony utilizando uma estratégia similar a apresentada na seção anterior. Por exemplo, suponha que você tem um serviço (uma classe PHP, por exemplo) que seu trabalho é enviar e-mails de um usuário para outro. Você pode restringir o uso dessa classe - não importa de onde está sendo utilizada - a usuários que tenham um perfil específico.

Para mais informações sobre como você pode utilizar o componente de segurança para proteger diferentes serviços e métodos de seu aplicativo, consulte `/cookbook/security/securing_services`.

Listas De Controle De Acesso (ACLs): Protegendo Objetos Específicos Do Banco De Dados

Imagine que você está projetando um sistema de blog onde seus usuários podem comentar seus posts. Agora, você quer que um usuário tenha a possibilidade de editar seus próprios comentários, mas não aqueles de outros usuários. Além disso, como administrador, você quer poder editar *todos* os comentários.

O componente de segurança possui um sistema de listas de controle de acesso (ACL) que te permite controlar acesso a instâncias individuais de um objeto no seu sistema. *Sem* ACL, você consegue proteger seu sistema para que somente usuários específicos possam editar os comentários. *Com* ACL, porém, você pode restringir ou permitir o acesso por comentário.

Para mais informação, veja o passo-a-passo: Listas de controle de acesso (ACLs).

Usuários

Nas seções anteriores, você aprendeu como proteger diferentes recursos exigindo um conjunto de *perfis* para o acesso a um recurso. Nesta seção exploraremos outro aspecto da autorização: os usuários.

De onde os usuários vêm? (*User Providers*)

Durante a autenticação, o usuário submete um conjunto de credenciais (normalmente login e senha). O trabalho do sistema de autenticação é verificar essas credenciais contra um conjunto de usuários. De onde essa lista de usuários vem então?

No Symfony2, usuários podem vir de qualquer lugar - um arquivo de configuração, um banco de dados, um serviço web ou qualquer outra fonte que desejar. Qualquer coisa que disponibiliza um ou mais usuários para o sistema de autenticação é conhecido como "user provider". O Symfony2 vem por padrão com os dois mais comuns: um que carrega os usuários do arquivo de configuração e outro que carrega os usuários do banco de dados.

Especificando usuários no arquivo de configuração O jeito mais fácil de especificar usuários é diretamnete no arquivo de configuração. De fato, você já viu isso em um exemplo neste capítulo.

- *YAML*

```yaml
# app/config/config.yml
security:
    # ...
    providers:
        default_provider:
            memory:
                users:
                    ryan:  { password: ryanpass, roles: 'ROLE_USER' }
                    admin: { password: kitten, roles: 'ROLE_ADMIN' }
```

- *XML*

```xml
<!-- app/config/config.xml -->
<config>
    <!-- ... -->
    <provider name="default_provider">
        <memory>
            <user name="ryan" password="ryanpass" roles="ROLE_USER" />
            <user name="admin" password="kitten" roles="ROLE_ADMIN" />
        </memory>
    </provider>
</config>
```

- *PHP*

```php
// app/config/config.php
$container->loadFromExtension('security', array(
    // ...
```

```
        'providers' => array(
            'default_provider' => array(
                'memory' => array(
                    'users' => array(
                        'ryan' => array('password' => 'ryanpass', 'roles' => 'ROLE_USER'),
                        'admin' => array('password' => 'kitten', 'roles' => 'ROLE_ADMIN'),
                    ),
                ),
            ),
        ),
    ));
```

Este *user provider* é chamado de "in-memory" user provider, já que os usuários não estão armazenados em nenhum banco de dados. O objeto usuário é fornecido pelo Symfony (`User`).

Dica: Qualquer user provider pode carregar usuários diretamente da configuração se especificar o parâmetro de configuração `users` e listar os usuários abaixo dele.

Cuidado: Se seu login é todo numérico (`77`, por exemplo) ou contém hífen (`user-name`, por exemplo), você deveria utilizar a sintaxe alternativa quando especificar usuários em YAML:

```
users:
    - { name: 77, password: pass, roles: 'ROLE_USER' }
    - { name: user-name, password: pass, roles: 'ROLE_USER' }
```

Para sites menores, este método é rápido e fácil de configurar. Para sistemas mais complexos, você provavelmente desejará carregar os usuários do banco de dados.

Carregando usuários do banco de dados Se você desejar carregar seus usuários através do Doctrine ORM, você pode facilmente o fazer criando uma classe `User` e configurando o `entity` provider

Nessa abordagem, você primeiro precisa criar sua própria classe `User`, que será persistida no banco de dados.

```
// src/Acme/UserBundle/Entity/User.php
namespace Acme\UserBundle\Entity;

use Symfony\Component\Security\Core\User\UserInterface;
use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity
 */
class User implements UserInterface
{
    /**
     * @ORM\Column(type="string", length="255")
     */
    protected $username;

    // ...
}
```

Ao que diz respeito ao sistema de segurança, o único requisito para sua classe `User` personalizada é que ela implemente a interface `UserInterface` . Isto significa que conceito de usuário pode ser qualquer um, desde que implemente essa interface.

Novo na versão 2.1: No Symfony 2.1, o método `equals` foi removido do `UserInterface`. Se você precisa sobrescrever a implementação default da lógica de comparação, implemente a nova interface `EquatableInterface`.

Nota: O objeto User será serializado e salvo na sessão entre requisições, por isso é recomendado que você implemente a interface Serializable em sua classe User. Isto é especialmente importante se sua classe `User` tem uma classe pai com propriedades private.

Em seguida, configure um user provider `entity` e aponte-o para sua classe `User`:

- *YAML*

```
# app/config/security.yml
security:
    providers:
        main:
            entity: { class: Acme\UserBundle\Entity\User, property: username }
```

- *XML*

```
<!-- app/config/security.xml -->
<config>
    <provider name="main">
        <entity class="Acme\UserBundle\Entity\User" property="username" />
    </provider>
</config>
```

- *PHP*

```
// app/config/security.php
$container->loadFromExtension('security', array(
    'providers' => array(
        'main' => array(
            'entity' => array('class' => 'Acme\UserBundle\Entity\User', 'property' => 'username'
        ),
    ),
));
```

Com a introdução desse novo provider, o sistema de autenticação tentará carregar o objeto `User` do banco de dados a partir do campo `username` da classe.

Nota: Este exemplo é somente para demonstrar a idéia básica por trás do provider `entity`. Para um exemplo completo, consulte `/cookbook/security/entity_provider`.

Para mais informações sobre como criar seu próprio provider (se precisar carregar usuários do seu serviço web por exemplo), consulte Como criar um Provider de Usuário Personalizado.

Protegendo a senha do usuário

Até agora, por simplicidade, todos os exemplos armazenavam as senhas dos usuários em texto puro (sendo armazenados no arquivo de configuração ou no banco de dados). Claro que em um aplicativo profissional você desejará proteger as senhas dos seus usuários por questões de segurança. Isto é facilmente conseguido mapeando sua classe User para algum "encoder" disponível. Por exemplo, para armazenar seus usuário em memória, mas proteger a senha deles através da função de hash *sha1*', faça o seguinte:

- *YAML*

```yaml
# app/config/config.yml
security:
    # ...
    providers:
        in_memory:
            memory:
                users:
                    ryan:  { password: bb87a29949f3a1ee0559f8a57357487151281386, roles: 'ROLE_US
                    admin: { password: 74913f5cd5f61ec0bcfdb775414c2fb3d161b620, roles: 'ROLE_AD

    encoders:
        Symfony\Component\Security\Core\User\User:
            algorithm:   sha1
            iterations: 1
            encode_as_base64: false
```

- *XML*

```xml
<!-- app/config/config.xml -->
<config>
    <!-- ... -->
    <provider name="in_memory">
        <memory>
            <user name="ryan" password="bb87a29949f3a1ee0559f8a57357487151281386" roles="ROLE_US
            <user name="admin" password="74913f5cd5f61ec0bcfdb775414c2fb3d161b620" roles="ROLE_A
        </memory>
    </provider>

    <encoder class="Symfony\Component\Security\Core\User\User" algorithm="sha1" iterations="1" e
</config>
```

- *PHP*

```php
// app/config/config.php
$container->loadFromExtension('security', array(
    // ...
    'providers' => array(
        'in_memory' => array(
            'memory' => array(
                'users' => array(
                    'ryan' => array('password' => 'bb87a29949f3a1ee0559f8a57357487151281386', 'r
                    'admin' => array('password' => '74913f5cd5f61ec0bcfdb775414c2fb3d161b620', '
                ),
            ),
        ),
    ),
    'encoders' => array(
        'Symfony\Component\Security\Core\User\User' => array(
            'algorithm'         => 'sha1',
            'iterations'        => 1,
            'encode_as_base64'  => false,
        ),
    ),
));
```

Ao definir `iterations` como `1` e `encode_as_base64` como false, a senha codificada é simplesmente obtida como o resultado de `sha1` após uma iteração apenas, sem codificação extra. Você pode agora calcular a senha codificada por código PHP (e.g. `hash('sha1', 'ryanpass')`) ou através de alguma ferramenta online como

functions-online.com .

Se você estiver criando seus usuário dinamicamente e os armazenando no banco de dados, você pode usar algoritmos the hash ainda mais complexos e então delegar em um objeto encoder para ajudar a codificar as senhas. Por exemplo, suponha que seu objeto User é `Acme\UserBundle\Entity\User` (como no exemplo acima). Primeiro, configure o encoder para aquele usuário:

- *YAML*

```yaml
# app/config/config.yml
security:
    # ...

    encoders:
        Acme\UserBundle\Entity\User: sha512
```

- *XML*

```xml
<!-- app/config/config.xml -->
<config>
    <!-- ... -->

    <encoder class="Acme\UserBundle\Entity\User" algorithm="sha512" />
</config>
```

- *PHP*

```php
// app/config/config.php
$container->loadFromExtension('security', array(
    // ...

    'encoders' => array(
        'Acme\UserBundle\Entity\User' => 'sha512',
    ),
));
```

Neste caso, você está utilizando um algoritmo mais forte `sha512`. Além disso, desde que você especificou o algoritmo (`sha512`) como um texto, o sistema irá, por padrão, utilizar a função de hash 5000 vezes em uma linha e então o codificar como base64. Em outras palavras, a senha foi muito codificada de maneira que a senha não pode ser decodificada (isto é, você não pode determinar qual a senha a partir da senha codificada).

Se você tem alguma espécie de formulário de registro para os visitantes, você precisará a senha codificada para poder armazenar. Não importa o algoritmo que configurar para sua classe User, a senha codificada pode sempre ser determinada da seguinte maneira a partir de um controller:

```php
$factory = $this->get('security.encoder_factory');
$user = new Acme\UserBundle\Entity\User();

$encoder = $factory->getEncoder($user);
$password = $encoder->encodePassword('ryanpass', $user->getSalt());
$user->setPassword($password);
```

Obtendo o objeto User

Após a autenticação, o objeto `User` do usuário atual pode ser acessado através do serviço `security.context`. De dentro de um controller, faça o seguinte:

```
public function indexAction()
{
    $user = $this->get('security.context')->getToken()->getUser();
}
```

No controller, também existe o atalho:

```
public function indexAction()
{
    $user = $this->getUser();
}
```

Nota: Usuários anônimos são tecnicamente autenticados, significando que o médodo `isAuthenticated()` de um objeto User autenticado anonimamente retornará verdadeiro. Para verificar se seu usuário está realmente autenticado, verifique se o perfil `IS_AUTHENTICATED_FULLY` está atribuído ao mesmo.

Utilizando múltiplos User Providers

Cada mecanismo de autenticação (exemplos: Autenticação HTTP, formulário de login, etc) usa exatamente um user provider, e utilizará, por padrão, o primeiro user provider configurado. O que acontece se você quiser que alguns de seus usuários sejam autenticados por arquivo de configuração e o resto por banco de dados? Isto é possível criando um novo user provider que ativa os dois juntos:

- *YAML*

```
# app/config/security.yml
security:
    providers:
        chain_provider:
            chain:
                providers: [in_memory, user_db]
        in_memory:
            users:
                foo: { password: test }
        user_db:
            entity: { class: Acme\UserBundle\Entity\User, property: username }
```

- *XML*

```
<!-- app/config/config.xml -->
<config>
    <provider name="chain_provider">
        <chain>
            <provider>in_memory</provider>
            <provider>user_db</provider>
        </chain>
    </provider>
    <provider name="in_memory">
        <user name="foo" password="test" />
    </provider>
    <provider name="user_db">
        <entity class="Acme\UserBundle\Entity\User" property="username" />
    </provider>
</config>
```

- *PHP*

```
// app/config/config.php
$container->loadFromExtension('security', array(
    'providers' => array(
        'chain_provider' => array(
            'chain' => array(
                'providers' => array('in_memory', 'user_db'),
            ),
        ),
        'in_memory' => array(
            'users' => array(
                'foo' => array('password' => 'test'),
            ),
        ),
        'user_db' => array(
            'entity' => array('class' => 'Acme\UserBundle\Entity\User', 'property' => 'username'
        ),
    ),
));
```

Agora, todos mecanismos de autenticação utilizarão o `chain_provider`, já que é o primeiro configurado. O `chain_provider` tentará carregar o usuário de ambos providers `in_memory` e `user_db`.

Dica: Se você não tem razões para separar seus usuários `in_memory` dos seus usuários `user_db`, você pode conseguir o mesmo resultado mais facilmente, combinando as duas origens em um único provider:

- *YAML*

```
# app/config/security.yml
security:
    providers:
        main_provider:
            memory:
                users:
                    foo: { password: test }
            entity:
                class: Acme\UserBundle\Entity\User,
                property: username
```

- *XML*

```
<!-- app/config/config.xml -->
<config>
    <provider name=="main_provider">
        <memory>
            <user name="foo" password="test" />
        </memory>
        <entity class="Acme\UserBundle\Entity\User" property="username" />
    </provider>
</config>
```

- *PHP*

```
// app/config/config.php
$container->loadFromExtension('security', array(
    'providers' => array(
        'main_provider' => array(
            'memory' => array(
                'users' => array(
                    'foo' => array('password' => 'test'),
```

```
                    ),
                ),
                'entity' => array('class' => 'Acme\UserBundle\Entity\User', 'property' => 'username'
            ),
        ),
    ));
```

Você pode ainda configurar o firewall ou mecanismos de autenticação individuais para utilizar um user provider específico. Novamente, a menos que um provider seja especificado explicitamente, o primeiro será sempre utilizado:

- *YAML*

```
# app/config/config.yml
security:
    firewalls:
        secured_area:
            # ...
            provider: user_db
            http_basic:
                realm: "Secured Demo Area"
                provider: in_memory
            form_login: ~
```

- *XML*

```
<!-- app/config/config.xml -->
<config>
    <firewall name="secured_area" pattern="^/" provider="user_db">
        <!-- ... -->
        <http-basic realm="Secured Demo Area" provider="in_memory" />
        <form-login />
    </firewall>
</config>
```

- *PHP*

```
// app/config/config.php
$container->loadFromExtension('security', array(
    'firewalls' => array(
        'secured_area' => array(
            // ...
            'provider' => 'user_db',
            'http_basic' => array(
                // ...
                'provider' => 'in_memory',
            ),
            'form_login' => array(),
        ),
    ),
));
```

Neste exemplo, se um usuário tentar se autenticar através de autenticação HTTP, o sistema utilizará o user provider `in_memory`. Se o usuário tentar, porém, se autenticar através do formulário de login, o provider `user_db` será usado (pois é o padrão para todo o firewall).

Para mais informações sobre a configuração do user provider e do firewall, veja `/reference/configuration/security`.

Perfis (Roles)

A idéia de um "perfil" é chave no processo de autorização. Para cada usuário é atribuído um conjunto de perfis e então cada recurso exige um ou mais perfis. Se um usuário tem os perfis requeridos, o acesso é concedido. Caso contrário, o acesso é negado.

Perfis são muito simples e basicamente textos que você pode inventar e utilizar de acordo com suas necessidades (embora perfis sejam objetos PHP internamente). Por exemplo, se precisar limitar acesso a uma seção administrativa do blog de seu website, você pode proteger a seção utilizando o perfil `ROLE_BLOG_ADMIN`. Este perfil não precisa de estar definido em lugar nenhum - você pode simplesmente usar o mesmo.

Nota: Todos os perfis **devem** começar com o prefixo `ROLE_` para serem gerenciados pelo Symfony2. Se você definir seus próprios perfis com uma classe `Role` dedicada (mais avançado), não utilize o prefixo `ROLE_`.

Hierarquia de Perfis

Ao invés de associar muitos perfis aos usuários, você pode defined regras de herança ao criar uma hierarquia de perfis:

- *YAML*

```yaml
# app/config/security.yml
security:
    role_hierarchy:
        ROLE_ADMIN:       ROLE_USER
        ROLE_SUPER_ADMIN: [ROLE_ADMIN, ROLE_ALLOWED_TO_SWITCH]
```

- *XML*

```xml
<!-- app/config/security.xml -->
<config>
    <role id="ROLE_ADMIN">ROLE_USER</role>
    <role id="ROLE_SUPER_ADMIN">ROLE_ADMIN, ROLE_ALLOWED_TO_SWITCH</role>
</config>
```

- *PHP*

```php
// app/config/security.php
$container->loadFromExtension('security', array(
    'role_hierarchy' => array(
        'ROLE_ADMIN'       => 'ROLE_USER',
        'ROLE_SUPER_ADMIN' => array('ROLE_ADMIN', 'ROLE_ALLOWED_TO_SWITCH'),
    ),
));
```

Na configuração acima, usuários com o perfil `ROLE_ADMIN` terão também o perfil `ROLE_USER`. O perfil `ROLE_SUPER_ADMIN` tem os `ROLE_ADMIN`, `ROLE_ALLOWED_TO_SWITCH` e `ROLE_USER` (herdado do `ROLE_ADMIN`).

Saindo do sistema

Normalmente, você também quer que seus usuários possam sair do sistema. Felizmente, o firewall consegue lidar com isso automaticamente quando o parâmetro de configuração `logout` está ativo:

- *YAML*

```yaml
# app/config/config.yml
security:
    firewalls:
        secured_area:
            # ...
            logout:
                path:   /logout
                target: /
    # ...
```

- *XML*

```xml
<!-- app/config/config.xml -->
<config>
    <firewall name="secured_area" pattern="^/">
        <!-- ... -->
        <logout path="/logout" target="/" />
    </firewall>
    <!-- ... -->
</config>
```

- *PHP*

```php
// app/config/config.php
$container->loadFromExtension('security', array(
    'firewalls' => array(
        'secured_area' => array(
            // ...
            'logout' => array('path' => 'logout', 'target' => '/'),
        ),
    ),
    // ...
));
```

Uma vez que está configurado no seu firewall, redirecionando o usuário para `/logout` (ou qualquer outro caminho que configurar em `path`), o usuário não estará mais autenticado. O usuário será então redirecionado para a página principal (o valor definido no parâmetro `target`). Ambas configurações `path` e `target` tem valor padrão iguais ao especificado aqui. Em outras palavras, a menos que precise personalizar, você pode simplesmente os omitir completamente e simplificar sua configuração:

- *YAML*

```yaml
logout: ~
```

- *XML*

```xml
<logout />
```

- *PHP*

```php
'logout' => array(),
```

Note que você *não* precisará implementar o controller para a URL `/logout` já que o firewall cuida disso. Você *deve*, entretanto, precisar criar uma rota para que possa usar para gerar a URL:

Aviso: Com o Symfony 2.1, você *deve* ter uma rota que corresponde ao seu caminho para logout. Sem esta rota, o logout não irá funcionar.

- *YAML*

```yaml
# app/config/routing.yml
logout:
    pattern:   /logout
```

- *XML*

```xml
<!-- app/config/routing.xml -->
<?xml version="1.0" encoding="UTF-8" ?>

<routes xmlns="http://symfony.com/schema/routing"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/routing http://symfony.com/schema/routing/rout

    <route id="logout" pattern="/logout" />

</routes>
```

- *PHP*

```php
// app/config/routing.php
use Symfony\Component\Routing\RouteCollection;
use Symfony\Component\Routing\Route;

$collection = new RouteCollection();
$collection->add('logout', new Route('/logout', array()));

return $collection;
```

Uma vez que o usuário não está mais autenticado, ele será redirecionado para o que tiver definido no parâmetro `target`. Para mais informações sobre a configuração de logout, veja `Security Configuration Reference`.

Controle de Acesso em Templates

Se você quiser checar se o usuário atual tem um determinado perfil de dentro de uma template, use a função:

- *Twig*

```twig
{% if is_granted('ROLE_ADMIN') %}
    <a href="...">Delete</a>
{% endif %}
```

- *PHP*

```php
<?php if ($view['security']->isGranted('ROLE_ADMIN')): ?>
    <a href="...">Delete</a>
<?php endif; ?>
```

Nota: Se você usar esta função e *não* estiver em uma URL que está atrás de um firewall ativo, uma exceção será gerada. Novamente, quase sempre é uma boa idéia ter um firewall principal que protege todas as URLs (como visto neste capítulo).

Controle de Acesso em Controllers

Se você quer verificar se o usuário atual tem um perfil de dentro de um controller, use o método `isGranted` do contexto de segurança:

```
public function indexAction()
{
    // show different content to admin users
    if ($this->get('security.context')->isGranted('ADMIN')) {
        // Load admin content here
    }
    // load other regular content here
}
```

Nota: Um firewall deve estar ativo ou uma exceção será gerada quanto o método `isGranted` for chamado. Veja a nota acima sobre templates para mais detalhes.

Passando por outro usuário

As vezes, é útil poder trocar de um usuário para outro sem ter que sair e se autenticar novamente (por exemplo quando você está depurando or tentando entender uma falha que um usuário vê e você não consegue reproduzir). Isto pode ser feito ativando o listener `switch_user` do firewall:

- *YAML*

```
# app/config/security.yml
security:
    firewalls:
        main:
            # ...
            switch_user: true
```

- *XML*

```
<!-- app/config/security.xml -->
<config>
    <firewall>
        <!-- ... -->
        <switch-user />
    </firewall>
</config>
```

- *PHP*

```
// app/config/security.php
$container->loadFromExtension('security', array(
    'firewalls' => array(
        'main'=> array(
            // ...
            'switch_user' => true
        ),
    ),
));
```

Para mudar para outro usuário, basta adicionar o parâmetro de URL `_switch_user` indicando o usuário (username) na URL atual:

http://example.com/somewhere?_switch_user=thomas

Para voltar ao usuário original, use como nome de usuário o texto `_exit`:

http://example.com/somewhere?_switch_user=_exit

Claro que esta funcionalidade precisar estar disponível para um grupo reduzido de usuários. Por padrão, o acesso é restrito a usuários que tem o perfil `ROLE_ALLOWED_TO_SWITCH`. O nome deste perfil pode ser modificado através do parâmetro de configuração `role`. Para segurança extra, você pode ainda mudar o nome do parâmetro de URL através da configuração `parameter`:

- *YAML*

```yaml
# app/config/security.yml
security:
    firewalls:
        main:
            // ...
            switch_user: { role: ROLE_ADMIN, parameter: _want_to_be_this_user }
```

- *XML*

```xml
<!-- app/config/security.xml -->
<config>
    <firewall>
        <!-- ... -->
        <switch-user role="ROLE_ADMIN" parameter="_want_to_be_this_user" />
    </firewall>
</config>
```

- *PHP*

```php
// app/config/security.php
$container->loadFromExtension('security', array(
    'firewalls' => array(
        'main'=> array(
            // ...
            'switch_user' => array('role' => 'ROLE_ADMIN', 'parameter' => '_want_to_be_this_user
        ),
    ),
));
```

Autenticação Sem Estado

Por padrão, o Symfony2 confia a um cookie (a Session) para persistir o contexto de segurança de um usuário. Se você utiliza, porém, certificados ou autenticação HTTP, por exemplo, persistência não é necessário já que as credenciais estão disponíveis em cada requisição. Neste caso, e se não precisar de armazenar nada entre as requisições, você pode ativar a autenticação sem estado (que significa que nenhum cookie será criado pelo Symfony2):

- *YAML*

```yaml
# app/config/security.yml
security:
    firewalls:
        main:
            http_basic: ~
            stateless:  true
```

- *XML*

```xml
<!-- app/config/security.xml -->
<config>
    <firewall stateless="true">
        <http-basic />
```

```
        </firewall>
    </config>
```

- *PHP*

```
    // app/config/security.php
    $container->loadFromExtension('security', array(
        'firewalls' => array(
            'main' => array('http_basic' => array(), 'stateless' => true),
        ),
    ));
```

Nota: Se utiliza formulário de login, o Symfony2 criará um cookie mesmo se você definir `stateless` como `true`.

Palavras Finais

Segurança pode ser um assunto profundo e complexo de se resolver em uma aplicação. Felizmente, o componente de segurança do Symfony segue um bom modelo baseado em *autenticação* e *autorização*. Autenticação, que sempre acontece antes, é gerenciada pelo firewall cujo trabalho é determinar a identidade do usuário através de diversos possíveis métodos (exemplo, autenticação HTTP, formulário de login, etc). No passo-a-passo, você encontrará exemplos de como outros métodos de autenticação podem ser utilizados, incluindo como implementar o funcionalidade de "Lembrar de mim" baseada em cookie.

Uma vez que o usuário está autenticado, a camada de autorização pode determinar se o usuário deve ou não deve ter acesso a um recurso específico. Comumente, *perfis* são aplicados a URLs, classes ou métodos e se o usuário atual não possuir o perfil, o acesso é negado. A camada de autorização, porém, é muito mais extensa e segue o sistema de votação onde várias partes podem determinar se o usuário atual deve ter acesso a determinado recurso. Saiba mais sobre este e outros tópicos no passo-a-passo.

Aprenda mais do Passo-a-Passo

- Forçando HTTP/HTTPS
- `Coloque usuários por IP na lista negra com um voter personalizado`
- Listas de Controle de Acesso (ACLs)
- `/cookbook/security/remember_me`

2.1.12 HTTP Cache

A natureza das aplicações web ricas é que elas sejam dinâmicas. Não importa quão eficiente seja sua aplicação, cada uma das requisições sempre terá uma carga maior do que se ela servisse um arquivo estático.

E, para a maior parte das aplicações web, isso não é problema. O Symfony 2 é extremamente rápido e, a menos que você esteja fazendo algo extramente pesado, as requisições serão retornadas rapidamente sem sobrecarregar demais o seu servidor.

Mas, a medida que seu site cresce, essa carga adicional pode se tornar um problema. O processamento que normalmente é efetuado a cada requisição deveria ser feito apenas uma vez. É exatamente esse o objetivo do cache.

Fazendo Cache nos Ombros de Gigantes

O modo mais efetivo de melhorar a performance de uma aplicação é fazendo cache da saída completa de uma página e então ignorar a aplicação totalmente nas requisições seguintes. É claro, nem sempre isso é possível para sites altamente dinâmicos. Ou será que é? Nesse capítulo, veremos como o sistema de cache do Symfony2 trabalha e por que nós acreditamos que esta é a melhor abordagem possível.

O sistema de cache do Symfony2 é diferente porque ele se baseia na simplicidade e no poder do cache HTTP como definido na especificação HTTP. Em vez de inventar uma nova metodologia de cache, o Symfony2 segue o padrão que define a comunicação básica na Web. Quando você entender os modelos fundamentais de validação e expiração de cache HTTP estará pronto para dominar o sistema de cache do Symfony2.

Para os propósitos de aprender como fazer cache com o Symfony2, cobriremos o assunto em quatro passos:

- **Passo 1**: Um *gateway cache*, ou proxy reverso, é uma camada independente que fica na frente da sua aplicação. O proxy reverso faz o cache das respostas quando elas são retornadas pela sua aplicação e responde as requisições com respostas cacheadas antes que elas atinjam sua aplicação. O Symfony2 fornece um proxy reverso próprio, mas qualquer proxy reverso pode ser usado.
- **Passo 2**: Cabeçalhos de cache:ref:*cache HTTP<http-cache-introduction>* são usados para comunicar com o gateway cache e qualquer outro cache entre sua aplicação e o cliente. O Symfony2 fornece padrões razoáveis e uma interface poderosa para interagir com os cabeçalhos de cache.
- **Passo 3**: *Expiração e validação* HTTP são dois modelos usados para determinar se o conteúdo cacheado é *atual/fresh* (pode ser reutilizado a partir do cache) ou se o conteúdo é *antigo/stale* (deve ser recriado pela aplicação).
- **Passo 4**: *Edge Side Includes* (ESI) permitem que sejam usados caches HTTP para fazer o cache de fragmentos de páginas (mesmo fragmentos aninhados) independentemente. Com o ESI, você pode até fazer o cache de uma página inteira por 60 minutos, e uma barra lateral embutida por apenas 5 minutos.

Como fazer cache com HTTP não é uma coisa apenas do Symfony, já existem muitos artigos sobre o assunto. Se você for iniciante em cache HTTP, recomendamos *fortemente* o artigo Things Caches Do do Ryan Tomayko. Outra fonte aprofundada é o Cache Tutorial do Mark Nottingham.

Fazendo Cache com um Gateway Cache

Quando se faz cache com HTTP, o *cache* é separado completamente da sua aplicação e se coloca entre sua aplicação e o cliente que está fazendo a requisição.

O trabalho do cache é receber requisições do cliente e transferi-las para sua aplicação. O cache também receberá de volta respostas da sua aplicação e as encaminhará para o cliente. O cache é o "middle-man" da comunicação requisição-resposta entre o cliente e sua aplicação.

Ao longo do caminho, o cache guardará toda resposta que seja considerada "cacheável" (Veja *Introdução ao Cache HTTP*). Se o mesmo recurso for requisitado novamente, o cache irá mandar a resposta cacheada para o cliente, ignorando completamente sua aplicação.

Esse tipo de cache é conhecido como um gateway cache HTTP e existem vários como o Varnish, o Squid in reverse proxy mode e o proxy reverso do Symfony2.

Tipos de Cache

Mas um gateway cache não é o único tipo de cache. Na verdade, os cabeçalhos de cache HTTP enviados pela sua aplicação são consumidos e interpretados por três tipos diferentes de cache:

- *Caches de Navegador*: Todo navegador vem com seu próprio cache local que é útil principalmente quando você aperta o "voltar" ou para imagens e outros assets. O cache do navegador é um cache *privado* assim os recursos cacheados não são compartilhados com ninguém mais.
- *Caches de Proxy*: Um proxy é um cache *compartilhado* assim muitas pessoas podem utilizar um único deles. Ele geralmente é instalado por grandes empresas e ISPs para reduzir a latência e o tráfego na rede.
- *Caches Gateway*: Como um proxy, ele também é um cache *compartilhado* mas no lado do servidor. Instalado por administradores de rede, ele torna os sites mais escaláveis, confiáveis e performáticos.

Dica: Caches gateway algumas vezes são referenciados como caches de proxy reverso, surrogate caches ou até aceleradores HTTP.

Nota: A diferença entre os caches *privados* e os *compartilhados* se torna mais óbvia a medida que começamos a falar sobre fazer cache de respostas com conteúdo que é específico para exatamente um usuário (e.g. informação de uma conta).

Toda resposta da sua aplicação irá provavelmente passar por um ou ambos os dois primeiros tipos de cache. Esses caches estão fora de seu controle mas eles seguem o direcionamento do cache HTTP definido na resposta.

Proxy Reverso do Symfony2

O Symfony2 vem com um proxy reverso (também chamado de gateway cache) escrito em PHP. É só habilitá-lo e as respostas cacheáveis da sua aplicação começaram a ser cacheadas no mesmo momento. Sua instalação é bem simples. Toda nova aplicação Symfony2 vem com um kernel de cache pré-configurado (`AppCache`) que encapsula o kernel padrão (`AppKernel`). O Kernel de cache *é* o proxy reverso.

Para habilitar o cache, altere o código do front controller para utilizar o kernel de cache:

```
// web/app.php

require_once __DIR__.'/../app/bootstrap.php.cache';
require_once __DIR__.'/../app/AppKernel.php';
require_once __DIR__.'/../app/AppCache.php';

use Symfony\Component\HttpFoundation\Request;

$kernel = new AppKernel('prod', false);
$kernel->loadClassCache();
// wrap the default AppKernel with the AppCache one
$kernel = new AppCache($kernel);
$request = Request::createFromGlobals();
$response = $kernel->handle($request);
$response->send();
$kernel->terminate($request, $response);
```

O kernel de cache funcionará imediatamente como um proxy reverso - fazendo cache das respostas da sua aplicação e retornando-as para o cliente.

Dica: O kernel de cache tem um método especial `getLog()` que retorna uma representação em texto do que ocorreu na camada de cache. No ambiente de desenvolvimento, utilize-o para depurar e validar sua estratégia de cache:

```
error_log($kernel->getLog());
```

O objeto `AppCache` tem uma configuração padrão razoável, mas ela pode receber um ajuste fino por meio de um conjunto de opções que podem ser definidas sobrescrevendo o método `getOptions()`:

```
// app/AppCache.php
class AppCache extends Cache
{
    protected function getOptions()
    {
        return array(
            'debug'                  => false,
            'default_ttl'            => 0,
            'private_headers'        => array('Authorization', 'Cookie'),
            'allow_reload'           => false,
            'allow_revalidate'       => false,
            'stale_while_revalidate' => 2,
            'stale_if_error'         => 60,
        );
    }
}
```

Dica: A menos que seja sobrescrita em `getOptions()`, a opção `debug` será definida como o valor padrão de depuração no `AppKernel` envolvido.

Aqui vai uma lista das opções principais:

- `default_ttl`: O número de segundos que uma entrada do cache deve ser considerada como atual quando nenhuma informação de atualização for passada na resposta. Os cabeçalhos explícitos `Cache-Control` e `Expires` sobrescrevem esse valor (padrão: `0`);
- `private_headers`: Conjunto de cabeçalhos de requisição que acionam o comportamento `Cache-Control` "privado" nas respostas que não declaram explicitamente se a resposta é `public` ou `private` por meio de uma diretiva `Cache-Control`. (padrão: `Authorization` e `Cookie`);
- `allow_reload`: Diz se o cliente pode forçar um recarregamento do cache incluindo uma diretiva `Cache-Control` "no-cache" na requisição. Defina ele como `true` para seguir a RFC 2616 (padrão: `false`);
- `allow_revalidate`: Diz se o cliente pode forçar uma revalidação do cache incluindo uma diretiva `Cache-Control` "max-age=0" na requisição. Defina ele como `true` para seguir a RFC 2616 (padrão: false);
- `stale_while_revalidate`: Diz o número padrão de segundos (a granularidade é o segundo como na precisão da Resposta TTL) durante o qual o cache pode retornar imediatamente uma resposta antiga enquanto ele faz a revalidação dela no segundo plano (padrão: `2`); essa configuração é sobrescrita pela extensão `stale-while-revalidate` do `Cache-Control` HTTP (veja RFC 5861);
- `stale_if_error`: Diz o número padrão de segundos (a granularidade é o segundo) durante o qual o cache pode fornecer uma resposta antiga quando um erro for encontrado (padrão: `60`). Essa configuração é sobrescrita pela extensão `stale-if-error` do `Cache-Control` HTTP (veja RFC 5861).

Se `debug` for `true`, o Symfony2 adiciona automaticamente um cabeçalho `X-Symfony-Cache` na resposta contendo informações úteis sobre o que o cache serviu ou deixou passar.

Mudando de um Proxy Reverso para Outro

O proxy reverso do Symfony2 é uma ferramenta importante quando estiver desenvolvendo o seu site ou quando você faz o deploy de seu site num servidor compartilhado onde você não pode instalar nada mais do que código PHP. Mas como ele é escrito em PHP, não há como ele ser tão rápido quanto um proxy escrito em C. É por isso que recomendamos fortemente que você utilize o Varnish ou o Squid no seu servidor de produção quando for possível. A boa notícia é que mudar entre um servidor de proxy para outro é fácil e transparente pois nenhuma alteração de código é necessária em sua aplicação. Inicie de forma simples com o proxy reverso do Symfony2 e depois atualize para o Varnish quando o seu tráfego aumentar.
Para mais informações de como usar o Varnish com o Symfony2, veja o capítulo How to use Varnish do cookbook.

Nota: A performance do proxy reverso do Symfony2 não depende da complexidade da sua aplicação. Isso acontece porque o kernel da aplicação só é carregado quando a requisição precisar ser passada para ele.

Introdução ao Cache HTTP

Para tirar vantagem das camadas de cache disponíveis, sua aplicação precisa ser capaz de informar quais respostas são cacheáveis e as regras que governam quando/como o cache o se torna antigo. Isso é feito configurando os cabeçalhos HTTP na sua resposta.

Dica: Lembre que o "HTTP" nada mais é do que uma linguagem (um linguagem de texto simples) que os clientes web (e.g navegadores) e os servidores web utilizam para se comunicar uns com os outros. Quando falamos sobre o cache HTTP, estamos falando sobre a parte dessa linguagem que permite que os clientes e servidores troquem informações relacionadas ao cache.

O HTTP define quatro cabeçalhos de cache para as respostas que devemos nos preocupar:

- `Cache-Control`
- `Expires`
- `ETag`
- `Last-Modified`

O cabeçalho mais importante e versátil é o cabeçalho `Cache-Control`, que na verdade é uma coleção de várias informações de cache.

Nota: Cada um dos cabeçalhos será explicado detalhadamente na seção *Expiração e Validação HTTP*.

O Cabeçalho Cache-Control

O cabeçalho `Cache-Control` é único pois ele contém não um, mas vários pedaços de informação sobre a possibilidade de cache de uma resposta. Cada pedaço de informação é separada por uma vírgula:

```
Cache-Control: private, max-age=0, must-revalidate

Cache-Control: max-age=3600, must-revalidate
```

O Symfony fornece uma abstração em volta do cabeçalho `Cache-Control` para deixar sua criação mais gerenciável:

```php
$response = new Response();

// mark the response as either public or private
$response->setPublic();
$response->setPrivate();

// set the private or shared max age
$response->setMaxAge(600);
$response->setSharedMaxAge(600);

// set a custom Cache-Control directive
$response->headers->addCacheControlDirective('must-revalidate', true);
```

Respostas Públicas vs Privadas

Tanto o gateway cache quando o proxy cache são considerados caches "compartilhados" pois o conteúdo cacheado é compartilhado por mais de um usuário. Se uma resposta específica de um usuário for incorretamente armazenada por um cache compartilhado, ela poderia ser retornada posteriormente para um número incontável de usuários diferentes. Imagine se a informação da sua conta fosse cacheada e depois retornada para todos os usuários que em seguida solicitassem a página da conta deles!

Para lidar com essa situação, cada resposta precisa ser configurada para ser pública ou privada:

- *public*: Indica que a resposta pode ser cacheada tanto por caches privados quanto pelos compartilhados;
- *private*: Indica que a mensagem toda ou parte da resposta é destinada para um único usuário e não deve ser cacheada por um cache compartilhado.

O Symfony tem como padrão conservador definir toda resposta como privada. Para se beneficiar dos caches compartilhados (como o proxy reverso do Symfony2), a resposta precisa ser definida como pública explicitamente.

Métodos Seguros

O cache HTTP só funciona para os métodos HTTP "seguros" (como o GET e o HEAD). Ser seguro significa que você não consegue alterar o estado da aplicação no servidor quando estiver respondendo a requisição (é claro que você pode logar a informação, fazer cache dos dados etc). Isso tem duas consequências importantes:

- Você *nunca* deve alterar o estado de sua aplicação quando estiver respondendo uma requisição GET ou HEAD. Mesmo se você não usar um gateway cache, a presença de caches proxy faz com que qualquer requisição GET ou HEAD possa atingir ou não seu servidor.
- Não espere que os métodos PUT, POST ou DELETE sejam cacheados. Esses métodos são destinados para serem utilizados quando se quer alterar o estado da sua aplicação (e.g. excluir uma postagem de um blog). Fazer cache desses métodos poderia fazer com que certas requisições não chegassem na sua aplicação e a alterasse.

Regras e Padrões de Cache

O HTTP 1.1 permite por padrão fazer o cache de qualquer coisa a menos que seja explícito que não num cabeçalho `Cache-Control`. Na prática, a maioria dos caches não faz nada quando as requisições tem um cookie, um cabeçalho de autorização, usam um método inseguro (i.e PUT, POST, DELETE) ou quando as respostas tem código de estado para redirecionamento.

O Symfony2 define automaticamente um cabeçalho `Cache-Control` conservador quando o desenvolvedor não definir nada diferente seguindo essas regras:

- Se não for definido cabeçalho de cache (`Cache-Control`, `Expires`, `ETag` or `Last-Modified`), `Cache-Control` é configurado como `no-cache`, indicando que não será feito cache da resposta;
- Se `Cache-Control` estiver vazio (mas outros cabeçalhos de cache estiverem presentes), seu valor é configurado como `private, must-revalidate`;
- Mas se pelo menos uma diretiva‘‘Cache-Control‘‘ estiver definida, e nenhuma diretiva ‘public’ ou `privada` tiver sido adicionada explicitamente, o Symfony2 adiciona automaticamente a diretiva `private` (exceto quando `s-maxage` estiver definido).

Expiração e Validação HTTP

A especificação HTTP define dois modelos de cache:

- Com o expiration model, você especifica simplesmente quanto tempo uma resposta deve ser considerada “atual” incluindo um cabeçalho `Cache-Control` e/ou um `Expires`. Os caches que entendem a expiração não farão a mesma requisição até que a versão cacheada atinja o tempo de expiração e se torne “antiga”.
- Quando as páginas são realmente dinâmicas (i.e. sua representação muda constantemente), o validation model é frequentemente necessário. Com esse modelo, o cache armazena a resposta, mas pergunta ao servidor a cada requisição se a resposta cacheada continua válida ou não. A aplicação utiliza um identificador único da resposta (o cabeçalho `Etag`) e/ou um timestamp (o cabeçalho `Last-Modified`) para verificar se a página mudou desde quando tinha sido cacheada.

O objetivo de ambos os modelos é nunca ter que gerar a mesma resposta duas vezes contando com o cache para guardar e retornar respostas “atuais”.

Lendo a Especificação HTTP

A especificação HTTP define uma linguagem simples mas poderosa com a qual clientes e servidores podem se comunicar. Como um desenvolvedor web, o modelo requisição-resposta da especificação domina o seu trabalho. Infelizmente o documento real da especificação - RFC 2616 - pode ser difícil de ler.
Existe um trabalho em andamento (HTTP Bis) para reescrever a RFC 2616. Ele não descreve uma nova versão do HTTP, mas principalmente esclarece a especificação HTTP original. A organização também melhorou pois a especificação foi dividida em sete partes; tudo que for relacionado ao cache HTTP pode ser encontrado em duas partes dedicadas (P4 - Conditional Requests e P6 - Caching: Browser and intermediary caches)
Como desenvolvedor web, nós recomendamos fortemente que você leia a especificação. Sua clareza e poder - ainda mais depois de dez anos da sua criação - é incalculável. Não se engane com a aparência da especificação - o conteúdo dela é muito mais bonito que sua capa.

Expiração

O modelo de expiração é o mais eficiente e simples dos dois modelos de cache e deve ser usado sempre que possível. Quando uma resposta é cacheada com uma expiração, o cache irá armazenar a resposta e retorná-la diretamente sem acessar a aplicação até que a resposta expire.

O modelo de expiração pode ser aplicado usando um desses dois, quase idênticos, cabeçalhos HTTP: `Expires` ou `Cache-Control`.

Expiração com o Cabeçalho `Expires`

De acordo com a especificação HTTP, “o campo do cabeçalho `Expires` diz a data/horário a partir do qual a resposta é considerada antiga.” O cabeçalho `Expires` pode ser definido com o método `setExpires() Response`. Ele

recebe uma instância de `DateTime` como argumento:

```
$date = new DateTime();
$date->modify('+600 seconds');

$response->setExpires($date);
```

O cabeçalho HTTP resultante se parecerá com isso:

```
Expires: Thu, 01 Mar 2011 16:00:00 GMT
```

Nota: O método `setExpires()` converte automaticamente a data para o fuso horário GMT como exigido na especificação.

O cabeçalho `Expires` sofre com duas limitações. Primeiro, o relógio no servidor web e o cache (e.g. o navegador) precisam estar sincronizados. A outra é que a especificação define que "servidores HTTP/1.1 não devem mandar datas `Expires` com mais de um ano no futuro."

Expiração com o Cabeçalho `Cache-Control`

Devido às limitações do cabeçalho `Expires`, na maioria das vezes, você deve usar no lugar dele o cabeçalho `Cache-Control`. Lembre-se que o cabeçalho `Cache-Control` é usado para especificar várias diretivas de cache diferentes. Para expiração, existem duas diretivas: `max-age` e `s-maxage`. A primeira é usada para todos os caches enquanto a segunda somente é utilizada por caches compartilhados:

```
// Define o número de segundos após o qual a resposta
// não será mais considerada atual
$response->setMaxAge(600);

// O mesmo que acima, mas apenas para caches compartilhados
$response->setSharedMaxAge(600);
```

O cabeçalho `Cache-Control` deve ter o seguinte formato (ele pode ter diretivas adicionais):

```
Cache-Control: max-age=600, s-maxage=600
```

Validação

Quando um recurso precisa ser atualizado logo que uma mudança for feita em dados relacionados, o modelo de expiração é insuficiente. Com o modelo de expiração, a aplicação não será acionada para retornar a resposta atualizada até que o cache finalmente se torne antigo.

O modelo de validação resolve esse problema. Nesse modelo, o cache continua a armazenar as respostas. A diferença é que, para cada requisição, o cache pede para a aplicação verificar se a respostas cacheada continua válida ou não. Se o cache ainda *for* válido, sua aplicação deve retornar um código de estado 304 e nenhum conteúdo. Isso diz para o cache que ele pode retornar a resposta cacheada.

Nesse modelo, você economiza principalmente banda pois a representação não é enviada duas vezes para o mesmo cliente (uma resposta 304 é mandada no lugar). Mas se projetar sua aplicação com cuidado, você pode ser capaz de pegar o mínimo de dados necessário para enviar uma resposta 304 e também economizar CPU (veja abaixo um exemplo de uma implementação).

Dica: O código de estado 304 significa "Not Modified". Isso é importante pois com esse código de estado *não* é colocado o conteúdo real que está sendo requisitado. Em vez disso, a resposta é simplesmente um conjunto leve de

direções que dizem ao cache para ele utilizar uma versão armazenada.

Assim como com a expiração, existem dois cabeçalhos HTTP diferentes que podem ser utilizados para implementar o modelo de validação: `ETag` e `Last-Modified`.

Validação com o Cabeçalho `ETag`

O cabeçalho‘‘ETag‘‘ é um cabeçalho em texto (chamado de “entity-tag”) que identifica de forma única uma representação do recurso alvo. Ele é totalmente gerado e configurado pela sua aplicação, dessa forma você pode dizer, por exemplo, se o recurso `/about` que foi armazenado pelo cache está atualizado com o que sua aplicação poderia retornar. Uma `ETag` é como uma impressão digital e é utilizada para comparar rapidamente se duas versões diferentes de um recurso são equivalentes. Como as impressões digitais, cada `ETag` precisa ser única em todos as representações do mesmo recurso.

Vamos analisar uma implementação simples que gera a ETag como o hash md5 do conteúdo:

```
public function indexAction()
{
    $response = $this->render('MyBundle:Main:index.html.twig');
    $response->setETag(md5($response->getContent()));
    $response->isNotModified($this->getRequest());

    return $response;
}
```

O método `Response::isNotModified()` compara o `ETag` enviado na `Request` com o que está definido na `Response`. Se os dois combinarem, o método define automaticamente o código de estado da `Response` como 304.

Esse algoritmo é simples o suficiente e bem genérico, mas você precisa criar a `Response` inteira antes de ser capaz de calcular a Etag, o que não é o melhor possível. Em outras palavras, isso economiza banda de rede mas não faz o mesmo com os ciclos de CPU.

Na seção *Otimizando seu Código com Validação*, nós mostraremos como a validação pode ser utilizada de forma mais inteligente para determinar a validade de um cache sem muito trabalho.

Dica: O Symfony2 também suporta ETags fracas passando `true` como segundo argumento para o método `setETag()`.

Validação com o Cabeçalho `Last-Modified`

O cabeçalho `Last-Modified` é a segunda forma de validação. De acordo com a especificação HTTP, “O campo do cabeçalho `Last-Modified` indica a data e o horário que o servidor de origem acredita que a representação foi modificada pela última vez.” Em outras palavras, a aplicação decide se o conteúdo cacheado foi atualizado ou não tendo como base se ele foi atualizado desde que a resposta foi cacheada.

Por exemplo, você pode usar a última data de atualização de todos os objetos necessários para calcular a representação do recurso como o valor para o cabeçalho `Last-Modified`:

```
public function showAction($articleSlug)
{
    // ...

    $articleDate = new \DateTime($article->getUpdatedAt());
    $authorDate = new \DateTime($author->getUpdatedAt());
```

```
    $date = $authorDate > $articleDate ? $authorDate : $articleDate;

    $response->setLastModified($date);
    $response->isNotModified($this->getRequest());

    return $response;
}
```

O método `Response::isNotModified()` compara o cabeçalho `If-Modified-Since` mandado pela requisição com o cabeçalho `Last-Modified` definido na resposta. Se eles forem equivalentes, a `Response` será configurada com um código de estado 304.

Nota: O cabeçalho da requisição `If-Modified-Since` é igual ao cabeçalho `Last-Modified` de uma resposta enviada ao cliente para um recurso

> específico. Essa é a forma como o cliente e o servidor se comunicam entre si e decidem se o recurso foi ou não atualizado desde que ele foi

cacheado.

Otimizando seu Código com Validação

O objetivo principal de qualquer estratégia de cache é aliviar a carga sobre a aplicação. Colocando de outra forma, quanto menos coisas você fizer na sua aplicação para retornar uma resposta 304, melhor. O método `Response::isNotModified()` faz exatamente isso expondo um padrão simples e eficiente:

```
public function showAction($articleSlug)
{
    // Pega a informação mínima para calcular
    // a ETag ou o valor Last-Modified
    // (baseado na Requisição, o dado é recuperado
    // de um banco de dados ou de um armazenamento chave-valor)

    $article = // ...

    // cria uma Resposta com uma ETag e/ou um cabeçalho Last-Modified
    $response = new Response();
    $response->setETag($article->computeETag());
    $response->setLastModified($article->getPublishedAt());

    // Verifica se a Resposta não é diferente da Requisição
    if ($response->isNotModified($this->getRequest())) {
        // retorna imediatamente a Resposta 304
        return $response;
    } else {
        // faça mais algumas coisas aqui - como buscar mais dados
        $comments = // ...

        // ou renderize um template com a $response que você já
        // inicializou
        return $this->render(
            'MyBundle:MyController:article.html.twig',
            array('article' => $article, 'comments' => $comments),
            $response
        );
```

```
    }
}
```

Quando a `Response` não tiver sido modificada, `isNotModified()` automaticamente define o código de estado para `304`, remove o conteúdo e remove alguns cabeçalhos que não podem estar presentes em respostas `304` (veja `setNotModified()`).

Variando a Resposta

Até agora assumimos que cada URI tem exatamente uma representação do recurso alvo. Por padrão, o cache HTTP é feito usando a URI do recurso como a chave do cache. Se duas pessoas requisitarem a mesma URI de um recurso passível de cache, a segunda pessoa receberá a versão cacheada.

Algumas vezes isso não é suficiente e diferentes versões da mesma URI precisam ser cacheadas baseando-se em um ou mais valores dos cabeçalhos de requisição. Por exemplo, se você comprimir a página quando o cliente suportar compressão, cada URI terá duas representações: uma quando o cliente suportar compressão e uma quando o cliente não suportar. Essa decisão é feita usando o valor do cabeçalho `Accept-Encoding` da requisição.

Nesse caso, precisamos que o cache armazene ambas as versões da requisição, para uma determinada URI, comprimida e não, e retorne-as se baseando no valor `Accept-Encoding` da requisição. Isso é feito utilizando o cabeçalho `Vary` da resposta, que é uma lista separada por vírgulas dos diferentes cabeçalhos cujos valores acionam um representação diferente do recurso requisitado:

```
Vary: Accept-Encoding, User-Agent
```

Dica: Esse cabeçalho `Vary` específico pode fazer o cache de diferentes versões de cada recurso baseado na URI e no valor dos cabeçalhos `Accept-Encoding` e `User-Agent` da requisição.

O objeto `Response` fornece um interface limpa para gerenciar o cabeçalho `Vary`:

```
// define um cabeçalho vary
$response->setVary('Accept-Encoding');

// define múltiplos cabeçalhos vary
$response->setVary(array('Accept-Encoding', 'User-Agent'));
```

O método `setVary()` recebe o nome de um cabeçalho ou um array de nomes de cabeçalhos para os quais a resposta varia.

Expiração e Validação

É claro que você pode usar ambas a validação e a expiração dentro da mesma `Response`. Como a expiração é mais importante que a validação, você pode se beneficiar facilmente do melhor dos dois mundos. Em outras palavras, utilizando ambas a expiração e a validação, você pode ordenar que o cache sirva o conteúdo cacheado, ao mesmo tempo que verifica em algum intervalo de tempo (a expiração) para ver se o conteúdo continua válido.

Mais Métodos Response

A classe Response fornece muitos outros métodos relacionados ao cache. Aqui seguem os mais úteis:

```
// Marca a Resposta como antiga
$response->expire();

// Força a resposta para retornar um 304 apropriado sem conteúdo
$response->setNotModified();
```

Adicionalmente, a maioria dos cabeçalhos HTTP relacionados ao cache podem ser definidos por meio do método `setCache()` apenas:

```
// Define as configurações de cache em uma única chamada
$response->setCache(array(
    'etag'          => $etag,
    'last_modified' => $date,
    'max_age'       => 10,
    's_maxage'      => 10,
    'public'        => true,
    // 'private'    => true,
));
```

Usando Edge Side Includes

Os gateway caches são uma excelente maneira de fazer o seu site ficar mais performático. Mas ele tem uma limitação: só conseguem fazer cache de páginas completas. Se você não conseguir cachear páginas completas ou se partes de uma página tiverem partes "mais" dinâmicas, você está sem sorte. Felizmente, o Symfony2 fornece uma solução para esses casos, baseado numa tecnologia chamada ESI, ou Edge Side Includes. A Akamai escreveu essa especificação quase 10 anos atrás, e ela permite que partes específicas de uma página tenham estratégias de cache diferentes da página principal.

A especificação ESI descreve tags que podem ser embutidas em suas páginas para comunicar com o gateway cache. Apenas uma tag é implementada no Symfony2, `include`, pois ela é a única que é útil fora do contexto da Akamai:

```
<html>
    <body>
        Some content

        <!-- Embed the content of another page here -->
        <esi:include src="http://..." />

        More content
    </body>
</html>
```

Nota: Perceba pelo exemplo que cada tag ESI tem uma URL completamente válida. Uma tag ESI representa um fragmento de página que pode ser recuperado pela URL informada.

Quando uma requisição é tratada, o gateway cache busca a página inteira do cache ou faz a requisição no backend da aplicação. Se a resposta contiver uma ou mais tags ESI, elas são processadas da mesma forma. Em outras palavras, o cache gateway pega o fragmento da página inserido ou do seu cache ou faz a requisição novamente para o backend da aplicação. Quado todas as tags ESI forem resolvidas, o gateway cache mescla cada delas na página principal e envia o conteúdo finalizado para o cliente.

Tudo isso acontece de forma transparente no nível do gateway cache (i.e. fora de sua aplicação). Como você pode ver, se você escolher tirar proveito das tags ESI, o Symfony2 faz com que o processo de incluí-las seja quase sem esforço.

Usando ESI no Symfony2

Primeiro, para usar ESI, tenha certeza de ter feito sua habilitação na configuração da sua aplicação:

- *YAML*

```
# app/config/config.yml
framework:
    # ...
    esi: { enabled: true }
```

- *XML*

```
<!-- app/config/config.xml -->
<framework:config ...>
    <!-- ... -->
    <framework:esi enabled="true" />
</framework:config>
```

- *PHP*

```
// app/config/config.php
$container->loadFromExtension('framework', array(
    // ...
    'esi'    => array('enabled' => true),
));
```

Agora, suponha que temos uma página que seja relativamente estática, exceto por um atualizador de notícias na parte inferior do conteúdo. Com o ESI, podemos fazer o cache do atualizador de notícias de forma independente do resto da página.

```
public function indexAction()
{
    $response = $this->render('MyBundle:MyController:index.html.twig');
    $response->setSharedMaxAge(600);

    return $response;
}
```

Nesse exemplo, informamos o tempo de vida para o cache da página inicial como dez minutos. Em seguida, vamos incluir o atualizador de notícias no template embutindo uma action. Isso é feito pelo helper `render` (Veja *Incorporação de Controllers* para mais detalhes).

Como o conteúdo embutido vem de outra página (ou controller nesse caso), o Symfony2 usa o helper padrão `render` para configurar as tags ESI:

- *Twig*

```
{% render '...:news' with {}, {'standalone': true} %}
```

- *PHP*

```
<?php echo $view['actions']->render('...:news', array(), array('standalone' => true)) ?>
```

Definindo `standalone` como `true`, você diz ao Symfony2 que a action deve ser renderizada como uma tag ESI. Você pode estar imaginando porque você iria querer utilizar um helper em vez de escrever a tag ESI você mesmo. Isso acontece porque a utilização de um helper faz a sua aplicação funcionar mesmo se não existir nenhum gateway cache instalado. Vamos ver como isso funciona.

Quando standalone está `false` (o padrão), o Symfony2 mescla o conteúdo da página incluída com a principal antes de mandar a resposta para o cliente. Mas quando standalone está `true`, *e* se o Symfony detectar que ele está falando

com um gateway cache que suporta ESI, ele gera uma tag ESI de inclusão. Mas se não houver um gateway cache ou se ele não suportar ESI, o Symfony2 apenas mesclará o conteúdo da página incluída com a principal como se tudo tivesse sido feito com o standalone definido como `false`.

Nota: O Symfony2 detecta se um gateway cache suporta ESI por meio de outra especificação da Akamai que é suportada nativamente pelo proxy reverso do Symfony2.

A action embutida agora pode especificar suas próprias regras de cache, de forma totalmente independente da página principal.

```
public function newsAction()
{
  // ...

  $response->setSharedMaxAge(60);
}
```

Com o ESI, o cache da página completa pode ficar válido por 600 segundos, mas o cache do componente de notícias será válido apenas nos últimos 60 segundos.

Um requisito do ESI, no entanto, é que a action embutida seja acessível por uma URL dessa forma o gateway cache pode acessá-la independentemente do restante da página. É claro que uma action não pode ser acessada pela URL a menos que exista uma rota que aponte para ela. O Symfony2 trata disso por meio de uma rota e um controller genéricos. Para que a tag ESI de inclusão funcione adequadamente, você precisa definir a rota `_internal`:

- *YAML*

```
# app/config/routing.yml
_internal:
    resource: "@FrameworkBundle/Resources/config/routing/internal.xml"
    prefix:   /_internal
```

- *XML*

```
<!-- app/config/routing.xml -->
<?xml version="1.0" encoding="UTF-8" ?>

<routes xmlns="http://symfony.com/schema/routing"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/routing http://symfony.com/schema/routing/rout

    <import resource="@FrameworkBundle/Resources/config/routing/internal.xml" prefix="/_internal
</routes>
```

- *PHP*

```
// app/config/routing.php
use Symfony\Component\Routing\RouteCollection;
use Symfony\Component\Routing\Route;

$collection->addCollection($loader->import('@FrameworkBundle/Resources/config/routing/internal.x

return $collection;
```

Dica: Como essa rota permite que todas as actions sejam acessadas por uma URL, talvez você queira protegê-la utilizando a funcionalidade de firewall do Symfony2 (restringindo o acesso para a faixa de IP do seu proxy reverso).

Veja a seção *Securing by IP* do Security Chapter para mais informações de como fazer isso.

Uma grande vantagem dessa estratégia de cache é que você pode fazer com que sua aplicação seja tão dinâmica quanto for necessário e ao mesmo tempo, acessar a aplicação o mínimo possível.

Nota: Assim que você começar a utilizar ESI, lembre-se de sempre usar a diretiva `s-maxage` em vez de `max-age`. Como o navegador somente recebe o recurso agregado, ele não tem ciência dos sub-componentes e assim ele irá obedecer a diretiva `max-age` e fazer o cache da página inteira. E você não quer isso.

O helper `render` suporta duas outras opções úteis:

- `alt`: usada como o atributo `alt` na tag ESI, que permite que você especifique uma URL alternativa para ser usada se o `src` não for encontrado;
- `ignore_errors`: se for configurado como true, um atributo `onerror` será adicionado ao ESI com um valor `continue` indicando que, em caso de falha, o gateway cache irá simplesmente remover a tag ESI silenciosamente.

Invalidação do Cache

> "Só tem duas coisas difíceis em Ciência da Computação: invalidação de cache e nomear coisas." –Phil Karlton

Você nunca deveria ter que invalidar dados em cache porque a invalidação já é feita nativamente nos modelos de cache HTTP. Se você usar validação, por definição você nunca precisaria invalidar algo; e se você usar expiração e precisar invalidar um recurso, isso significa que você definiu a data de expiração com um valor muito longe.

Nota: Como invalidação é um tópico específico para cada tipo de proxy reverso, se você não se preocupa com invalidação, você pode alternar entre proxys reversos sem alterar nada no código da sua aplicação.

Na verdade, todos os proxys reversos fornecem maneiras de expurgar dados do cache, mas você deveria evitá-los o máximo possível. A forma mais padronizada é expurgar o cache de uma determinada URL requisitando-a com o método HTTP especial `PURGE`.

Aqui vai como você pode configurar o proxy reverso do Symfony2 para suportar o método HTTP `PURGE`:

```
// app/AppCache.php
class AppCache extends Cache
{
    protected function invalidate(Request $request)
    {
        if ('PURGE' !== $request->getMethod()) {
            return parent::invalidate($request);
        }

        $response = new Response();
        if (!$this->getStore()->purge($request->getUri())) {
            $response->setStatusCode(404, 'Not purged');
        } else {
            $response->setStatusCode(200, 'Purged');
        }

        return $response;
```

```
    }
}
```

Cuidado: Você tem que proteger o método HTTP `PURGE` de alguma forma para evitar que pessoas aleatórias expurguem seus dados cacheados.

Sumário

O Symfony2 foi desenhado para seguir as regras já testadas: HTTP. O cache não é exceção. Dominar o sistema de cache do Symfony2 significa se familiarizar com os modelos de cache HTTP e utilizá-los de forma efetiva. Para isso, em vez de depender apenas da documentação do Symfony2 e exemplos de código, você deveria buscar mais conteúdo relacionado com o cache HTTP e caches gateway como o Varnish.

Learn more from the Cookbook

- Como usar Varnish para aumentar a velocidade do meu Website

2.1.13 Traduções

O termo "internacionalização" se refere ao processo de abstrair strings e outras peças com localidades específicas para fora de sua aplicação e dentro de uma camada onde eles podem ser traduzidos e convertidos baseados na localização do usuário (em outras palavras, idioma e páis). Para o texto, isso significa englobar cada um com uma função capaz de traduzir o texto (ou "messagem") dentro do idioma do usuário:

```
// text will *always* print out in English
echo 'Hello World';

// text can be translated into the end-user's language or default to English
echo $translator->trans('Hello World');
```

Nota: O termo *localidade* se refere rigorosamente ao idioma e país do usuário. Isto pode ser qualquer string que sua aplicação usa então para gerenciar traduções e outras diferenças de formato (ex: formato de moeda). Nós recomendamos o código de *linguagem* ISO639-1 , um underscore (_), então o código de *país* ISO3166 (ex: `fr_FR` para Francês/França).

Nesse capítulo, nós iremos aprender como preparar uma aplicação para suportar múltiplas localidades e então como criar traduções para localidade e então como criar traduções para múltiplas localidades. No geral, o processo tem vários passos comuns:

1. Abilitar e configurar o componente `Translation` do Symfony;
2. Abstrair strings (em outras palavras, "messagens") por englobá-las em chamadas para o `Translator`;
3. Criar translation resources para cada localidade suportada que traduza cada mensagem na aplicação;
4. Determinar, definir e gerenciar a localidade do usuário para o pedido e opcionalmente em toda a sessão do usuário.

Configuração

Traduções são suportadas por um `Translator` service que usa o localidadedo usuário para observar e retornar mensagens traduzidas. Antes de usar isto, abilite o `Translator` na sua configuração:

- *YAML*

```yaml
# app/config/config.yml
framework:
    translator: { fallback: en }
```

- *XML*

```xml
<!-- app/config/config.xml -->
<framework:config>
    <framework:translator fallback="en" />
</framework:config>
```

- *PHP*

```php
// app/config/config.php
$container->loadFromExtension('framework', array(
    'translator' => array('fallback' => 'en'),
));
```

A opção `fallback` define a localidade alternativa quando uma tradução não existe no localidadedo usuário.

Dica: Quando a tradução não existe para uma localidade, o tradutor primeiro tenta encontrar a tradução para o idioma (`fr` se o localidade é `fr_FR` por exemplo). Se isto também falhar, procura uma tradução usando a localidade alternativa.

A localidade usada em traduções é a que está armazenada no pedido. Isto é tipicamente definido através do atributo `_locale` em suas rotas (veja *A localidade e a URL*).

Tradução básica

Tradução do texto é feita done através do serviço `translator` (`Translator`). Para traduzir um bloco de texto (chamado de *messagem*), use o método `trans()`. Suponhamos, por exemplo, que estamos traduzindo uma simples mensagem de dentro do controller:

```php
public function indexAction()
{
    $t = $this->get('translator')->trans('Symfony2 is great');

    return new Response($t);
}
```

Quando esse código é executado, Symfony2 irá tentar traduzir a mensagem "Symfony2 is great" baseada na `localidade` do usuário. Para isto funcionar, nós precisamos informar o Symfony2 como traduzir a mensagem por um "translation resource", que é uma coleção de traduções de mensagens para um localidade especificada. Esse "dicionário" de traduções pode ser criado em diferentes formatos variados, sendo XLIFF o formato recomendado:

- *XML*

```xml
<!-- messages.fr.xliff -->
<?xml version="1.0"?>
<xliff version="1.2" xmlns="urn:oasis:names:tc:xliff:document:1.2">
    <file source-language="en" datatype="plaintext" original="file.ext">
        <body>
            <trans-unit id="1">
                <source>Symfony2 is great</source>
                <target>J'aime Symfony2</target>
```

```
                </trans-unit>
            </body>
        </file>
    </xliff>
```

- *PHP*

```
    // messages.fr.php
    return array(
        'Symfony2 is great' => 'J\'aime Symfony2',
    );
```

- *YAML*

```
    # messages.fr.yml
    Symfony2 is great: J'aime Symfony2
```

Agora, se o idioma do localidade do usuário é Francês (ex: `fr_FR` ou `fr_BE`), a mensagem irá ser traduzida para `J'aime Symfony2`.

O processo de tradução

Para realmente traduzir a mensagem, Symfony2 usa um processo simples:

- A `localidade` do usuário atual, que está armazenada no pedido (ou armazenada como `_locale` na sessão), é determinada;
- Um catálogo de mensagens traduzidas é carregado de translation resources definidos pelo `locale` (ex: `fr_FR`). Messagens da localidade alternativa são também carregadas e adiconadas ao catalogo se elas realmente não existem. O resultado final é um grande "dicionário" de traduções. Veja *Catálogo de Mensagens* para mais detalhes;
- Se a mensagem é localizada no catálogo, retorna a tradução. Se não, o tradutor retorna a mensagem original.

Quando usa o método `trans()`, Symfony2 procura pelo string exato dentro do catálogo de mensagem apropriada e o retorna (se ele existir).

Espaços reservados de mensagem

Às vezes, uma mensagem conteúdo uma variável precisa ser traduzida:

```
public function indexAction($name)
{
    $t = $this->get('translator')->trans('Hello '.$name);

    return new Response($t);
}
```

Entretanto criar uma tradução para este string é impossível visto que o tradutor irá tentar achar a mensagem exata, incluindo porções da variável (ex: "Hello Ryan" ou "Hello Fabien"). Ao invés escrever uma tradução para toda interação possível da mesma variável `$name` , podemos substituir a variável com um "espaço reservado":

```
public function indexAction($name)
{
    $t = $this->get('translator')->trans('Hello %name%', array('%name%' => $name));

    new Response($t);
}
```

Symfony2 irá procurar uma tradução da mensagem pura (`Hello %name%`) e *então* substitui o espaço reservado com os valores deles. Criar uma tradução é exatamente como foi feito anteriormente:

- *XML*

```
<!-- messages.fr.xliff -->
<?xml version="1.0"?>
<xliff version="1.2" xmlns="urn:oasis:names:tc:xliff:document:1.2">
    <file source-language="en" datatype="plaintext" original="file.ext">
        <body>
            <trans-unit id="1">
                <source>Hello %name%</source>
                <target>Bonjour %name%</target>
            </trans-unit>
        </body>
    </file>
</xliff>
```

- *PHP*

```
// messages.fr.php
return array(
    'Hello %name%' => 'Bonjour %name%',
);
```

- *YAML*

```
# messages.fr.yml
'Hello %name%': Hello %name%
```

Nota: Os espaços reservados podem suportar qualquer outro forma já que a mensagem inteira é reconstruída usando a função PHP strtr function. Entretanto, a notação `%var%` é requerida quando traduzir em templates Twig, e é no geral uma convenção sensata a seguit.

Como podemos ver, criar uma tradução é um processo de dois passos:

1. Abstrair a mensagem que precisa ser traduzida por processamento através do `Translator`.
2. Criar uma tradução para a mensagem em cada localidade que você escolha dar suporte.

O segundo passo é feito mediante criar catálogos de mensagem que definem as traduções para qualquer número de localidades diferentes.

Catálogo de Mensagens

Quando uma mensagem é traduzida, Symfony2 compila um catálogo de mensagem para a localidade do usuário e investiga por uma tradução da mensagem. Um catálogo de mensagens é como um dicionário de traduções para uma localidade específica. Por exemplo, o catálogo para a localidade‘‘fr_FR‘‘ poderia conter a seguinte tradução:

Symfony2 is Great => J’aime Symfony2

É responsabilidade do desenvolvedor (ou tradutor) de uma aplicação internacionalizada criar essas traduções. Traduções são armazenadas no sistema de arquivos e descoberta pelo Symfony, graças a algumas convenções.

Dica: Cada vez que você criar um *novo* translation resource (ou instalar um pacote que inclua o translation resource), tenha certeza de limpar o cache então aquele Symfony poderá detectar o novo translation resource:

```
php app/console cache:clear
```

Localização de Traduções e Convenções de Nomenclatura

O Symfony2 procura por arquivos de mensagem (em outras palavras, traduções) nas seguintes localizações:

- o diretório `<kernel root directory>/Resources/translations`;
- o diretório `<kernel root directory>/Resources/<bundle name>/translations`;
- o diretório `Resources/translations/` do bundle.

Os locais são apresentados com a prioridade mais alta em primeiro lugar. Isso significa que você pode sobrescrever as mensagens de tradução de um bundle em qualquer um dos 2 diretórios no topo.

O mecanismo de substituição funciona em um nível chave: apenas as chaves sobrescritas precisam ser listadas em um arquivo de mensagem de maior prioridade. Quando a chave não é encontrada em um arquivo de mensagem, o tradutor automaticamente alternará para os arquivos de mensagem menos prioritários.

O nome de arquivo das traduções é também importante, já que Symfony2 usa uma convenção para determinar detalhes sobre as traduções. Cada arquivo de messagem deve ser nomeado de acordo com o seguinte padrão: `domínio.localidade.carregador`:

- **domínio**: Uma forma opcional de organizar mensagens em grupos (ex: `admin`, `navigation` ou o padrão `messages`) - veja *Usando Domínios de Mensagem*;
- **localidade**: A localidade para a qual a tradução é feita (ex: `en_GB`, `en`, etc);
- **carregador**: Como Symfony2 deveria carregar e analisar o arquivo (ex: `xliff`, `php` or `yml`).

O carregador poder ser o nome de qualquer carregador registrado. Por padrão, Symfony providencia os seguintes carregadores:

- `xliff`: arquivo XLIFF;
- `php`: arquivo PHP;
- `yml`: arquivo YAML.

A escolha de qual carregador é inteiramente tua e é uma questão de gosto.

Nota: Você também pode armazenar traduções em um banco de dados, ou outro armazenamento ao providenciar uma classe personalizada implementando a interface `LoaderInterface`. Veja `Custom Translation Loaders` abaixo para aprender como registrar carregadores personalizados.

Criando traduções

Cada arquivo consiste de uma série de pares de tradução de id para um dado domínio e locale. A id é o identificador para a tradução individual, e pode ser a mensagem da localidade principal (ex: “Symfony is great”) de sua aplicaçãp ou um identificador único (ex: “symfony2.great” - veja a barra lateral abaixo):

- *XML*

```
<!-- src/Acme/DemoBundle/Resources/translations/messages.fr.xliff -->
<?xml version="1.0"?>
<xliff version="1.2" xmlns="urn:oasis:names:tc:xliff:document:1.2">
    <file source-language="en" datatype="plaintext" original="file.ext">
```

```
            <body>
                <trans-unit id="1">
                    <source>Symfony2 is great</source>
                    <target>J'aime Symfony2</target>
                </trans-unit>
                <trans-unit id="2">
                    <source>symfony2.great</source>
                    <target>J'aime Symfony2</target>
                </trans-unit>
            </body>
        </file>
    </xliff>
```

- *PHP*

```
    // src/Acme/DemoBundle/Resources/translations/messages.fr.php
    return array(
        'Symfony2 is great' => 'J\'aime Symfony2',
        'symfony2.great'    => 'J\'aime Symfony2',
    );
```

- *YAML*

```
    # src/Acme/DemoBundle/Resources/translations/messages.fr.yml
    Symfony2 is great: J'aime Symfony2
    symfony2.great:    J'aime Symfony2
```

Symfony2 irá descobrir esses arquivos e usá-los quando ou traduzir "Symfony2 is great" ou "symfony2.great" no localidade do idioma Francês (ex: `fr_FR` ou `fr_BE`).

Usando Mensagens Reais ou Palavras-Chave

Esse exemplo ilustra duas diferentes filosofias quando criar mensagens para serem traduzidas:

```
$t = $translator->trans('Symfony2 is great');

$t = $translator->trans('symfony2.great');
```

No primeiro método, mensagens são escritas no idioma do localidade padrão (Inglês neste caso). Aquela mensagem é então usada como "id" quando criar traduções.
No segundo método, mensagens são realmente "palavras-chave" que contém a idéia da mensagem. A mensagem de palavras-chave é então usada como o "id" de qualquer tradução. Neste caso, traduções devem ser feitas para o locale padrão (em outras palavras, traduzir `symfony2.great` para `Symfony2 is great`).
O segundo método é prático porque a chave da mensagem não precisará ser mudada em cada arquivo de tradução se decidirmos que a mensagem realmente deveria ler "Symfony2 is really great" no localidade padrão.
A escolha de qual método usar é inteiramente sua, mas o formato de "palavra-chave" é frequentemente recomendada.
Adicionalmente, os formatos de arquivo `php` e `yaml` suportam ids encaixadas para que você evite repetições se você usar palavras-chave ao invés do texto real para suas ids:

- *YAML*

```
symfony2:
    is:
        great: Symfony2 is great
        amazing: Symfony2 is amazing
    has:
        bundles: Symfony2 has bundles
user:
    login: Login
```

- *PHP*

```
return array(
    'symfony2' => array(
        'is' => array(
            'great' => 'Symfony2 is great',
            'amazing' => 'Symfony2 is amazing',
        ),
        'has' => array(
            'bundles' => 'Symfony2 has bundles',
        ),
    ),
    'user' => array(
        'login' => 'Login',
    ),
);
```

Os níveis múltiplos são achatados em id unitária / pares de tradução ao adicionar um ponto (.) entre cada nível, portanto os exemplos acima são equivalentes ao seguinte:

- *YAML*

```
symfony2.is.great: Symfony2 is great
symfony2.is.amazing: Symfony2 is amazing
symfony2.has.bundles: Symfony2 has bundles
user.login: Login
```

- *PHP*

```
return array(
    'symfony2.is.great' => 'Symfony2 is great',
    'symfony2.is.amazing' => 'Symfony2 is amazing',
    'symfony2.has.bundles' => 'Symfony2 has bundles',
    'user.login' => 'Login',
);
```

Usando Domínios de mensagem

Como nós vimos, arquivos de mensagem são organizados em diferentes localidades que eles traduzem. Os arquivos de mensagem podem também ser organizados além de "domínios". O domínio padrão é `messages`. Por exemplo, suponha que, para organização, traduções foram divididas em três domínios diferentes: `messages`, `admin` e `navigation`. A tradução Francesa teria os seguintes arquivos de mensagem:

- `messages.fr.xliff`
- `admin.fr.xliff`
- `navigation.fr.xliff`

Quando traduzir strings que não estão no domínio padrão (`messages`), você deve especificar o domínio como terceiro argumento de `trans()`:

```
$this->get('translator')->trans('Symfony2 is great', array(), 'admin');
```

Symfony2 irá pesquisar pela mensagem no domínio''admin'' da localidade do usuário.

Tratando a localidade do Usuário

A localidade do usuário atual é armazenada no pedido e é acessível através do objeto ''request':

```
// access the reqest object in a standard controller
$request = $this->getRequest();

$locale = $request->getLocale();

$request->setLocale('en_US');
```

Também é possível armazenar a localidade na sessão em vez do pedido. Se você fizer isso, cada pedido posterior terá esta localidade.

```
$this->get('session')->set('_locale', 'en_US');
```

Veja a seção *A localidade e a URL* abaixo sobre como setar a localidade através de roteamento.

Localidade padrão e alternativa

Se a localidade não foi fixada explicitamente na sessão , o parâmetro de configuração `fallback_locale` será usada pelo `Translator`. O parâmetro é padronizado para `en` (veja *Configuração*).

Alternativamente, você pode garantir que uma localidade é definida em cada pedido do usuário definindo um `default_locale` para o framework:

- *YAML*

```
# app/config/config.yml
framework:
    default_locale: en
```

- *XML*

```
<!-- app/config/config.xml -->
<framework:config>
    <framework:default-locale>en</framework:default-locale>
</framework:config>
```

- *PHP*

```
// app/config/config.php
$container->loadFromExtension('framework', array(
    'default_locale' => 'en',
));
```

Novo na versão 2.1: O parâmetro `default_locale` foi definido debaixo da chave session originalmente, entretanto, com o 2.1 isto foi movido. Foi movido porque a localidade agora é definida no pedido ao invés da sessão.

A localidade e a URL

Uma vez que você pode armazenar a localidade do usuário na sessão, pode ser tentador usar a mesma URL para mostrar o recurso em muitos idiomas diferentes baseados na localidade do usuário.Por exemplo, `http://www.example.com/contact` poderia mostrar conteúdo em Inglês para um usuário e Francês para outro usuário. Infelizmente, isso viola uma regra fundamental da Web: que um URL particular retorne o mesmo recurso independente do usuário. Para complicar ainda o problema, qual versão do conteúdo deveria ser indexado pelas ferramentas de pesquisa ?

Uma melhor política é incluir a localidade na URL. Isso é totalmente suportado pelo sistema de roteamnto usando o parâmetro especial `_locale`:

- *YAML*

```
contact:
    pattern:   /{_locale}/contact
    defaults:  { _controller: AcmeDemoBundle:Contact:index, _locale: en }
    requirements:
        _locale: en|fr|de
```

- *XML*

```
<route id="contact" pattern="/{_locale}/contact">
    <default key="_controller">AcmeDemoBundle:Contact:index</default>
    <default key="_locale">en</default>
    <requirement key="_locale">en|fr|de</requirement>
</route>
```

- *PHP*

```
use Symfony\Component\Routing\RouteCollection;
use Symfony\Component\Routing\Route;

$collection = new RouteCollection();
$collection->add('contact', new Route('/{_locale}/contact', array(
    '_controller' => 'AcmeDemoBundle:Contact:index',
    '_locale'     => 'en',
), array(
    '_locale'     => 'en|fr|de'
)));

return $collection;
```

Quando usar o parâmetro especial *_locale* numa rota, a localidade encontrada será *automaticamente estabelecida na sessão do usuário*. Em outras palavras, se um usuário visita a URI `/fr/contact`, a localidade‘‘fr‘‘ será automaticamente estabelecida como a localidade para a sessão do usuário.

Você pode agora usar a localidade do usuário para criar rotas para outras páginas traduzidas na sua aplicação.

Pluralização

Pluralização de mensagem é um tópico difícil já que as regras podem ser bem complexas. Por convenção, aqui está a representação matemática das regrad de pluralização Russa:

```
(($number % 10 == 1) && ($number % 100 != 11)) ? 0 : ((($number % 10 >= 2) && ($number % 10 <= 4) &&
```

Como você viu, em Russo, você pode ter três formas diferentes de plural, cada uma com index de 0, 1 ou 2. Para cada forma, o plural é diferente, e então a tradução também é diferente.

Quando uma tradução tem formas diferentes devido à pluralização, você pode providenciar todas as formas como string separadas por barra vertical (`|`):

```
'There is one apple|There are %count% apples'
```

Para traduzir mensagens pluralizadas, use o método: `transChoice()`

```
$t = $this->get('translator')->transChoice(
    'There is one apple|There are %count% apples',
    10,
    array('%count%' => 10)
);
```

O segundo argumento (`10` neste exemplo), é o *número* de objetos sendo descritos e é usado para determinar qual tradução usar e também para preencher o espaço reservado `%count%`.

Baseado em certo número, o tradutor escolhe a forma correta do plural. Em Inglês, muitas palavras tem uma forma singular quando existe exatamente um objeto e uma forma no plural para todos os outros números (0, 2, 3...). Então, se `count` é `1`, o tradutor usará a primeira string (`There is one apple`) como tradução. Senão irá usar `There are %count% apples`.

Aqui está a tradução Francesa:

```
'Il y a %count% pomme|Il y a %count% pommes'
```

Mesmo se a string parecer similar (é feita de duas substrings separadas por barra vertical), as regras Francesas são diferentes: a primeira forma (sem plural) é usada quando `count` is `0` or `1`. Então, o tradutor irá automaticamente usar a primeira string (`Il y a %count% pomme`) quando `count` é `0` ou `1`.

Cada localidade tem sua própria lista de regras, com algumas tendo tanto quanto seis formas diferentes de plural com regras complexas por trás de quais números de mapas de quais formas no plural. As regras são bem simples para Inglês e Francês, mas para Russo, você poderia querer um palpite para conhecer quais regras combinam com qual string. Para ajudar tradutores, você pode opcionalmente "atribuir uma tag" a cada string:

```
'one: There is one apple|some: There are %count% apples'

'none_or_one: Il y a %count% pomme|some: Il y a %count% pommes'
```

As tags são realmente as únicas pistas para tradutores e não afetam a lógica usada para determinar qual forma plural usar. As tags podem ser qualquer string descritiva que termine com dois pontos (`:`). As tags traduzidas também não são necessariamente a mesma que as da mensagem original.

Pluralização de Intervalo Explícito

A maneira mais fácil de pluralizar uma mensagem é deixar o Symfony2 usar lógica interna para escolher qual string usar, baseando em um número fornecido. Às vezes, você irá precisar de mais controle ou querer uma tradução diferente para casos específicos (para `0`, ou quando o contador é negativo, por exemplo). Para certos casos, você pode usar intervalos matemáticos explícitos:

```
'{0} There are no apples|{1} There is one apple|]1,19] There are %count% apples|[20,Inf] There are ma
```

Os intervalos seguem a notação ISO 31-11. A string acima especifica quatro intervalos diferentes: exatamente `0`, exatamente `1`, `2-19`, e `20` e mais altos.

Você também pode misturar regras matemáticas explícitas e regras padrão. Nesse caso, se o contador não combinar com um intervalo específico, as regras padrão, terão efeito após remover as regras explícitas:

```
'{0} There are no apples|[20,Inf] There are many apples|There is one apple|a_few: There are %count% a
```

Por exemplo, para `1` maçã, a regra padrão `There is one apple` será usada. Para `2-19` apples, a segunda regra padrão `There are %count% apples` será selecionada.

Uma classe `Interval` pode representar um conjunto finito de números:

```
{1,2,3,4}
```

Ou números entre dois outros números:

```
[1, +Inf[
]-1,2[
```

O delimitador esquerdo pode ser `[` (inclusivo) or `]` (exclusivo). O delimitador direito pode ser `[` (exclusivo) or `]` (inclusivo). Além de números, você pode usar `-Inf` e `+Inf` para infinito.

Traduções em Templates

A maior parte do tempo, traduções ocorrem em templates. Symfony2 providencia suporte nativo para ambos os templates PHP e Twig.

Templates Twig

Symfony2 providencia tags Twig especializadas (`trans` e `transchoice`) para ajudar com tradução de mensagem de *blocos estáticos de texto*:

```
{% trans %}Hello %name%{% endtrans %}

{% transchoice count %}
    {0} There are no apples|{1} There is one apple|]1,Inf] There are %count% apples
{% endtranschoice %}
```

A tag `transchoice` automaticamente obtém a variável `%count%` do contexto atual e a passa para o tradutor. Esse mecanismo só funciona quando você usa um espaço reservado seguindo o padrão `%var%`.

Dica: Se você precisa usar o caractere de percentual (`%`) em uma string, escape dela ao dobrá-la: `{% trans %}Percent: %percent%%%{% endtrans %}`

Você também pode especificar o domínio da mensagem e passar algumas variáveis adicionais:

```
{% trans with {'%name%': 'Fabien'} from "app" %}Hello %name%{% endtrans %}

{% trans with {'%name%': 'Fabien'} from "app" into "fr" %}Hello %name%{% endtrans %}

{% transchoice count with {'%name%': 'Fabien'} from "app" %}
    {0} There is no apples|{1} There is one apple|]1,Inf] There are %count% apples
{% endtranschoice %}
```

Os filtros `trans` e `transchoice` podem ser usados para traduzir *textos de variáveis* e expressões complexas:

```
{{ message | trans }}

{{ message | transchoice(5) }}

{{ message | trans({'%name%': 'Fabien'}, "app") }}

{{ message | transchoice(5, {'%name%': 'Fabien'}, 'app') }}
```

Dica: Usando as tags de tradução ou filtros que tenham o mesmo efeito, mas com uma diferença sutil: saída para escape automático só é aplicada para variáveis traduzidas por utilização de filtro. Em outras palavras, se você precisar estar certo que sua variável traduzida *não* é uma saída para escape, você precisa aplicar o filtro bruto após o filtro de tradução.

```
{# text translated between tags is never escaped #}
{% trans %}
    <h3>foo</h3>
{% endtrans %}

{% set message = '<h3>foo</h3>' %}

{# a variable translated via a filter is escaped by default #}
{{ message | trans | raw }}

{# but static strings are never escaped #}
{{ '<h3>foo</h3>' | trans }}
```

Novo na versão 2.1: Agora você pode definir o domínio de tradução para um template Twig inteiro com uma única tag:

```
{% trans_default_domain "app" %}
```

Note que isso somente influencia o template atual, e não qualquer template "incluído" (para evitar efeitos colaterais).

Templates PHP

O serviço tradutor é acessível em templates PHP através do helper `translator`:

```
<?php echo $view['translator']->trans('Symfony2 is great') ?>

<?php echo $view['translator']->transChoice(
    '{0} There is no apples|{1} There is one apple|]1,Inf[ There are %count% apples',
    10,
    array('%count%' => 10)
) ?>
```

Forçando a Localidade Tradutora

Quando traduzir a mensagem, Symfony2 usa a localidade do pedido atual ou a localidade `alternativa` se necessária. Você também pode especificar manualmente a localidade a usar para a tradução:

```
$this->get('translator')->trans(
    'Symfony2 is great',
```

```
    array(),
    'messages',
    'fr_FR',
);

$this->get('translator')->trans(
    '{0} There are no apples|{1} There is one apple|]1,Inf[ There are %count% apples',
    10,
    array('%count%' => 10),
    'messages',
    'fr_FR',
);
```

Traduzindo Conteúdo de Banco de Dados

Quando a tradução do conteúdo do banco de dados deveria ser manipulada pelo Doctrine através do Translatable Extension. Para mais informações, veja a documentação para aquela biblioteca.

Sumário

Com o componente Translation do Symfony2, criar uma aplicação internacionalizada não precisa mais ser um processo dolorido e desgastante para somente uns passos básicos:

- Mensagens abstratas na sua aplicação ao englobar cada uma ou com métodos `trans()` ou `transChoice()`;
- Traduza cada mensagem em localidades múltiplas ao criar arquivos de tradução de mensagem. Symfony2 descobre e processa cada arquivo porque o nome dele segue uma convenção específica;
- Gerenciar a localidade do usuário, que é armazenada no pedido, mas também pode ser definida na sessão do usuário.

2.1.14 Container de Serviço

Uma aplicação PHP moderna é cheia de objetos. Um objeto pode facilitar a entrega de mensagens de e-mail enquanto outro pode permitir persistir as informações em um banco de dados. Em sua aplicação, você pode criar um objeto que gerencia seu estoque de produtos, ou outro objeto que processa dados de uma API de terceiros. O ponto é que uma aplicação moderna faz muitas coisas e é organizada em muitos objetos que lidam com cada tarefa.

Neste capítulo, vamos falar sobre um objeto PHP especial no Symfony2 que ajuda você a instanciar, organizar e recuperar os muitos objetos da sua aplicação. Esse objeto, chamado de container de serviço, lhe permitirá padronizar e centralizar a forma como os objetos são construídos em sua aplicação. O container facilita a sua vida, é super rápido e enfatiza uma arquitetura que promove código reutilizável e desacoplado. E, como todas as classes principais do Symfony2 usam o container, você vai aprender como estender, configurar e usar qualquer objeto no Symfony2. Em grande parte, o container de serviço é o maior contribuinte à velocidade e extensibilidade do Symfony2.

Finalmente, configurar e usar o container de serviço é fácil. Ao final deste capítulo, você estará confortável criando os seus próprios objetos através do container e personalizando objetos a partir de qualquer bundle de terceiros. Você vai começar a escrever código que é mais reutilizável, testável e desacoplado, simplesmente porque o container de serviço torna fácil escrever bom código.

O que é um Serviço?

Simplificando, um Serviço é qualquer objeto PHP que realiza algum tipo de tarefa "global". É um nome genérico proposital, usado em ciência da computação, para descrever um objeto que é criado para uma finalidade específica

(por exemplo, entregar e-mails). Cada serviço é usado em qualquer parte da sua aplicação sempre que precisar da funcionalidade específica que ele fornece. Você não precisa fazer nada de especial para construir um serviço: basta escrever uma classe PHP com algum código que realiza uma tarefa específica. Parabéns, você acabou de criar um serviço!

Nota: Como regra geral, um objeto PHP é um serviço se ele é usado globalmente em sua aplicação. Um único serviço `Mailer` é utilizado globalmente para enviar mensagens de e-mail, enquanto os muitos objetos `Message` que ele entrega *não* são serviços. Do mesmo modo, um objeto `Product` não é um serviço, mas um objeto que persiste os objetos `Product` para um banco de dados *é* um serviço.

Então, porque ele é especial? A vantagem de pensar em "serviços" é que você começa a pensar em separar cada pedaço de funcionalidade de sua aplicação em uma série de serviços. Uma vez que cada serviço realiza apenas um trabalho, você pode facilmente acessar cada serviço e usar a sua funcionalidade, sempre que você precisar. Cada serviço pode também ser mais facilmente testado e configurado já que ele está separado das outras funcionalidades em sua aplicação. Esta idéia é chamada de arquitetura orientada à serviços e não é exclusiva do Symfony2 ou até mesmo do PHP. Estruturar a sua aplicação em torno de um conjunto de classes de serviços independentes é uma das melhores práticas de orientação à objeto bem conhecida e confiável. Essas habilidades são a chave para ser um bom desenvolvedor em praticamente qualquer linguagem.

O que é um Service Container?

Um Container de Serviço (ou *container de injeção de dependência*) é simplesmente um objeto PHP que gerencia a instanciação de serviços (ex. objetos). Por exemplo, imagine que temos uma classe PHP simples que envia mensagens de e-mail. Sem um container de serviço, precisamos criar manualmente o objeto sempre que precisarmos dele:

```
use Acme\HelloBundle\Mailer;

$mailer = new Mailer('sendmail');
$mailer->send('ryan@foobar.net', ... );
```

Isso é bastante fácil. A classe imaginária `Mailer` nos permite configurar o método utilizado para entregar as mensagens de e-mail (por exemplo: `sendmail`, `smtp`, etc). Mas, e se quiséssemos usar o serviço de mailer em outro lugar? Nós certamente não desejamos repetir a configuração do mailer *sempre* que nós precisamos usar o objeto `Mailer`. E se precisarmos mudar o `transport` de `sendmail` para `smtp` em toda a aplicação? Nós precisaremos localizar cada lugar em que adicionamos um serviço `Mailer` e alterá-lo.

Criando/Configurando Serviços no Container

Uma resposta melhor é deixar o container de serviço criar o objeto `Mailer` para você. Para que isso funcione, é preciso *ensinar* o container como criar o serviço `Mailer`. Isto é feito através de configuração, que pode ser especificada em YAML, XML ou PHP:

- *YAML*

```
# app/config/config.yml
services:
    my_mailer:
        class:        Acme\HelloBundle\Mailer
        arguments:    [sendmail]
```

- *XML*

```
<!-- app/config/config.xml -->
<services>
```

```
        <service id="my_mailer" class="Acme\HelloBundle\Mailer">
            <argument>sendmail</argument>
        </service>
    </services>
```

- *PHP*

```
    // app/config/config.php
    use Symfony\Component\DependencyInjection\Definition;

    $container->setDefinition('my_mailer', new Definition(
        'Acme\HelloBundle\Mailer',
        array('sendmail')
    ));
```

Nota: Quando o Symfony2 é inicializado, ele constrói o container de serviço usando a configuração da aplicação (por padrão (`app/config/config.yml`). O arquivo exato que é carregado é ditado pelo método `AppKernel::registerContainerConfiguration()` , que carrega um arquivo de configuração do ambiente específico (por exemplo `config_dev.yml` para o ambiente `dev` ou `config_prod.yml` para o `prod`).

Uma instância do objeto `Acme\HelloBundle\Mailer` está agora disponível através do container de serviço. O container está disponível em qualquer controlador tradicional do Symfony2 onde você pode acessar os serviços do container através do método de atalho `get()`:

```
class HelloController extends Controller
{
    // ...

    public function sendEmailAction()
    {
        // ...
        $mailer = $this->get('my_mailer');
        $mailer->send('ryan@foobar.net', ... );
    }
}
```

Quando requisitamos o serviço `my_mailer` a partir do container, o container constrói o objeto e o retorna. Esta é outra vantagem importante do uso do container de serviço. Ou seja, um serviço *nunca* é construído até que ele seja necessário. Se você definir um serviço e nunca usá-lo em um pedido, o serviço nunca será criado. Isso economiza memória e aumenta a velocidade de sua aplicação. Isto também significa que não há nenhuma perda ou apenas uma perda insignificante de desempenho ao definir muitos serviços. Serviços que nunca são usados, nunca são construídos.

Como um bônus adicional, o serviço `Mailer` é criado apenas uma vez e a mesma instância é retornada cada vez que você requisitar o serviço. Isso é quase sempre o comportamento que você precisa (é mais flexível e poderoso), mas vamos aprender, mais tarde, como você pode configurar um serviço que possui várias instâncias.

Parâmetros do Serviço

A criação de novos serviços (objetos, por exemplo) através do container é bastante simples. Os parâmetros tornam a definição dos serviços mais organizada e flexível:

- *YAML*

```
    # app/config/config.yml
    parameters:
        my_mailer.class:      Acme\HelloBundle\Mailer
```

```
        my_mailer.transport:  sendmail

    services:
        my_mailer:
            class:        %my_mailer.class%
            arguments:    [%my_mailer.transport%]
```

- *XML*

```
    <!-- app/config/config.xml -->
    <parameters>
        <parameter key="my_mailer.class">Acme\HelloBundle\Mailer</parameter>
        <parameter key="my_mailer.transport">sendmail</parameter>
    </parameters>

    <services>
        <service id="my_mailer" class="%my_mailer.class%">
            <argument>%my_mailer.transport%</argument>
        </service>
    </services>
```

- *PHP*

```
    // app/config/config.php
    use Symfony\Component\DependencyInjection\Definition;

    $container->setParameter('my_mailer.class', 'Acme\HelloBundle\Mailer');
    $container->setParameter('my_mailer.transport', 'sendmail');

    $container->setDefinition('my_mailer', new Definition(
        '%my_mailer.class%',
        array('%my_mailer.transport%')
    ));
```

O resultado final é exatamente o mesmo de antes - a diferença está apenas em *como* definimos o serviço. Quando as strings `my_mailer.class` e `my_mailer.transport` estão envolvidas por sinais de porcentagem (`%`), o container sabe que deve procurar por parâmetros com esses nomes. Quando o container é construído, ele procura o valor de cada parâmetro e utiliza-o na definição do serviço.

Nota: O sinal de porcentagem dentro de um parâmetro ou argumento, como parte da string, deve ser escapado com um outro sinal de porcentagem:

```
<argument type="string">http://symfony.com/?foo=%%s&bar=%%d</argument>
```

A finalidade dos parâmetros é alimentar informação para os serviços. Naturalmente, não há nada de errado em definir o serviço sem o uso de quaisquer parâmetros. Os parâmetros, no entanto, possuem várias vantagens:

- separação e organização de todas as "opções" dos serviços sob uma única chave `parameters`;
- os valores do parâmetro podem ser usados em múltiplas definições de serviços;
- ao criar um serviço em um bundle (vamos mostrar isso em breve), o uso de parâmetros permite que o serviço seja facilmente customizado em sua aplicação.

A escolha de usar ou não os parâmetros é com você. Bundles de terceiros de alta qualidade *sempre* utilizarão parâmetros, pois, eles tornam mais configurável o serviço armazenado no container. Para os serviços em sua aplicação, entretanto, você pode não precisar da flexibilidade dos parâmetros.

Parâmetros Array

Os parâmetros não precisam ser strings, eles também podem ser arrays. Para o formato XML , você precisará usar o atributo type=”collection” em todos os parâmetros que são arrays.

- *YAML*

```yaml
# app/config/config.yml
parameters:
    my_mailer.gateways:
        - mail1
        - mail2
        - mail3
    my_multilang.language_fallback:
        en:
            - en
            - fr
        fr:
            - fr
            - en
```

- *XML*

```xml
<!-- app/config/config.xml -->
<parameters>
    <parameter key="my_mailer.gateways" type="collection">
        <parameter>mail1</parameter>
        <parameter>mail2</parameter>
        <parameter>mail3</parameter>
    </parameter>
    <parameter key="my_multilang.language_fallback" type="collection">
        <parameter key="en" type="collection">
            <parameter>en</parameter>
            <parameter>fr</parameter>
        </parameter>
        <parameter key="fr" type="collection">
            <parameter>fr</parameter>
            <parameter>en</parameter>
        </parameter>
    </parameter>
</parameters>
```

- *PHP*

```php
// app/config/config.php
use Symfony\Component\DependencyInjection\Definition;

$container->setParameter('my_mailer.gateways', array('mail1', 'mail2', 'mail3'));
$container->setParameter('my_multilang.language_fallback',
                         array('en' => array('en', 'fr'),
                               'fr' => array('fr', 'en'),
                         ));
```

Importando outros Recursos de Configuração do Container

Dica: Nesta seção, vamos referenciar os arquivos de configuração de serviço como *recursos*. Isto é para destacar o fato de que, enquanto a maioria dos recursos de configuração será em arquivos (por exemplo, YAML, XML, PHP), o

Symfony2 é tão flexível que a configuração pode ser carregada de qualquer lugar (por exemplo, de um banco de dados ou até mesmo através de um web service externo).

O service container é construído usando um único recurso de configuração (por padrão `app/config/config.yml`). Todas as outras configurações de serviço (incluindo o núcleo do Symfony2 e configurações de bundles de terceiros) devem ser importadas dentro desse arquivo de uma forma ou de outra. Isso lhe dá absoluta flexibilidade sobre os serviços em sua aplicação.

Configurações de serviços externos podem ser importadas de duas maneiras distintas. Primeiro, vamos falar sobre o método que você vai usar mais frequentemente na sua aplicação: a diretiva `imports`. Na seção seguinte, vamos falar sobre o segundo método, que é mais flexível e preferido para a importação de configuração de serviços dos bundles de terceiros.

Importando configuração com `imports`

Até agora, adicionamos a nossa definição do container de serviço `my_mailer` diretamente no arquivo de configuração da aplicação (por exemplo: `app/config/config.yml`). Naturalmente, já que a própria classe `Mailer` reside dentro do `AcmeHelloBundle`, faz mais sentido colocar a definição do container `my_mailer` dentro do bundle também.

Primeiro, mova a definição do container `my_mailer` dentro de um novo arquivo container de recurso dentro do `AcmeHelloBundle`. Se os diretórios `Resources` ou `Resources/config` não existirem, adicione eles.

- *YAML*

```
# src/Acme/HelloBundle/Resources/config/services.yml
parameters:
    my_mailer.class:      Acme\HelloBundle\Mailer
    my_mailer.transport:  sendmail

services:
    my_mailer:
        class:        %my_mailer.class%
        arguments:    [%my_mailer.transport%]
```

- *XML*

```
<!-- src/Acme/HelloBundle/Resources/config/services.xml -->
<parameters>
    <parameter key="my_mailer.class">Acme\HelloBundle\Mailer</parameter>
    <parameter key="my_mailer.transport">sendmail</parameter>
</parameters>

<services>
    <service id="my_mailer" class="%my_mailer.class%">
        <argument>%my_mailer.transport%</argument>
    </service>
</services>
```

- *PHP*

```
// src/Acme/HelloBundle/Resources/config/services.php
use Symfony\Component\DependencyInjection\Definition;

$container->setParameter('my_mailer.class', 'Acme\HelloBundle\Mailer');
$container->setParameter('my_mailer.transport', 'sendmail');

$container->setDefinition('my_mailer', new Definition(
```

```
        '%my_mailer.class%',
        array('%my_mailer.transport%')
    ));
```

A definição em si não mudou, apenas a sua localização. Claro que o container de serviço não sabe sobre o novo arquivo de recurso. Felizmente, nós podemos facilmente importar o arquivo de recurso usando a chave `imports` na configuração da aplicação.

- *YAML*

```
# app/config/config.yml
imports:
    - { resource: @AcmeHelloBundle/Resources/config/services.yml }
```

- *XML*

```
<!-- app/config/config.xml -->
<imports>
    <import resource="@AcmeHelloBundle/Resources/config/services.xml"/>
</imports>
```

- *PHP*

```
// app/config/config.php
$this->import('@AcmeHelloBundle/Resources/config/services.php');
```

A diretiva `imports` permite à sua aplicação incluir os recursos de configuração do container de serviço de qualquer outra localização (mais comumente, de bundles). A localização do `resource`, para arquivos, é o caminho absoluto para o arquivo de recurso. A sintaxe especial `@AcmeHello` resolve o caminho do diretório do bundle `AcmeHelloBundle`. Isso ajuda você a especificar o caminho para o recurso sem preocupar-se mais tarde caso mover o `AcmeHelloBundle` para um diretório diferente.

Importando Configuração através de Extensões do Container

Ao desenvolver no Symfony2, você vai mais comumente usar a diretiva `imports` para importar a configuração do container dos bundles que você criou especificamente para a sua aplicação. As configurações de container de bundles de terceiros, incluindo os serviços do núcleo do Symfony2, são geralmente carregadas usando um outro método que é mais flexível e fácil de configurar em sua aplicação.

Veja como ele funciona. Internamente, cada bundle define os seus serviços de forma semelhante a que vimos até agora. Ou seja, um bundle utiliza um ou mais arquivos de configurações de recurso (normalmente XML) para especificar os parâmetros e serviços para aquele bundle. No entanto, em vez de importar cada um desses recursos diretamente a partir da sua configuração da aplicação usando a diretiva `imports`, você pode simplesmente invocar uma *extensão do container de serviço* dentro do bundle que faz o trabalho para você. Uma extensão do container de serviço é uma classe PHP criada pelo autor do bundle para realizar duas coisas:

- importar todos os recursos de container de serviço necessários para configurar os serviços para o bundle;
- fornecer configuração simples e semântica para que o bundle possa ser configurado sem interagir com os parâmetros da configuração do container de serviço.

Em outras palavras, uma extensão do container de serviço configura os serviços para um bundle em seu nome. E, como veremos em breve, a extensão fornece uma interface sensível e de alto nível para configurar o bundle.

Considere o `FrameworkBundle` - o bundle do núcleo do framework Symfony2 - como um exemplo. A presença do código a seguir na configuração da sua aplicação invoca a extensão do container de serviço dentro do `FrameworkBundle`:

- *YAML*

```yaml
# app/config/config.yml
framework:
    secret:          xxxxxxxxxx
    form:            true
    csrf_protection: true
    router:        { resource: "%kernel.root_dir%/config/routing.yml" }
    # ...
```

- *XML*

```xml
<!-- app/config/config.xml -->
<framework:config secret="xxxxxxxxxx">
    <framework:form />
    <framework:csrf-protection />
    <framework:router resource="%kernel.root_dir%/config/routing.xml" />
    <!-- ... -->
</framework:config>
```

- *PHP*

```php
// app/config/config.php
$container->loadFromExtension('framework', array(
    'secret'          => 'xxxxxxxxxx',
    'form'            => array(),
    'csrf-protection' => array(),
    'router'          => array('resource' => '%kernel.root_dir%/config/routing.php'),
    // ...
));
```

Quando é realizado o parse da configuração, o container procura uma extensão que pode lidar com a diretiva de configuração `framework`. A extensão em questão, que reside no `FrameworkBundle`, é chamada e a configuração do serviço para o `FrameworkBundle` é carregada. Se você remover totalmente a chave `framework` de seu arquivo de configuração da aplicação, os serviços do núcleo do Symfony2 não serão carregados. O ponto é que você está no controle: o framework Symfony2 não contém qualquer tipo de magia ou realiza quaisquer ações que você não tenha controle.

Claro que você pode fazer muito mais do que simplesmente "ativar" a extensão do container de serviço do `FrameworkBundle`. Cada extensão permite que você facilmente personalize o bundle, sem se preocupar com a forma como os serviços internos são definidos.

Neste caso, a extensão permite que você personalize as configurações `error_handler`, `csrf_protection`, `router` e muito mais. Internamente, o `FrameworkBundle` usa as opções especificadas aqui para definir e configurar os serviços específicos para ele. O bundle se encarrega de criar todos os `parameters` e `services` necessários para o container de serviço, permitindo ainda que grande parte da configuração seja facilmente personalizada. Como um bônus adicional, a maioria das extensões do container de serviço também são inteligentes o suficiente para executar a validação - notificando as opções que estão faltando ou que estão com o tipo de dados incorreto.

Ao instalar ou configurar um bundle, consulte a documentação do bundle para verificar como os serviços para o bundle devem ser instalados e configurados. As opções disponíveis para os bundles do núcleo podem ser encontradas no Reference Guide.

Nota: Nativamente, o container de serviço somente reconhece as diretivas `parameters`, `services` e `imports`. Quaisquer outras diretivas são manipuladas pela extensão do container de serviço.

Se você deseja expor configuração amigável em seus próprios bundles, leia o "Como expor uma Configuração Semântica para um Bundle" do cookbook.

Referenciando (Injetando) Serviços

Até agora, nosso serviço original `my_mailer` é simples: ele recebe apenas um argumento em seu construtor, que é facilmente configurável. Como você verá, o poder real do container é percebido quando você precisa criar um serviço que depende de um ou mais outros serviços no container.

Vamos começar com um exemplo. Suponha que temos um novo serviço, `NewsletterManager`, que ajuda a gerenciar a preparação e entrega de uma mensagem de e-mail para uma coleção de endereços. Claro que o serviço `my_mailer` já está muito bom ao entregar mensagens de e-mail, por isso vamos usá-lo dentro do `NewsletterManager` para lidar com a entrega das mensagens. Esta simulação de classe seria parecida com:

```
namespace Acme\HelloBundle\Newsletter;

use Acme\HelloBundle\Mailer;

class NewsletterManager
{
    protected $mailer;

    public function __construct(Mailer $mailer)
    {
        $this->mailer = $mailer;
    }

    // ...
}
```

Sem usar o container de serviço, podemos criar um novo `NewsletterManager` facilmente dentro de um controlador:

```
public function sendNewsletterAction()
{
    $mailer = $this->get('my_mailer');
    $newsletter = new Acme\HelloBundle\Newsletter\NewsletterManager($mailer);
    // ...
}
```

Esta abordagem é boa, mas, e se decidirmos mais tarde que a classe `NewsletterManager` precisa de um segundo ou terceiro argumento? E se decidirmos refatorar nosso código e renomear a classe? Em ambos os casos, você precisa encontrar todos os locais onde o `NewsletterManager` é instanciado e modificá-lo. Certamente, o container de serviço nos dá uma opção muito mais atraente:

- *YAML*

```
    # src/Acme/HelloBundle/Resources/config/services.yml
    parameters:
        # ...
        newsletter_manager.class: Acme\HelloBundle\Newsletter\NewsletterManager

    services:
        my_mailer:
            # ...
        newsletter_manager:
            class:     %newsletter_manager.class%
            arguments: [@my_mailer]
```

- *XML*

```
    <!-- src/Acme/HelloBundle/Resources/config/services.xml -->
    <parameters>
        <!-- ... -->
        <parameter key="newsletter_manager.class">Acme\HelloBundle\Newsletter\NewsletterManager</par
    </parameters>

    <services>
        <service id="my_mailer" ... >
          <!-- ... -->
        </service>
        <service id="newsletter_manager" class="%newsletter_manager.class%">
            <argument type="service" id="my_mailer"/>
        </service>
    </services>
```

- *PHP*

```
    // src/Acme/HelloBundle/Resources/config/services.php
    use Symfony\Component\DependencyInjection\Definition;
    use Symfony\Component\DependencyInjection\Reference;

    // ...
    $container->setParameter('newsletter_manager.class', 'Acme\HelloBundle\Newsletter\NewsletterMana

    $container->setDefinition('my_mailer', ... );
    $container->setDefinition('newsletter_manager', new Definition(
        '%newsletter_manager.class%',
        array(new Reference('my_mailer'))
    ));
```

Em YAML, a sintaxe especial `@my_mailer` diz ao container para procurar por um serviço chamado `my_mailer` e passar esse objeto para o construtor do `NewsletterManager`. Neste caso, entretanto, o serviço especificado `my_mailer` deve existir. Caso ele não existir, será gerada uma exceção. Você pode marcar suas dependências como opcionais - o que será discutido na próxima seção.

O uso de referências é uma ferramenta muito poderosa que lhe permite criar classes de serviços independentes com dependências bem definidas. Neste exemplo, o serviço `newsletter_manager` precisa do serviço `my_mailer` para funcionar. Quando você define essa dependência no container de serviço, o container cuida de todo o trabalho de instanciar os objetos.

Dependências Opcionais: Injeção do Setter

Injetando dependências no construtor dessa maneira é uma excelente forma de assegurar que a dependência estará disponível para o uso. No entanto, se você possuir dependências opcionais para uma classe, a "injeção do setter" pode ser uma opção melhor. Isto significa injetar a dependência usando uma chamada de método ao invés do construtor. A classe ficaria assim:

```
namespace Acme\HelloBundle\Newsletter;

use Acme\HelloBundle\Mailer;

class NewsletterManager
{
    protected $mailer;

    public function setMailer(Mailer $mailer)
    {
```

```
        $this->mailer = $mailer;
    }

    // ...
}
```

Para injetar a dependência pelo método setter somente é necessária uma mudança da sintaxe:

- *YAML*

```
    # src/Acme/HelloBundle/Resources/config/services.yml
    parameters:
        # ...
        newsletter_manager.class: Acme\HelloBundle\Newsletter\NewsletterManager

    services:
        my_mailer:
            # ...
        newsletter_manager:
            class:     %newsletter_manager.class%
            calls:
                - [ setMailer, [ @my_mailer ] ]
```

- *XML*

```
    <!-- src/Acme/HelloBundle/Resources/config/services.xml -->
    <parameters>
        <!-- ... -->
        <parameter key="newsletter_manager.class">Acme\HelloBundle\Newsletter\NewsletterManager</par
    </parameters>

    <services>
        <service id="my_mailer" ... >
          <!-- ... -->
        </service>
        <service id="newsletter_manager" class="%newsletter_manager.class%">
            <call method="setMailer">
                 <argument type="service" id="my_mailer" />
            </call>
        </service>
    </services>
```

- *PHP*

```
    // src/Acme/HelloBundle/Resources/config/services.php
    use Symfony\Component\DependencyInjection\Definition;
    use Symfony\Component\DependencyInjection\Reference;

    // ...
    $container->setParameter('newsletter_manager.class', 'Acme\HelloBundle\Newsletter\NewsletterMana

    $container->setDefinition('my_mailer', ... );
    $container->setDefinition('newsletter_manager', new Definition(
        '%newsletter_manager.class%'
    ))->addMethodCall('setMailer', array(
        new Reference('my_mailer')
    ));
```

Nota: As abordagens apresentadas nesta seção são chamadas de "injeção de construtor" e "injeção de setter". O

container de serviço do Symfony2 também suporta a "injeção de propriedade".

Tornando Opcionais as Referências

Às vezes, um de seus serviços pode ter uma dependência opcional, ou seja, a dependência não é exigida por seu serviço para funcionar corretamente. No exemplo acima, o serviço `my_mailer` *deve* existir, caso contrário, uma exceção será gerada. Ao modificar a definição do serviço `newsletter_manager` , você pode tornar esta referência opcional. O container irá, então, injetá-lo se ele existir e não irá fazer nada caso contrário:

- *YAML*

```
# src/Acme/HelloBundle/Resources/config/services.yml
parameters:
    # ...

services:
    newsletter_manager:
        class:     %newsletter_manager.class%
        arguments: [@?my_mailer]
```

- *XML*

```
<!-- src/Acme/HelloBundle/Resources/config/services.xml -->

<services>
    <service id="my_mailer" ... >
      <!-- ... -->
    </service>
    <service id="newsletter_manager" class="%newsletter_manager.class%">
        <argument type="service" id="my_mailer" on-invalid="ignore" />
    </service>
</services>
```

- *PHP*

```
// src/Acme/HelloBundle/Resources/config/services.php
use Symfony\Component\DependencyInjection\Definition;
use Symfony\Component\DependencyInjection\Reference;
use Symfony\Component\DependencyInjection\ContainerInterface;

// ...
$container->setParameter('newsletter_manager.class', 'Acme\HelloBundle\Newsletter\NewsletterMana

$container->setDefinition('my_mailer', ... );
$container->setDefinition('newsletter_manager', new Definition(
    '%newsletter_manager.class%',
    array(new Reference('my_mailer', ContainerInterface::IGNORE_ON_INVALID_REFERENCE))
));
```

Em YAML, a sintaxe especial `@?` diz ao container de serviço que a dependência é opcional. Naturalmente, o `NewsletterManager` também deve ser escrito para permitir uma dependência opcional:

```
public function __construct(Mailer $mailer = null)
{
    // ...
}
```

Serviços do Núcleo do Symfony e de Bundles de Terceiros

Desde que o Symfony2 e todos os bundles de terceiros configuram e recuperam os seus serviços através do container, você pode acessá-los facilmente ou até mesmo usá-los em seus próprios serviços. Para manter tudo simples, o Symfony2, por padrão, não exige que controladores sejam definidos como serviços. Além disso, o Symfony2 injeta o container de serviço inteiro em seu controlador. Por exemplo, para gerenciar o armazenamento de informações em uma sessão do usuário, o Symfony2 fornece um serviço `session`, que você pode acessar dentro de um controlador padrão da seguinte forma:

```
public function indexAction($bar)
{
    $session = $this->get('session');
    $session->set('foo', $bar);

    // ...
}
```

No Symfony2, você vai usar constantemente os serviços fornecidos pelo núcleo do Symfony ou outros bundles de terceiros para executar tarefas como renderização de templates (`templating`), envio de e-mails (`mailer`), ou acessar informações sobre o pedido(`request`).

Podemos levar isto um passo adiante, usando esses serviços dentro dos serviços que você criou para a sua aplicação. Vamos modificar o `NewsletterManager` para utilizar o serviço de `mailer` real do Symfony2 (no lugar do `my_mailer`). Vamos também passar o serviço de templating engine para o `NewsletterManager` então ele poderá gerar o conteúdo de e-mail através de um template:

```
namespace Acme\HelloBundle\Newsletter;

use Symfony\Component\Templating\EngineInterface;

class NewsletterManager
{
    protected $mailer;

    protected $templating;

    public function __construct(\Swift_Mailer $mailer, EngineInterface $templating)
    {
        $this->mailer = $mailer;
        $this->templating = $templating;
    }

    // ...
}
```

A configuração do service container é fácil:

- *YAML*

```
    services:
        newsletter_manager:
            class:     %newsletter_manager.class%
            arguments: [@mailer, @templating]
```

- *XML*

```
    <service id="newsletter_manager" class="%newsletter_manager.class%">
        <argument type="service" id="mailer"/>
        <argument type="service" id="templating"/>
    </service>
```

- *PHP*

```
$container->setDefinition('newsletter_manager', new Definition(
    '%newsletter_manager.class%',
    array(
        new Reference('mailer'),
        new Reference('templating')
    )
));
```

O serviço `newsletter_manager` agora tem acesso aos serviços do núcleo `mailer` and `templating`. Esta é uma forma comum de criar serviços específicos para sua aplicação que aproveitam o poder de diferentes serviços dentro do framework.

Dica: Certifique-se de que a entrada `swiftmailer` aparece na configuração da sua aplicação. Como mencionamos em *Importando Configuração através de Extensões do Container*, a chave `swiftmailer` invoca a extensão de serviço do `SwiftmailerBundle`, que registra o serviço `mailer`.

Configuração Avançada do Container

Como vimos, a definição de serviços no interior do container é fácil, geralmente envolvendo uma chave de configuração `service` e alguns parâmetros. No entanto, o container possui várias outras ferramentas disponíveis que ajudam a adicionar uma *tag* para uma funcionalidade especial nos serviços, criar serviços mais complexos e executar operações após o container estar construído.

Marcando Serviços como público / privado

Ao definir os serviços, normalmente você vai desejar poder acessar essas definições dentro do código da sua aplicação. Esses serviços são chamados `publicos`. Por exemplo, o serviço `doctrine` registrado com o container quando se utiliza o DoctrineBundle é um serviço público, já que você pode acessá-lo via:

```
$doctrine = $container->get('doctrine');
```

No entanto, existem casos de uso em que você não deseja que um serviço seja público. Isto é comum quando um serviço é definido apenas porque poderia ser usado como um argumento para um outro serviço.

Nota: Se você usar um serviço privado como um argumento para mais de um serviço, isto irá resultar em duas instâncias diferentes sendo usadas, já que, a instanciação do serviço privado é realizada inline (por exemplo: `new PrivateFooBar()`).

Simplesmente falando: Um serviço será privado quando você não quiser acessá-lo diretamente em seu código.

Aqui está um exemplo:

- *YAML*

```
services:
   foo:
     class: Acme\HelloBundle\Foo
     public: false
```

- *XML*

```
<service id="foo" class="Acme\HelloBundle\Foo" public="false" />
```

- *PHP*

```
$definition = new Definition('Acme\HelloBundle\Foo');
$definition->setPublic(false);
$container->setDefinition('foo', $definition);
```

Agora que o serviço é privado, você *não* pode chamar:

```
$container->get('foo');
```

No entanto, se o serviço foi marcado como privado, você ainda pode adicionar um alias para ele (veja abaixo) para acessar este serviço (através do alias).

Nota: Os serviços são públicos por padrão.

Definindo Alias

Ao usar bundles do núcleo ou de terceiros dentro de sua aplicação, você pode desejar usar atalhos para acessar alguns serviços. Isto é possível adicionando alias à eles e, além disso, você pode até mesmo adicionar alias para serviços que não são públicos.

- *YAML*

```
services:
   foo:
     class: Acme\HelloBundle\Foo
   bar:
     alias: foo
```

- *XML*

```
<service id="foo" class="Acme\HelloBundle\Foo"/>

<service id="bar" alias="foo" />
```

- *PHP*

```
$definition = new Definition('Acme\HelloBundle\Foo');
$container->setDefinition('foo', $definition);

$containerBuilder->setAlias('bar', 'foo');
```

Isto significa que, ao usar o container diretamente, você pode acessar o serviço `foo` pedindo pelo serviço `bar` como segue:

```
$container->get('bar'); // retorna o serviço foo
```

Incluindo Arquivos

Podem haver casos de uso quando é necessário incluir outro arquivo antes do serviço em si ser carregado. Para fazer isso, você pode usar a diretiva `file`.

- *YAML*

```
services:
   foo:
     class: Acme\HelloBundle\Foo\Bar
     file: %kernel.root_dir%/src/path/to/file/foo.php
```

- *XML*

```
<service id="foo" class="Acme\HelloBundle\Foo\Bar">
    <file>%kernel.root_dir%/src/path/to/file/foo.php</file>
</service>
```

- *PHP*

```
$definition = new Definition('Acme\HelloBundle\Foo\Bar');
$definition->setFile('%kernel.root_dir%/src/path/to/file/foo.php');
$container->setDefinition('foo', $definition);
```

Observe que o symfony irá, internamente, chamar a função PHP require_once o que significa que seu arquivo será incluído apenas uma vez por pedido.

Tags (`tags`)

Da mesma forma que podem ser definidas tags para um post de um blog na web com palavras como "Symfony" ou "PHP", também podem ser definidas tags para os serviços configurados em seu container. No container de serviço, a presença de uma tag significa que o serviço destina-se ao uso para um fim específico. Veja o seguinte exemplo:

- *YAML*

```
services:
    foo.twig.extension:
        class: Acme\HelloBundle\Extension\FooExtension
        tags:
            -  { name: twig.extension }
```

- *XML*

```
<service id="foo.twig.extension" class="Acme\HelloBundle\Extension\FooExtension">
    <tag name="twig.extension" />
</service>
```

- *PHP*

```
$definition = new Definition('Acme\HelloBundle\Extension\FooExtension');
$definition->addTag('twig.extension');
$container->setDefinition('foo.twig.extension', $definition);
```

A tag `twig.extension` é uma tag especial que o `TwigBundle` usa durante a configuração. Ao definir a tag `twig.extension` para este serviço, o bundle saberá que o serviço `foo.twig.extension` deve ser registrado como uma extensão Twig com o Twig. Em outras palavras, o Twig encontra todos os serviços com a tag `twig.extension` e automaticamente registra-os como extensões.

Tags, então, são uma forma de dizer ao Symfony2 ou outros bundles de terceiros que o seu serviço deve ser registrado ou usado de alguma forma especial pelo bundle.

Segue abaixo uma lista das tags disponíveis com os bundles do núcleo do Symfony2. Cada uma delas tem um efeito diferente no seu serviço e muitas tags requerem argumentos adicionais (além do parâmetro `name`).

- assetic.filter

- assetic.templating.php
- data_collector
- form.field_factory.guesser
- kernel.cache_warmer
- kernel.event_listener
- monolog.logger
- routing.loader
- security.listener.factory
- security.voter
- templating.helper
- twig.extension
- translation.loader
- validator.constraint_validator

Saiba mais

- `/components/dependency_injection/factories`
- `/components/dependency_injection/parentservices`
- Como definir Controladores como Serviços

2.1.15 Performance

O Symfony2 é rápido, logo após a sua instalação. Claro, se você realmente precisa de mais velocidade, há muitas maneiras para tornar o Symfony ainda mais rápido. Neste capítulo, você vai explorar muitas das formas mais comuns e poderosas para tornar a sua aplicação Symfony ainda mais rápida.

Use um Cache de Código Byte (ex.: APC)

Uma das melhores (e mais fáceis) coisas que você deve fazer para melhorar a sua performance é usar um "cache de código byte". A idéia de um cache de código byte é remover a necessidade de constantemente recompilar o código fonte PHP. Há uma série de *Caches de código byte* _ disponíveis, alguns dos quais são de código aberto. O cache de código byte mais amplamente usado é, provavelmente, o APC

Usar um cache de código byte não tem um lado negativo, e o Symfony2 foi arquitetado para executar realmente bem neste tipo de ambiente.

Mais otimizações

Caches de código byte normalmente monitoram as alterações nos arquivos fonte. Isso garante que, se o fonte de um arquivo for alterado, o código byte é recompilado automaticamente. Isto é realmente conveniente, mas, obviamente, adiciona uma sobrecarga.

Por este motivo, alguns caches de código byte oferecem uma opção para desabilitar estas verificações. Obviamente, quando desabilitar estas verificações, caberá ao administrador do servidor garantir que o cache seja limpo sempre que houver qualquer alteração nos arquivos fonte. Caso contrário, as atualizações que você fizer nunca serão vistas.

Por exemplo, para desativar estas verificações no APC, basta adicionar `apc.stat=0` ao seu arquivo de configuração php.ini.

Utilize um Autoloader com cache (ex.: `ApcUniversalClassLoader`)

Por padrão, a edição standard do Symfony2 usa o `UniversalClassLoader` no arquivo autoloader.php . Este autoloader é fácil de usar, pois ele encontra automaticamente as novas classes que você colocou nos diretórios registrados.

Infelizmente, isso tem um custo, devido ao carregador iterar sobre todos os namespaces configurados para encontrar um determinado arquivo, fazendo chamadas `file_exists` até que, finalmente, encontre o arquivo que estiver procurando.

A solução mais simples é armazenar em cache a localização de cada classe após ter sido localizada pela primeira vez. O Symfony vem com um carregador de classe - `ApcClassLoader` - que faz exatamente isso. Para usá-lo, basta adaptar seu arquivo front controller. Se você estiver usando a Distribuição Standard, este código já deve estar disponível com comentários neste arquivo:

```
// app.php
// ...

$loader = require_once __DIR__.'/../app/bootstrap.php.cache';

// Use APC for autoloading to improve performance
// Change 'sf2' by the prefix you want in order to prevent key conflict with another application
/*
$loader = new ApcClassLoader('sf2', $loader);
$loader->register(true);
*/

// ...
```

Nota: Ao usar o autoloader APC, se você adicionar novas classes, elas serão encontradas automaticamente e tudo vai funcionar da mesma forma como antes (ou seja, sem razão para "limpar" o cache). No entanto, se você alterar a localização de um namespace ou prefixo em particular, você vai precisar limpar o cache do APC. Caso contrário, o autoloader ainda estará procurando pelo local antigo para todas as classes dentro desse namespace.

Utilize arquivos de inicialização (bootstrap)

Para garantir a máxima flexibilidade e reutilização de código, as aplicações do Symfony2 aproveitam uma variedade de classes e componentes de terceiros. Mas, carregar todas essas classes de arquivos separados em cada requisição pode resultar em alguma sobrecarga. Para reduzir essa sobrecarga, a Edição Standard do Symfony2 fornece um script para gerar o chamado arquivo de inicialização, que consiste em múltiplas definições de classes em um único arquivo. Ao incluir este arquivo (que contém uma cópia de muitas das classes core), o Symfony não precisa incluir nenhum dos arquivos fonte contendo essas classes. Isto reduzirá bastante a E/S no disco.

Se você estiver usando a Edição Standard do Symfony2, então, você provavelmente já está usando o arquivo de inicialização. Para ter certeza, abra o seu `front controller` (geralmente `app.php`) e, certifique-se que existe a seguinte linha:

```
require_once __DIR__.'/../app/bootstrap.php.cache';
```

Note que existem duas desvantagens ao usar um arquivo de inicialização:

- O arquivo precisa ser regerado, sempre que houver qualquer mudança nos fontes originais (ex.: quando você atualizar o fonte do Symfony2 ou as bibliotecas vendor);

- Quando estiver debugando, é necessário colocar `break points` dentro do arquivo de inicialização.

Se você estiver usando a Edição Standard do Symfony2, o arquivo de inicialização é automaticamente reconstruído após a atualização das bibliotecas vendor através do comando `php composer.phar install`.

Arquivos de inicialização e caches de código byte

Mesmo quando se utiliza um cache de código byte, o desempenho irá melhorar quando se utiliza um arquivo de inicialização, pois, haverá menos arquivos para monitorar as mudanças. Claro, se este recurso está desativado no cache de código byte (ex.: `apc.stat=0` no APC), não há mais motivo para usar um arquivo de inicialização.

2.1.16 API estável do Symfony2

A API estável do Symfony2 é um subconjunto de todos métodos públicos publicados (componentes e bundles principais) que possuem as seguintes propriedades:

- O namespace e o nome da classe não mudarão;
- O nome do método não mudará;
- A assinatura do método (parâmetros e tipo de retorno) não mudará;
- A função do método (o que ele faz) não mudará;

A implementação em si pode mudar. O único caso que pode causar alteração na API estável será para correção de algum problema de segurança.

A API estável é baseada em uma lista (whitelist), marcada com *@api*. Assim, tudo que não esteja marcado com *@api* não é parte da API estável.

Dica: Qualquer bundle de terceiros deveria também publicar sua própria API estável.

Atualmente na versão Symfony 2.0 os seguintes componentes têm API pública marcada:

- BrowserKit
- ClassLoader
- Console
- CssSelector
- DependencyInjection
- DomCrawler
- EventDispatcher
- Finder
- HttpFoundation
- HttpKernel
- Locale
- Process
- Routing
- Templating

- Translation
- Validator
- Yaml
- Fundamentos de Symfony e HTTP
- Symfony2 versus o PHP puro
- Instalando e Configurando o Symfony
- Controlador
- Roteamento
- Criando e usando Templates
- Bancos de Dados e Doctrine
- Testes
- Validação
- Formulários
- Segurança
- HTTP Cache
- Traduções
- Container de Serviço
- Performance
- API estável do Symfony2
- Fundamentos de Symfony e HTTP
- Symfony2 versus o PHP puro
- Instalando e Configurando o Symfony
- Controlador
- Roteamento
- Criando e usando Templates
- Bancos de Dados e Doctrine
- Testes
- Validação
- Formulários
- Segurança
- HTTP Cache
- Traduções
- Container de Serviço
- Performance
- API estável do Symfony2

CAPÍTULO 3

Cookbook

3.1 Cookbook

3.1.1 Workflow

Como Criar e Armazenar um Projeto Symfony2 no git

Dica: Embora este artigo seja especificamente sobre git, os mesmos princípios genéricos aplicam-se caso você estiver armazenando o seu projeto no Subversion.

Uma vez que você leu `/book/page_creation` e familiarizou-se com o Symfony, já está, sem dúvida, pronto para começar o seu próprio projeto. Neste artigo do cookbook, você aprenderá a melhor maneira de iniciar um novo projeto Symfony2 que será armazenado usando o sistema gerenciador de controle de versão git.

Configuração Inicial do Projeto

Para começar, você precisa baixar o Symfony e inicializar o seu repositório git local:

1. Baixe a Edição Standard do Symfony2 sem vendors.
2. Descompacte a distribuição. Será criado um diretório chamado Symfony com a sua nova estrutura do projeto, arquivos de configuração, etc. Renomeie-o como desejar.
3. Crie um novo arquivo chamado `.gitignore` no raiz de seu novo projeto (Ex.: junto com o arquivo `composer.json`) e cole o conteúdo seguinte nele. Os arquivos correspondentes a estes padrões serão ignorados pelo git:

```
/web/bundles/
/app/bootstrap*
/app/cache/*
/app/logs/*
/vendor/
/app/config/parameters.yml
```

4. Copie `app/config/parameters.yml` para `app/config/parameters.yml.dist`. O arquivo `parameters.yml` é ignorado pelo git (veja acima), de modo que as configurações específicas da máquina, como senhas de banco de dados, não sejam comitadas. Criando o arquivo `parameters.yml.dist`, novos desenvolvedores podem, rapidamente, clonar o projeto, copiar este arquivo para `parameters.yml`, personalizá-lo e iniciar o desenvolvimento.

5. Inicialize o seu repositório git:

```
$ git init
```

6. Adicione todos os arquivos iniciais ao git:

```
$ git add .
```

7. Crie um commit inicial para o seu projeto:

```
$ git commit -m "Initial commit"
```

8. Finalmente, baixe todas as bibliotecas vendor de terceiros executando o composer. Para detalhes, veja *Atualizando os Vendors*.

Neste ponto, você tem um projeto Symfony2 totalmente funcional que está corretamente comitado com o git. Você pode iniciar imediatamente o desenvolvimento, comitando as novas alterações em seu repositório git.

Você pode continuar acompanhando o capítulo `/book/page_creation` para saber mais sobre como configurar e desenvolver dentro da sua aplicação.

Dica: A Edição Standard do Symfony2 vem com algumas funcionalidades exemplo. Para remover o código de exemplo, siga as instruções contidas no Readme da Edição Standard.

Gerenciando Bibliotecas Vendor com bin/vendors e deps

Cada projeto Symfony usa um grande grupo de bibliotecas "vendor" de terceiros.

Por padrão, estas bibliotecas são baixadas executando o script `php bin/vendors install`. Este script lê o arquivo `deps`, e baixa as bibliotecas ali informadas no diretório `vendor/`. Ele também lê o arquivo `deps.lock`, fixando cada biblioteca listada ao respectivo hash do commit git.

Nesta configuração, as bibliotecas vendor não fazem parte de seu repositório git, nem mesmo como sub-módulos. Em vez disso, contamos com os arquivos `deps` e `deps.lock` e o script `bin/vendors` para gerenciar tudo. Esses arquivos são parte de seu repositório, então, as versões necessárias de cada biblioteca de terceiros tem controle de versão no git, e você pode usar o script vendors para trazer o seu projeto atualizado.

Sempre que um desenvolvedor clona um projeto, ele(a) deve executar o script `php bin/vendors install` para garantir que todas as bibliotecas vendor necessárias foram baixadas.

Atualizando o Symfony

Uma vez que o Symfony é apenas um grupo de bibliotecas de terceiros e estas bibliotecas são totalmente controladas através do `deps` e `deps.lock`, atualizar o Symfony significa, simplesmente, atualizar cada um desses arquivos para combinar seu estado na última Edição Standard do Symfony.
Claro, se você adicionou novas entradas ao `deps` ou `deps.lock`, certifique-se de substituir apenas as partes originais (ou seja, não excluir as suas entradas personalizadas).

Cuidado: Há também um comando `php bin/vendors update`, mas isso não tem nada a ver com a atualização do seu projeto e você normalmente não precisará utilizá-lo. Este comando é usado para congelar as versões de todas as suas bibliotecas vendor atualizando-as para a versão especificada em `deps` e gravando-as no arquivo `deps.lock`.
Além disso, se você deseja simplesmente atualizar o arquivo `deps.lock` para o que já tem instalado, então, você pode simplesmente executar `php bin/vendors lock` para armazenar os identificadores git SHA apropriados no arquivo deps.lock.

Vendors e sub-módulos Em vez de usar o sistema `composer.json` para gerenciar suas bibliotecas vendor, você pode optar por usar o git submodules nativo. Não há nada de errado com esta abordagem, embora o sistema `composer.json` é a forma oficial de resolver este problema e provavelmente mais fácil de lidar. Ao contrário do `git submodules`, o `Composer` é esperto o suficiente para calcular quais bibliotecas dependem de outras bibliotecas.

Armazenando o seu Projeto em um Servidor Remoto

Agora, você tem um projeto Symfony2 totalmente funcional armazenado com o git. Entretanto, na maioria dos casos, você também vai querer guardar o seu projeto em um servidor remoto tanto para fins de backup quanto para que outros desenvolvedores possam colaborar com o projeto.

A maneira mais fácil para armazenar o seu projeto em um servidor remoto é via GitHub. Repositórios públicos são gratuitos, porém, você terá que pagar uma taxa mensal para hospedar repositórios privados.

Alternativamente, você pode armazenar seu repositório git em qualquer servidor, criando um repositório barebones e, então, enviá-lo. Uma biblioteca que ajuda neste gerenciamento é a Gitolite.

3.1.2 Controlador

Como personalizar as páginas de erro

Quando qualquer exceção é lançada no Symfony2, ela é capturada dentro da classe `Kernel` e, eventualmente, encaminhada para um controlador especial, `TwigBundle:Exception:show`, para o tratamento. Este controlador, que reside no interior do núcleo do `TwigBundle`, determina qual template de erro será exibido e o código de status que deve ser definido para a exceção.

Páginas de erro pode ser personalizadas de duas maneiras diferentes, dependendo da quantidade de controle que você precisa:

1. Personalizar os templates de erro das diferentes páginas de erro (explicado abaixo);
2. Substituir o controlador de exceção padrão `TwigBundle::Exception:show` pelo seu próprio controlador e manipulá-lo como quiser (veja exception_controller in the Twig reference);

Dica: A personalização da manipulação de exceção é, na verdade, muito mais poderosa do que o que está escrito aqui. Um evento interno é lançado, `kernel.exception`, que permite controle completo sobre o tratamento de exceção. Para mais informações, veja kernel-kernel.exception.

Todos os templates de erro residem dentro do `TwigBundle`. Para sobrescrever os templates, simplesmente contamos com o método padrão para sobrescrever templates que reside dentro de um bundle. Para maiores informações, visite *Sobrepondo Templates de Pacote*.

Por exemplo, para sobrescrever o template padrão de erro que é exibido ao usuário final, adicione um novo template em `app/Resources/TwigBundle/views/Exception/error.html.twig`:

```
<!DOCTYPE html>
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <title>An Error Occurred: {{ status_text }}</title>
</head>
```

```
<body>
    <h1>Oops! An Error Occurred</h1>
    <h2>The server returned a "{{ status_code }} {{ status_text }}".</h2>
</body>
</html>
```

Dica: Se você não estiver familiarizado com o Twig, não se preocupe. O Twig é uma templating engine simples, poderosa e opcional que integra o `Symfony2`. Para mais informações sobre o Twig, consulte Criando e usando Templates.

Além da página HTML de erro padrão, o Symfony fornece uma página padrão de erro para muitos dos formatos de resposta mais comuns, incluindo JSON (`error.json.twig`), XML (`error.xml.twig`) e até mesmo JavaScript (`error.js.twig`), somente para citar alguns. Para sobrescrever qualquer um destes templates, apenas adicione um novo arquivo com o mesmo nome no diretório `app/Resources/TwigBundle/views/Exception`. Esta é a forma padrão de sobrescrever qualquer template que reside dentro de um bundle.

Personalizando a Página 404 e outras Páginas de Erro

Você também pode personalizar os templates de erro específicos de acordo com o código de status HTTP. Por exemplo, crie um template `app/Resources/TwigBundle/views/Exception/error404.html.twig` para exibir uma página especial para erros 404 (página não encontrada).

O Symfony usa o seguinte algoritmo para determinar qual o template que deve usar:

- Primeiro, ele procura por um template para o formato e código de status especificado (como `error404.json.twig`);
- Se não existir, ele procura um template para o formato especificado (como `error.json.twig`);
- Se ainda não existir, ele volta para o template HTML (como `error.html.twig`).

Dica: Para ver a lista completa de templates de erro padrão, consulte o diretório `Resources/views/Exception` do `TwigBundle`. Na instalação Standard do Symfony2, o `TwigBundle` pode ser encontrado em `vendor/symfony/src/Symfony/Bundle/TwigBundle`. Muitas vezes, a maneira mais fácil para personalizar uma página de erro é copiá-la do `TwigBundle` para o `app/Resources/TwigBundle/views/Exception` e então modificá-la.

Nota: As páginas de exceção amigáveis de depuração, mostradas para o desenvolvedor, podem ser personalizadas da mesma forma, criando templates como `exception.html.twig` para a página de exceção padrão HTML ou `exception.json.twig` para a página de exceção JSON.

Como definir Controladores como Serviços

No livro, você aprendeu como um controlador pode ser facilmente usado quando ele estende a classe base `Controller`. Mesmo isso funcionando bem, os controladores também podem ser especificados como serviços.

Para referir-se a um controlador que é definido como um serviço, utilize a notação de um único dois pontos (:) . Por exemplo, supondo que você definiu um serviço chamado `my_controller` e que deseja encaminhar para um método chamado `indexAction()` dentro do serviço:

```
$this->forward('my_controller:indexAction', array('foo' => $bar));
```

Você precisa usar a mesma notação ao definir o valor da rota `_controller`:

```
my_controller:
    pattern:   /
    defaults:  { _controller: my_controller:indexAction }
```

Para utilizar um controlador desta forma, ele deve estar definido na configuração do container de serviço. Para mais informações, consulte o capítulo Service Container

Ao usar um controlador definido como um serviço, ele provavelmente não estenderá a classe base `Controller`. Em vez de contar com seus métodos de atalho, você vai interagir diretamente com os serviços que você precisa. Felizmente, isto é normalmente muito fácil e a classe base `Controller` em si é uma grande fonte sobre a forma de realizar muitas das tarefas comuns.

Nota: Especificar um controlador como um serviço leva um pouco mais de trabalho. A principal vantagem é que todo o controlador ou quaisquer serviços passados para o controlador podem ser modificados através da configuração do container de serviço. Isto é especialmente útil no desenvolvimento de um bundle open-source ou qualquer bundle que será utilizado em vários projetos diferentes. Assim, mesmo se você não especificar os seus controladores como serviços, provavelmente verá isto feito em alguns bundles open-source do Symfony2.

Usando Anotação no Roteamento

Ao usar anotações para configurar as rotas em um controlador definido como um serviço, é necessário especificar o serviço da seguinte forma:

```
/**
 * @Route("/blog", service="my_bundle.annot_controller")
 * @Cache(expires="tomorrow")
 */
class AnnotController extends Controller
{
}
```

Neste exemplo, `my_bundle.annot_controller` deve ser o id da instância `AnnotController` definida no container de serviço. Isto está documentado no capítulo `/bundles/SensioFrameworkExtraBundle/annotations/routing.`

3.1.3 Roteamento

Como forçar as rotas a usar sempre HTTPS ou HTTP

Às vezes, você deseja proteger algumas rotas e ter certeza de que elas serão sempre acessadas através do protocolo HTTPS. O componente de Roteamento permite que você aplique o esquema URI através da condição `_scheme`:

- *YAML*

```
    secure:
        pattern:  /secure
        defaults: { _controller: AcmeDemoBundle:Main:secure }
        requirements:
            _scheme:  https
```

- *XML*

```
<?xml version="1.0" encoding="UTF-8" ?>

<routes xmlns="http://symfony.com/schema/routing"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/routing http://symfony.com/schema/routing/rout

    <route id="secure" pattern="/secure">
        <default key="_controller">AcmeDemoBundle:Main:secure</default>
        <requirement key="_scheme">https</requirement>
    </route>
</routes>
```

- *PHP*

```
use Symfony\Component\Routing\RouteCollection;
use Symfony\Component\Routing\Route;

$collection = new RouteCollection();
$collection->add('secure', new Route('/secure', array(
    '_controller' => 'AcmeDemoBundle:Main:secure',
), array(
    '_scheme' => 'https',
)));

return $collection;
```

A configuração acima força a rota `secure` à sempre usar HTTPS.

Ao gerar a URL `secure`, e se o schema atual for HTTP, o Symfony irá gerar automaticamente uma URL absoluta com HTTPS como esquema:

```
# If the current scheme is HTTPS
{{ path('secure') }}
# generates /secure

# If the current scheme is HTTP
{{ path('secure') }}
# generates https://example.com/secure
```

A condição também é aplicada para as solicitações de entrada. Se você tentar acessar o caminho `/secure` com HTTP, você será automaticamente redirecionado para a mesma URL, mas com o esquema HTTPS.

O exemplo acima utiliza `https` para o `_scheme`, mas você também pode forçar uma URL à sempre utilizar `http`.

Nota: O componente Security fornece outra forma de aplicar HTTP ou HTTPs através da configuração `requires_channel`. Este método alternativo é mais adequado para proteger uma "área" do seu site (todas as URLs sob o `/admin`) ou quando você quiser proteger URLs definidas em um bundle de terceiros.

Como permitir um caractere "/" em um parâmetro de rota

Às vezes, você precisa compor URLs com parâmetros que podem conter uma barra `/`. Por exemplo, considere a rota clássica `/hello/{name}`. Por padrão, `/hello/Fabien` irá corresponder a esta rota, mas `/hello/Fabien/Kris` não irá. Isso ocorre porque o Symfony usa esse caractere como separador entre as partes da rota.

Este guia aborda como você pode modificar a rota para que `/hello/Fabien/Kris` corresponda à rota `/hello/{name}`, onde `{name}` iguala `Fabien/Kris`.

Configure a Rota

Por padrão, o componente de roteamento do Symfony requer que os parâmetros correspondam com o seguinte path de expressão regular: `[^/]+`. Isso significa que todos os caracteres são permitidos, exceto `/`.

Você deve explicitamente permitir que a `/` faça parte de seu parâmetro, especificando um path regex mais permissivo.

- *YAML*

```
_hello:
    path: /hello/{name}
    defaults: { _controller: AcmeDemoBundle:Demo:hello }
    requirements:
        name: ".+"
```

- *XML*

```
<?xml version="1.0" encoding="UTF-8" ?>

<routes xmlns="http://symfony.com/schema/routing"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/routing http://symfony.com/schema/routing/rout

    <route id="_hello" path="/hello/{name}">
        <default key="_controller">AcmeDemoBundle:Demo:hello</default>
        <requirement key="name">.+</requirement>
    </route>
</routes>
```

- *PHP*

```
use Symfony\Component\Routing\RouteCollection;
use Symfony\Component\Routing\Route;

$collection = new RouteCollection();
$collection->add('_hello', new Route('/hello/{name}', array(
    '_controller' => 'AcmeDemoBundle:Demo:hello',
), array(
    'name' => '.+',
)));

return $collection;
```

- *Annotations*

```
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;

class DemoController
{
    /**
     * @Route("/hello/{name}", name="_hello", requirements={"name" = ".+"})
     */
    public function helloAction($name)
    {
        // ...
```

```
        }
    }
```

É isso! Agora, o parâmetro `{name}` pode conter o caractere `/`.

Como configurar um redirecionamento para outra rota sem um controlador personalizado

Este guia explica como configurar um redirecionamento de uma rota para outra sem o uso de um controlador personalizado.

Suponha que não há nenhum controlador padrão útil para o caminho `/` da sua aplicação e você quer redirecionar os pedidos para `/app`.

Sua configuração será parecida com a seguinte:

```
AppBundle:
    resource: "@App/Controller/"
    type:     annotation
    prefix:   /app

root:
    pattern: /
    defaults:
        _controller: FrameworkBundle:Redirect:urlRedirect
        path: /app
        permanent: true
```

Neste exemplo, você configura uma rota para o caminho `/` e deixa o class:*Symfony\Bundle\FrameworkBundle\Controller\RedirectController* lidar com ela. Este controlador vem com o Symfony e oferece duas ações para redirecionar o pedido:

- `urlRedirect` redireciona para outro *caminho*. Você deve fornecer o parâmetro `path` contendo o caminho do recurso para o qual deseja redirecionar.
- `redirect` (não mostrado aqui) redireciona para outra *rota*. Você deve fornecer o parâmetro `route` com o *nome* da rota para a qual você quer redirecionar.

O `permanent` informa ambos os métodos para emitir um código de status HTTP 301 em vez do código de status padrão `302`.

Como usar métodos HTTP além do GET e POST em Rotas

O método de um pedido HTTP é um dos requisitos que pode ser verificado ao ver se ele corresponde a uma rota. Isto é introduzido no capítulo de roteamento do livro "Roteamento" com exemplos usando GET e POST. Você também pode usar outros verbos HTTP desta forma. Por exemplo, se você tem um post de blog, então, você pode usar o mesmo padrão de URL para mostrá-lo, fazer alterações e removê-lo pela correspondência nos métodos GET, PUT e DELETE.

- *YAML*

```
    blog_show:
        pattern:  /blog/{slug}
        defaults: { _controller: AcmeDemoBundle:Blog:show }
        requirements:
            _method:  GET

    blog_update:
        pattern:  /blog/{slug}
        defaults: { _controller: AcmeDemoBundle:Blog:update }
```

```
        requirements:
            _method:  PUT

    blog_delete:
        pattern:  /blog/{slug}
        defaults: { _controller: AcmeDemoBundle:Blog:delete }
        requirements:
            _method:  DELETE
```

- *XML*

```
    <?xml version="1.0" encoding="UTF-8" ?>

    <routes xmlns="http://symfony.com/schema/routing"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://symfony.com/schema/routing http://symfony.com/schema/routing/rout

        <route id="blog_show" pattern="/blog/{slug}">
            <default key="_controller">AcmeDemoBundle:Blog:show</default>
            <requirement key="_method">GET</requirement>
        </route>

        <route id="blog_update" pattern="/blog/{slug}">
            <default key="_controller">AcmeDemoBundle:Blog:update</default>
            <requirement key="_method">PUT</requirement>
        </route>

        <route id="blog_delete" pattern="/blog/{slug}">
            <default key="_controller">AcmeDemoBundle:Blog:delete</default>
            <requirement key="_method">DELETE</requirement>
        </route>
    </routes>
```

- *PHP*

```
    use Symfony\Component\Routing\RouteCollection;
    use Symfony\Component\Routing\Route;

    $collection = new RouteCollection();
    $collection->add('blog_show', new Route('/blog/{slug}', array(
        '_controller' => 'AcmeDemoBundle:Blog:show',
    ), array(
        '_method' => 'GET',
    )));

    $collection->add('blog_update', new Route('/blog/{slug}', array(
        '_controller' => 'AcmeDemoBundle:Blog:update',
    ), array(
        '_method' => 'PUT',
    )));

    $collection->add('blog_delete', new Route('/blog/{slug}', array(
        '_controller' => 'AcmeDemoBundle:Blog:delete',
    ), array(
        '_method' => 'DELETE',
    )));

    return $collection;
```

Infelizmente, a vida não é tão simples assim, já que a maioria dos navegadores não suporta o envio de solicitações PUT e DELETE. Felizmente o Symfony2 fornece uma maneira simples de trabalhar com esta limitação. Ao incluir um parâmetro `_method` na query string ou nos parâmetros de um pedido HTTP, o Symfony2 irá usá-lo como o método ao fazer a correspondência de rotas. Isto pode ser feito facilmente em formulários com um campo oculto. Suponha que você tenha um formulário para editar um post no blog:

```
<form action="{{ path('blog_update', {'slug': blog.slug}) }}" method="post">
    <input type="hidden" name="_method" value="PUT" />
    {{ form_widget(form) }}
    <input type="submit" value="Update" />
</form>
```

O pedido submetido agora vai corresponder à rota `blog_update` e a `updateAction` será utilizada para processar o formulário.

Do mesmo modo, o formulário de exclusão pode ser alterado para parecer com o seguinte:

```
<form action="{{ path('blog_delete', {'slug': blog.slug}) }}" method="post">
    <input type="hidden" name="_method" value="DELETE" />
    {{ form_widget(delete_form) }}
    <input type="submit" value="Delete" />
</form>
```

Ele irá então corresponder à rota `blog_delete`.

Como usar os Parâmetros do Container de Serviço em suas Rotas

Novo na versão 2.1: A possibilidade de usar parâmetros em suas rotas foi adicionada no Symfony 2.1.

Às vezes pode ser útil tornar algumas partes de suas rotas configuráveis globalmente. Por exemplo, se você construir um site internacionalizado , você provavelmente vai começar com uma ou duas localidades. Certamente você irá adicionar um requisito em suas rotas para evitar que um usuário utilize uma localidade além das suportadas.

Você *pode* codificar manualmente seu requisito `_locale` em todas as suas rotas. Mas uma solução melhor é usar um parâmetro configurável do container de serviço dentro de sua configuração de roteamento:

- *YAML*

```
contact:
    path:  /{_locale}/contact
    defaults: { _controller: AcmeDemoBundle:Main:contact }
    requirements:
        _locale: %acme_demo.locales%
```

- *XML*

```
<?xml version="1.0" encoding="UTF-8" ?>

<routes xmlns="http://symfony.com/schema/routing"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/routing http://symfony.com/schema/routing/rout

    <route id="contact" path="/{_locale}/contact">
        <default key="_controller">AcmeDemoBundle:Main:contact</default>
        <requirement key="_locale">%acme_demo.locales%</requirement>
    </route>
</routes>
```

- *PHP*

```
use Symfony\Component\Routing\RouteCollection;
use Symfony\Component\Routing\Route;

$collection = new RouteCollection();
$collection->add('contact', new Route('/{_locale}/contact', array(
    '_controller' => 'AcmeDemoBundle:Main:contact',
), array(
    '_locale' => '%acme_demo.locales%',
)));

return $collection;
```

Agora você pode controlar e definir o parâmetro `acme_demo.locales` em algum lugar no seu container:

- *YAML*

```
# app/config/config.yml
parameters:
    acme_demo.locales: en|es
```

- *XML*

```
<!-- app/config/config.xml -->
<parameters>
    <parameter key="acme_demo.locales">en|es</parameter>
</parameters>
```

- *PHP*

```
// app/config/config.php
$container->setParameter('acme_demo.locales', 'en|es');
```

Você também pode usar um parâmetro para definir o seu path da rota (ou parte do seu path):

- *YAML*

```
some_route:
    path:  /%acme_demo.route_prefix%/contact
    defaults: { _controller: AcmeDemoBundle:Main:contact }
```

- *XML*

```
<?xml version="1.0" encoding="UTF-8" ?>

<routes xmlns="http://symfony.com/schema/routing"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/routing http://symfony.com/schema/routing/rout

    <route id="some_route" path="/%acme_demo.route_prefix%/contact">
        <default key="_controller">AcmeDemoBundle:Main:contact</default>
    </route>
</routes>
```

- *PHP*

```
use Symfony\Component\Routing\RouteCollection;
use Symfony\Component\Routing\Route;

$collection = new RouteCollection();
$collection->add('some_route', new Route('/%acme_demo.route_prefix%/contact', array(
    '_controller' => 'AcmeDemoBundle:Main:contact',
```

```
    ));

    return $collection;
```

Nota: Assim como em arquivos normais de configuração do container de serviço, se você precisar de um `%` na sua rota, você pode escapar o sinal de porcentagem duplicando ele , por exemplo, `/score-50%%`, irá converter para `/score-50%`.

No entanto, como os caracteres `%` incluídos em qualquer URL são automaticamente codificados, a URL resultante desse exemplo seria `/score-50%25` (`%25` é o resultado da codificação do caractere `%`).

Como criar um Loader de Rota personalizado

Um loader de rota personalizado permite que você adicione rotas a uma aplicação sem incluí-las, por exemplo, num arquivo yaml. Isto é útil quando você tem um bundle, mas não quer adicionar manualmente as rotas ao bundle em `app/config/routing.yml`. Isso pode ser especialmente importante quando você quer tornar o bundle reutilizável, ou quando o disponibilizar como código aberto, porque iria retardar o processo de instalação e torná-lo suscetível a erros.

Alternativamente, você também pode usar um loader de rota personalizado quando quiser que as suas rotas sejam geradas ou localizadas automaticamente com base em alguma convenção ou padrão. Um exemplo é o FOSRestBundle onde o roteamento é gerado com base nos nomes dos métodos de ação em um controlador.

Nota: Há muitos bundles que usam seus próprios loaders de rotas para realizar casos como os descritos acima, por exemplo FOSRestBundle, KnpRadBundle e SonataAdminBundle.

Carregando Rotas

As rotas em uma aplicação Symfony são carregadas pelo `DelegatingLoader`. Este loader usa vários outros loaders (delegados) para carregar recursos de diferentes tipos, por exemplo, arquivos YAML ou anotações `@Route` e `@Method` em arquivos de controlador. Os loaders especializados implementam a `LoaderInterface` e, portanto, tem dois métodos importantes: `supports()` e `load()`.

Considere essas linhas do `routing.yml`:

```
_demo:
    resource: "@AcmeDemoBundle/Controller/DemoController.php"
    type:     annotation
    prefix:   /demo
```

Quando o loader principal realiza o parse, ele tenta todos os loaders delegados e chama seu método `supports()` com o determinado recurso (`@AcmeDemoBundle/Controller/DemoController.php`) e tipo (`annotation`) como argumentos. Quando um dos loaders retorna `true`, seu método `load()` será chamado, que deve retornar um `RouteCollection` contendo objetos `Route`.

Criando um Loader Personalizado

Para carregar rotas de alguma fonte personalizada (ou seja, de algo diferente de anotações, arquivos YAML ou XML), você precisa criar um loader de rota personalizado. Este loader deve implementar a `LoaderInterface`.

O loader de exemplo abaixo, suporta o carregamento de recursos de roteamento com um tipo `extra`. O tipo `extra` não é importante - você pode simplesmente inventar qualquer tipo de recurso que desejar. O próprio nome do recurso não é realmente usado no exemplo:

```
namespace Acme\DemoBundle\Routing;

use Symfony\Component\Config\Loader\LoaderInterface;
use Symfony\Component\Config\Loader\LoaderResolverInterface;
use Symfony\Component\Routing\Route;
use Symfony\Component\Routing\RouteCollection;

class ExtraLoader implements LoaderInterface
{
    private $loaded = false;

    public function load($resource, $type = null)
    {
        if (true === $this->loaded) {
            throw new \RuntimeException('Do not add the "extra" loader twice');
        }

        $routes = new RouteCollection();

        // prepare a new route
        $pattern = '/extra/{parameter}';
        $defaults = array(
            '_controller' => 'AcmeDemoBundle:Demo:extra',
        );
        $requirements = array(
            'parameter' => '\d+',
        );
        $route = new Route($pattern, $defaults, $requirements);

        // add the new route to the route collection:
        $routeName = 'extraRoute';
        $routes->add($routeName, $route);

        return $routes;
    }

    public function supports($resource, $type = null)
    {
        return 'extra' === $type;
    }

    public function getResolver()
    {
        // needed, but can be blank, unless you want to load other resources
        // and if you do, using the Loader base class is easier (see below)
    }

    public function setResolver(LoaderResolverInterface $resolver)
    {
        // same as above
    }
}
```

Nota: Certifique-se que o controlador que você especificou realmente existe.

Agora defina um serviço para o `ExtraLoader`:

- *YAML*

```
services:
    acme_demo.routing_loader:
        class: Acme\DemoBundle\Routing\ExtraLoader
        tags:
            - { name: routing.loader }
```

- *XML*

```
<?xml version="1.0" ?>
<container xmlns="http://symfony.com/schema/dic/services"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/dic/services http://symfony.com/schema/dic/ser

    <services>
        <service id="acme_demo.routing_loader" class="Acme\DemoBundle\Routing\ExtraLoader">
            <tag name="routing.loader" />
        </service>
    </services>
</container>
```

- *PHP*

```
use Symfony\Component\DependencyInjection\Definition;

$container
    ->setDefinition(
        'acme_demo.routing_loader',
        new Definition('Acme\DemoBundle\Routing\ExtraLoader')
    )
    ->addTag('routing.loader')
;
```

Observe a tag `routing.loader`. Todos os serviços com esta tag serão marcados como potenciais loaders de rota e adicionados como roteadores especializados para o `DelegatingLoader`.

Usando o Loader Personalizado Se você não fez nada mais, seu loader de roteamento personalizado *não* será chamado. Em vez disso, você só precisa adicionar algumas linhas extras para a configuração de roteamento:

- *YAML*

```
# app/config/routing.yml
AcmeDemoBundle_Extra:
    resource: .
    type: extra
```

- *XML*

```
<?xml version="1.0" encoding="UTF-8" ?>
<routes xmlns="http://symfony.com/schema/routing"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/routing http://symfony.com/schema/routing/rout
```

```
        <import resource="." type="extra" />
    </routes>
```

- *PHP*

```
    // app/config/routing.php
    use Symfony\Component\Routing\RouteCollection;

    $collection = new RouteCollection();
    $collection->addCollection($loader->import('.', 'extra'));

    return $collection;
```

A parte importante aqui é a chave `type`. Seu valor deve ser "extra". Este é o tipo que nosso `ExtraLoader` suporta e que irá certificar-se que seu método `load()` é chamado. A chave `resource` é insignificante para o `ExtraLoader`, então defina ela como ".".

Nota: O cache das rotas definidas usando loaders personalizados será feita automaticamente pelo framework. Assim, sempre que você mudar alguma coisa na classe do loader, não se esqueça de limpar o cache.

Loaders mais Avançados

Na maioria dos casos é melhor não implementar a `LoaderInterface` você mesmo, mas estender do `Loader`. Esta classe sabe como usar um `LoaderResolver` para carregar os recursos de roteamento secundários.

Claro que você ainda precisa implementar `supports()` e `load()`. Sempre que quiser carregar outro recurso - por exemplo, um arquivo de configuração Yaml - você pode chamar o `import()` method:

```
namespace Acme\DemoBundle\Routing;

use Symfony\Component\Config\Loader\Loader;
use Symfony\Component\Routing\RouteCollection;

class AdvancedLoader extends Loader
{
    public function load($resource, $type = null)
    {
        $collection = new RouteCollection();

        $resource = '@AcmeDemoBundle/Resources/config/import_routing.yml';
        $type = 'yaml';

        $importedRoutes = $this->import($resource, $type);

        $collection->addCollection($importedRoutes);

        return $collection;
    }

    public function supports($resource, $type = null)
    {
        return $type === 'advanced_extra';
    }
}
```

Nota: O nome e o tipo do recurso da configuração de roteamento importada pode ser qualquer coisa que é normalmente suportada pelo loader de configuração de roteamento (YAML, XML, PHP, anotação, etc.)

Redirecionar URLs com uma Barra no Final

O objetivo deste cookbook é demonstrar como redirecionar URLs com uma barra no final para a mesma URL sem a barra no final (por exemplo, `/en/blog/` para `/en/blog`).

Crie um controlador que irá corresponder a qualquer URL com uma barra no final, remova a barra do final (mantendo os parâmetros de consulta, se houver) e redirecione para a nova URL com um código de status 301 de resposta:

```
// src/Acme/DemoBundle/Controller/RedirectingController.php
namespace Acme\DemoBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Request;

class RedirectingController extends Controller
{
    public function removeTrailingSlashAction(Request $request)
    {
        $pathInfo = $request->getPathInfo();
        $requestUri = $request->getRequestUri();

        $url = str_replace($pathInfo, rtrim($pathInfo, ' /'), $requestUri);

        return $this->redirect($url, 301);
    }
}
```

Depois disso, crie uma rota para este controlador que corresponde sempre que uma URL com uma barra no final é solicitada. Não se esqueça de colocar esta rota por último no seu sistema, como explicado a seguir:

- *YAML*

```
remove_trailing_slash:
    path: /{url}
    defaults: { _controller: AcmeDemoBundle:Redirecting:removeTrailingSlash }
    requirements:
        url: .*/$
        _method: GET
```

- *XML*

```
<?xml version="1.0" encoding="UTF-8" ?>
<routes xmlns="http://symfony.com/schema/routing">
    <route id="remove_trailing_slash" path="/{url}">
        <default key="_controller">AcmeDemoBundle:Redirecting:removeTrailingSlash</default>
        <requirement key="url">.*/$</requirement>
        <requirement key="_method">GET</requirement>
    </route>
</routes>
```

- *PHP*

```
use Symfony\Component\Routing\RouteCollection;
use Symfony\Component\Routing\Route;
```

```
    $collection = new RouteCollection();
    $collection->add(
        'remove_trailing_slash',
        new Route(
            '/{url}',
            array(
                '_controller' => 'AcmeDemoBundle:Redirecting:removeTrailingSlash',
            ),
            array(
                'url' => '.*/$',
                '_method' => 'GET',
            )
        )
    );
```

Nota: Redirecionar uma solicitação POST não funciona bem em navegadores antigos. Um 302 em uma solicitação POST iria enviar uma solicitação GET após o redirecionamento por motivos legados. Por essa razão, a rota aqui corresponde apenas solicitações GET.

Cuidado: Certifique-se de incluir esta rota em sua configuração de roteamento no final de sua lista de rotas. Caso contrário, você corre o risco de redirecionar rotas reais (incluindo as rotas principais do Symfony2) que realmente *têm* uma barra no final do seu caminho.

3.1.4 Assetic

Como usar o Assetic para o Gerenciamento de Assets

O Assetic combina duas idéias principais: assets e filtros. Os assets são arquivos CSS, JavaScript e arquivos de imagem. Os filtros são coisas que podem ser aplicadas à esses arquivos antes deles serem servidos ao navegador. Isto permite uma separação entre os arquivos asset armazenados na aplicação e os arquivos que são efetivamente apresentados ao usuário.

Sem o Assetic, você somente poderia servir os arquivos que são armazenados diretamente na aplicação:

- *Twig*

```
    <script src="{{ asset('js/script.js') }}" type="text/javascript" />
```

- *PHP*

```
    <script src="<?php echo $view['assets']->getUrl('js/script.js') ?>" type="text/javascript" />
```

Mas *com* o Assetic, você pode manipular esses assets da forma que desejar (ou carregá-los de qualquer lugar) antes de serví-los. Isto significa que você pode:

- Minificar e combinar todos os seus arquivos CSS e JS
- Executar todos (ou apenas alguns) dos seus arquivos CSS ou JS através de algum tipo de compilador, como o LESS, SASS ou CoffeeScript
- Executar otimizações em suas imagens

Assets

O uso do Assetic oferece muitas vantagens sobre servir diretamente os arquivos. Os arquivos não precisam ser armazenados onde eles são servidos e podem ser buscados a partir de várias fontes, como, por exemplo, a partir de um bundle:

- *Twig*

```
{% javascripts '@AcmeFooBundle/Resources/public/js/*' %}
    <script type="text/javascript" src="{{ asset_url }}"></script>
{% endjavascripts %}
```

- *PHP*

```
<?php foreach ($view['assetic']->javascripts(
    array('@AcmeFooBundle/Resources/public/js/*')) as $url): ?>
    <script type="text/javascript" src="<?php echo $view->escape($url) ?>"></script>
<?php endforeach; ?>
```

Dica: Para buscar folhas de estilo CSS, você pode usar as mesmas metodologias vistas aqui, exceto com a tag *stylesheets* :

- *Twig*

```
{% stylesheets '@AcmeFooBundle/Resources/public/css/*' %}
    <link rel="stylesheet" href="{{ asset_url }}" />
{% endstylesheets %}
```

- *PHP*

```
<?php foreach ($view['assetic']->stylesheets(
                                    array('@AcmeFooBundle/Resources/public/css/*')
                                ) as $url): ?>
    <link rel="stylesheet" href="<?php echo $view->escape($url) ?>" />
<?php endforeach; ?>
```

Neste exemplo, todos os arquivos no diretório `Resources/public/js/` do `AcmeFooBundle` serão carregados e servidos em um local diferente. A tag atual renderizada pode parecer simplesmente com:

```
<script src="/app_dev.php/js/abcd123.js"></script>
```

Nota: Este é um ponto-chave: uma vez que você deixar o Assetic lidar com seus assets, os arquivos são servidos a partir de um local diferente. Isto *pode* causar problemas com os arquivos CSS que referenciam imagens pelo seu caminho relativo. No entanto, isso pode ser corrigido usando o filtro `cssrewrite`, que atualiza os caminhos nos arquivos CSS para refletir a sua nova localização.

Combinando Assets Você também pode combinar vários arquivos em um único. Isto ajuda a reduzir o número de solicitações HTTP, o que é ótimo para o desempenho front-end. Também permite que você mantenha os arquivos mais facilmente, dividindo-os em partes gerenciáveis. Isso pode ajudar com a possibilidade de reutilização, uma vez que você pode facilmente dividir os arquivos específicos do projeto daqueles que podem ser usados em outras aplicações, mas ainda serví-los como um único arquivo:

- *Twig*

```
{% javascripts
    '@AcmeFooBundle/Resources/public/js/*'
    '@AcmeBarBundle/Resources/public/js/form.js'
    '@AcmeBarBundle/Resources/public/js/calendar.js' %}
    <script src="{{ asset_url }}"></script>
{% endjavascripts %}
```

- *PHP*

```
<?php foreach ($view['assetic']->javascripts(
    array('@AcmeFooBundle/Resources/public/js/*',
          '@AcmeBarBundle/Resources/public/js/form.js',
          '@AcmeBarBundle/Resources/public/js/calendar.js')) as $url): ?>
    <script src="<?php echo $view->escape($url) ?>"></script>
<?php endforeach; ?>
```

No ambiente *dev*, cada arquivo ainda é servido individualmente, de modo que você pode depurar problemas mais facilmente. No entanto, no ambiente *prod*, serão processados como uma única tag *script*.

Dica: Se você é novo no Assetic e tentar usar a sua aplicação no ambiente `prod` (utilizando o controlador `app.php`), você provavelmente verá que todos os seus CSS e JS estão corrompidos. Não se preocupe! Isso é de propósito. Para detalhes sobre a utilização do Assetic no ambiente *prod*, consulte *Dump dos arquivos de asset*.

E a combinação de arquivos não se aplica apenas para *seus* arquivos. Você também pode usar o Assetic para combinar assets de terceiros, tais como jQuery, como seu próprio em um único arquivo:

- *Twig*

```
{% javascripts
    '@AcmeFooBundle/Resources/public/js/thirdparty/jquery.js'
    '@AcmeFooBundle/Resources/public/js/*' %}
    <script src="{{ asset_url }}"></script>
{% endjavascripts %}
```

- *PHP*

```
<?php foreach ($view['assetic']->javascripts(
    array('@AcmeFooBundle/Resources/public/js/thirdparty/jquery.js',
          '@AcmeFooBundle/Resources/public/js/*')) as $url): ?>
    <script src="<?php echo $view->escape($url) ?>"></script>
<?php endforeach; ?>
```

Filtros

Uma vez que são gerenciados pelo Assetic, você pode aplicar filtros em seus assets antes deles serem servidos. Isso inclui filtros que comprimem a saída de seus assets para tamanhos de arquivos menores (e melhor otimização do front-end). Outros filtros podem compilar os arquivos JavaScript a partir de arquivos CoffeeScript e processar SASS em CSS. Na verdade, o Assetic tem uma longa lista de filtros disponíveis.

Muitos dos filtros não fazem o trabalho diretamente, mas usam bibliotecas existentes de terceiros para fazer o trabalho pesado. Isto significa que muitas vezes você vai precisar instalar uma biblioteca de terceiro para usar um filtro. A grande vantagem de usar o Assetic para chamar estas bibliotecas (em oposição a usá-las diretamente) é que, em vez de ter que executá-las manualmente depois de trabalhar nos arquivos, o Assetic irá cuidar disto para você e remover completamente esta etapa do seu processo de desenvolvimento e implantação.

Para usar um filtro, primeiro você precisa especificá-lo na configuração do Assetic. Adicionar um filtro aqui não significa que ele está sendo usado - apenas significa que está disponível para uso (vamos usar o filtro abaixo).

Por exemplo, para usar o JavaScript YUI Compressor, a configuração seguinte deve ser acrescentada:

- *YAML*

```yaml
# app/config/config.yml
assetic:
    filters:
        yui_js:
            jar: "%kernel.root_dir%/Resources/java/yuicompressor.jar"
```

- *XML*

```xml
<!-- app/config/config.xml -->
<assetic:config>
    <assetic:filter
        name="yui_js"
        jar="%kernel.root_dir%/Resources/java/yuicompressor.jar" />
</assetic:config>
```

- *PHP*

```php
// app/config/config.php
$container->loadFromExtension('assetic', array(
    'filters' => array(
        'yui_js' => array(
            'jar' => '%kernel.root_dir%/Resources/java/yuicompressor.jar',
        ),
    ),
));
```

Agora, para efetivamente *usar* o filtro em um grupo de arquivos JavaScript, adicione-o em seu template:

- *Twig*

```
{% javascripts '@AcmeFooBundle/Resources/public/js/*' filter='yui_js' %}
    <script src="{{ asset_url }}"></script>
{% endjavascripts %}
```

- *PHP*

```
<?php foreach ($view['assetic']->javascripts(
    array('@AcmeFooBundle/Resources/public/js/*'),
    array('yui_js')) as $url): ?>
    <script src="<?php echo $view->escape($url) ?>"></script>
<?php endforeach; ?>
```

Um guia mais detalhado sobre a configuração e uso dos filtros Assetic, bem como detalhes do modo de depuração do Assetic pode ser encontrado em Como Minificar JavaScripts e Folhas de Estilo com o YUI Compressor.

Controlando a URL usada

Se desejar, você pode controlar as URLs que o Assetic produz. Isto é feito a partir do template e é relativo à raiz do documento público:

- *Twig*

```
{% javascripts '@AcmeFooBundle/Resources/public/js/*' output='js/compiled/main.js' %}
    <script src="{{ asset_url }}"></script>
{% endjavascripts %}
```

- *PHP*

```
<?php foreach ($view['assetic']->javascripts(
    array('@AcmeFooBundle/Resources/public/js/*'),
    array(),
    array('output' => 'js/compiled/main.js')
) as $url): ?>
    <script src="<?php echo $view->escape($url) ?>"></script>
<?php endforeach; ?>
```

Nota: O Symfony também contém um método de *busting* de cache, onde a URL final gerada pelo Assetic contém um parâmetro de query, que pode ser incrementado através de configuração em cada implantação. Para mais informações, consulte a opção de configuração ref-framework-assets-version.

Dump dos arquivos de asset

No ambiente `dev`, o Assetic gera caminhos para os arquivos CSS e JavaScript que não existem fisicamente em seu computador. Mas, eles renderizam mesmo assim porque um controlador interno do Symfony abre os arquivos e serve de volta o conteúdo (após a execução de quaisquer filtros).

Este tipo de publicação dinâmica dos assets processados é ótima porque significa que você pode ver imediatamente o novo estado de quaisquer arquivos de assets que foram alterados. Também é ruim, porque pode ser muito lento. Se você estiver usando uma série de filtros, pode ser francamente frustrante.

Felizmente, o Assetic fornece uma forma de fazer o dump de seus assets para arquivos reais, em vez de ser gerado dinamicamente.

Dump dos arquivos asset no ambiente `prod` No ambiente `prod`, seus JS e CSS são representados por uma única tag cada. Em outras palavras, em vez de ver cada arquivo JavaScript que você está incluindo no seu código fonte, é provável que você só veja algo semelhante a:

```
<script src="/app_dev.php/js/abcd123.js"></script>
```

Além disso, esse arquivo **não** existe realmente, nem é renderizado de forma dinâmica pelo Symfony (pois os arquivos de asset estão no ambiente `dev`). Isto é de propósito - deixar o Symfony gerar esses arquivos dinamicamente em um ambiente de produção é muito lento.

Em vez disso, cada vez que você usar a sua aplicação no ambiente "prod " (e, portanto, cada vez que você implantar), você deve executar o seguinte comando:

```
$ php app/console assetic:dump --env=prod --no-debug
```

Isso vai gerar fisicamente e escrever cada arquivo que você precisa (por exemplo, `/js/abcd123.js`). Se você atualizar qualquer um de seus assets, é necessário executar o comando novamente para gerar o novo arquivo.

Dump dos arquivos de asset no ambiente `dev` Por padrão, cada caminho de asset gerado no ambiente `dev` é gerenciado dinamicamente pelo Symfony. Isso não tem desvantagem (você pode ver as suas alterações imediatamente), com exceção de que os assets podem visivelmente carregar mais lentos. Se você sentir que seus assets estão carregando muito lentamente, siga este guia.

Primeiro, diga ao Symfony para parar de tentar processar estes arquivos dinamicamente. Faça a seguinte alteração em seu arquivo `config_dev.yml`:

- *YAML*

```yaml
# app/config/config_dev.yml
assetic:
    use_controller: false
```

- *XML*

```xml
<!-- app/config/config_dev.xml -->
<assetic:config use-controller="false" />
```

- *PHP*

```php
// app/config/config_dev.php
$container->loadFromExtension('assetic', array(
    'use_controller' => false,
));
```

Em seguida, uma vez que o Symfony não está mais gerando esses assets para você, você vai precisar fazer o dump deles manualmente. Para isso, execute o seguinte:

```
$ php app/console assetic:dump
```

Esta fisicamente grava todos os arquivos ativos que você precisa para seu ‘‘ ‘‘ dev produção. A grande desvantagem é que você precisa executar este cada vez você atualizar um ativo. Felizmente, passando o ‘‘ - assistir opção ‘‘, o comando automaticamente regenerar ativos * como eles mudam *:

```
$ php app/console assetic:dump --watch
```

Uma vez que executar este comando no ambiente `dev` pode gerar vários arquivos, geralmente é uma boa idéia apontar os seus arquivos assets gerados para algum diretório isolado (por exemplo, `/js/compiled`), para manter as coisas organizadas:

- *Twig*

```
{% javascripts '@AcmeFooBundle/Resources/public/js/*' output='js/compiled/main.js' %}
    <script src="{{ asset_url }}"></script>
{% endjavascripts %}
```

- *PHP*

```
<?php foreach ($view['assetic']->javascripts(
    array('@AcmeFooBundle/Resources/public/js/*'),
    array(),
    array('output' => 'js/compiled/main.js')
) as $url): ?>
    <script src="<?php echo $view->escape($url) ?>"></script>
<?php endforeach; ?>
```

Como Minificar JavaScripts e Folhas de Estilo com o YUI Compressor

A Yahoo! oferece um excelente utilitário, chamado YUI Compressor, para minificar JavaScripts e folhas de estilo, assim, eles são carregados mais rapidamente. Graças ao Assetic, você pode tirar proveito desta ferramenta de forma muito fácil.

Baixe o JAR do YUI Compressor

O YUI Compressor é escrito em Java e distribuído como um JAR. Faça o download do JAR no site da Yahoo! e salve-o em `app/Resources/java/yuicompressor.jar`.

Configure os Filtros do YUI

Agora você precisa configurar dois filtros Assetic em sua aplicação, um para minificar os JavaScripts com o YUI Compressor e um para minificar as folhas de estilo:

- *YAML*

```yaml
# app/config/config.yml
assetic:
    filters:
        yui_css:
            jar: "%kernel.root_dir%/Resources/java/yuicompressor.jar"
        yui_js:
            jar: "%kernel.root_dir%/Resources/java/yuicompressor.jar"
```

- *XML*

```xml
<!-- app/config/config.xml -->
<assetic:config>
    <assetic:filter
        name="yui_css"
        jar="%kernel.root_dir%/Resources/java/yuicompressor.jar" />
    <assetic:filter
        name="yui_js"
        jar="%kernel.root_dir%/Resources/java/yuicompressor.jar" />
</assetic:config>
```

- *PHP*

```php
// app/config/config.php
$container->loadFromExtension('assetic', array(
    'filters' => array(
        'yui_css' => array(
            'jar' => '%kernel.root_dir%/Resources/java/yuicompressor.jar',
        ),
        'yui_js' => array(
            'jar' => '%kernel.root_dir%/Resources/java/yuicompressor.jar',
        ),
    ),
));
```

Você agora tem acesso a dois novos filtros Assetic em sua aplicação: `yui_css` e `yui_js`. Eles utilizarão o YUI Compressor para minificar as folhas de estilo e JavaScripts, respectivamente.

Minifique os seus Assets

Você agora tem o YUI Compressor configurado, mas nada vai acontecer até aplicar um desses filtros para um asset. Uma vez que os seus assets fazem parte da camada de visão, este trabalho é feito em seus templates:

- *Twig*

```
{% javascripts '@AcmeFooBundle/Resources/public/js/*' filter='yui_js' %}
<script src="{{ asset_url }}"></script>
{% endjavascripts %}
```

- *PHP*

```
<?php foreach ($view['assetic']->javascripts(
    array('@AcmeFooBundle/Resources/public/js/*'),
    array('yui_js')) as $url): ?>
<script src="<?php echo $view->escape($url) ?>"></script>
<?php endforeach; ?>
```

Nota: O exemplo acima assume que você tem um bundle chamado `AcmeFooBundle` e os seus arquivos JavaScript estão no diretório‘‘Resources/public/js‘‘ sob o seu bundle. Entretante, isso não é importante - você pode incluir os seus arquivos JavaScript, não importa onde eles estiverem.

Com a adição do filtro `yui_js` para as tags asset acima, você deve agora ver os JavaScripts minificados sendo carregados muito mais rápido. O mesmo processo pode ser repetido para minificar as suas folhas de estilo.

- *Twig*

```
{% stylesheets '@AcmeFooBundle/Resources/public/css/*' filter='yui_css' %}
<link rel="stylesheet" type="text/css" media="screen" href="{{ asset_url }}" />
{% endstylesheets %}
```

- *PHP*

```
<?php foreach ($view['assetic']->stylesheets(
    array('@AcmeFooBundle/Resources/public/css/*'),
    array('yui_css')) as $url): ?>
<link rel="stylesheet" type="text/css" media="screen" href="<?php echo $view->escape($url) ?>" /
<?php endforeach; ?>
```

Desative a minificação no modo de depuração

Os JavaScripts e as folhas de estilo minificados são muito difíceis de ler, e muito menos depurar. Devido a isso, o Assetic permite desabilitar um certo filtro quando a sua aplicação está no modo de depuração. Você pode fazer isso prefixando o nome do filtro em seu template com um ponto de interrogação: `?`. Isto diz ao Assetic para apenas aplicar esse filtro quando o modo de depuração está desligado.

- *Twig*

```
{% javascripts '@AcmeFooBundle/Resources/public/js/*' filter='?yui_js' %}
<script src="{{ asset_url }}"></script>
{% endjavascripts %}
```

- *PHP*

```
<?php foreach ($view['assetic']->javascripts(
    array('@AcmeFooBundle/Resources/public/js/*'),
    array('?yui_js')) as $url): ?>
<script src="<?php echo $view->escape($url) ?>"></script>
<?php endforeach; ?>
```

Como usar o Assetic para otimização de imagem com funções do Twig

Dentre os seus vários filtros, o Assetic possui quatro que podem ser utilizados para a otimização de imagens on-the-fly. Isso permite obter os benefícios de tamanhos menores dos arquivos sem ter que usar um editor de imagens para processar cada imagem. Os resultados são armazenados em cache e pode ser feito o dump para produção de modo que não há impacto no desempenho para seus usuários finais.

Usando o jpegoptim

Jpegoptim é um utilitário para otimizar arquivos JPEG. Para usá-lo com o Assetic, adicione o seguinte na configuração do Assetic:

- *YAML*

```
# app/config/config.yml
assetic:
    filters:
        jpegoptim:
            bin: path/to/jpegoptim
```

- *XML*

```
<!-- app/config/config.xml -->
<assetic:config>
    <assetic:filter
        name="jpegoptim"
        bin="path/to/jpegoptim" />
</assetic:config>
```

- *PHP*

```
// app/config/config.php
$container->loadFromExtension('assetic', array(
    'filters' => array(
        'jpegoptim' => array(
            'bin' => 'path/to/jpegoptim',
        ),
    ),
));
```

Nota: Observe que, para usar o jpegoptim, você deve instalá-lo em seu sistema. A opção `bin` aponta para a localização do binário compilado.

Ele agora pode ser usado em um template:

- *Twig*

```
{% image '@AcmeFooBundle/Resources/public/images/example.jpg'
    filter='jpegoptim' output='/images/example.jpg'
%}
<img src="{{ asset_url }}" alt="Example"/>
{% endimage %}
```

- *PHP*

```
<?php foreach ($view['assetic']->images(
    array('@AcmeFooBundle/Resources/public/images/example.jpg'),
    array('jpegoptim')) as $url): ?>
<img src="<?php echo $view->escape($url) ?>" alt="Example"/>
<?php endforeach; ?>
```

Removendo todos os dados EXIF Por padrão, a execução desse filtro remove apenas algumas das informações meta armazenadas no arquivo. Os dados EXIF e comentários não são removidos, mas você pode removê-los usando a opção `strip_all`:

- *YAML*

```
# app/config/config.yml
assetic:
    filters:
        jpegoptim:
            bin: path/to/jpegoptim
            strip_all: true
```

- *XML*

```
<!-- app/config/config.xml -->
<assetic:config>
    <assetic:filter
        name="jpegoptim"
        bin="path/to/jpegoptim"
        strip_all="true" />
</assetic:config>
```

- *PHP*

```
// app/config/config.php
$container->loadFromExtension('assetic', array(
    'filters' => array(
        'jpegoptim' => array(
            'bin' => 'path/to/jpegoptim',
            'strip_all' => 'true',
        ),
    ),
));
```

Diminuindo a qualidade máxima Por padrão, o nível de qualidade do JPEG não é afetado. Você pode ganhar reduções adicionais no tamanho dos arquivos ao ajustar a configuração de qualidade máxima para um valor inferior ao nível atual das imagens. Isto irá, claro, custar a qualidade de imagem:

- *YAML*

```
# app/config/config.yml
assetic:
    filters:
        jpegoptim:
            bin: path/to/jpegoptim
            max: 70
```

- *XML*

```
<!-- app/config/config.xml -->
<assetic:config>
    <assetic:filter
        name="jpegoptim"
        bin="path/to/jpegoptim"
        max="70" />
</assetic:config>
```

- *PHP*

```
// app/config/config.php
$container->loadFromExtension('assetic', array(
    'filters' => array(
        'jpegoptim' => array(
            'bin' => 'path/to/jpegoptim',
            'max' => '70',
        ),
    ),
));
```

Sintaxe curta: Função Twig

Se você estiver usando o Twig, é possível conseguir tudo isso com uma sintaxe curta, ao habilitar e usar uma função especial do Twig. Comece adicionando a seguinte configuração:

- *YAML*

```
# app/config/config.yml
assetic:
    filters:
        jpegoptim:
            bin: path/to/jpegoptim
    twig:
        functions:
            jpegoptim: ~
```

- *XML*

```
<!-- app/config/config.xml -->
<assetic:config>
    <assetic:filter
        name="jpegoptim"
        bin="path/to/jpegoptim" />
    <assetic:twig>
        <assetic:twig_function
            name="jpegoptim" />
    </assetic:twig>
</assetic:config>
```

- *PHP*

```
// app/config/config.php
$container->loadFromExtension('assetic', array(
    'filters' => array(
        'jpegoptim' => array(
            'bin' => 'path/to/jpegoptim',
        ),
    ),
```

```
        'twig' => array(
            'functions' => array('jpegoptim'),
            ),
        ),
    ));
```

O template Twig pode agora ser alterado para o seguinte:

```
<img src="{{ jpegoptim('@AcmeFooBundle/Resources/public/images/example.jpg') }}"
     alt="Example"/>
```

Você pode especificar o diretório de saída na configuração da seguinte forma:

- *YAML*

```
    # app/config/config.yml
    assetic:
        filters:
            jpegoptim:
                bin: path/to/jpegoptim
        twig:
            functions:
                jpegoptim: { output: images/*.jpg }
```

- *XML*

```
    <!-- app/config/config.xml -->
    <assetic:config>
        <assetic:filter
            name="jpegoptim"
            bin="path/to/jpegoptim" />
        <assetic:twig>
            <assetic:twig_function
                name="jpegoptim"
                output="images/*.jpg" />
        </assetic:twig>
    </assetic:config>
```

- *PHP*

```
    // app/config/config.php
    $container->loadFromExtension('assetic', array(
        'filters' => array(
            'jpegoptim' => array(
                'bin' => 'path/to/jpegoptim',
            ),
        ),
        'twig' => array(
            'functions' => array(
                'jpegoptim' => array(
                    output => 'images/*.jpg'
                ),
            ),
        ),
    ));
```

Como Aplicar um filtro Assetic a uma extensão de arquivo específica

Os filtros `Assetic` podem ser aplicados à arquivos individuais, grupos de arquivos ou até mesmo, como você verá aqui, aos arquivos que possuem uma extensão específica. Para mostrar como lidar com cada opção, vamos supor que você deseja usar o filtro `CoffeeScript` do `Assetic`, que compila arquivos `CoffeeScript` em Javascript.

A configuração principal é apenas os caminhos para `coffee` e `node`. Elas apontam, por padrão, para `/usr/bin/coffee` e `/usr/bin/node`, respectivamente:

- *YAML*

```yaml
# app/config/config.yml
assetic:
    filters:
        coffee:
            bin: /usr/bin/coffee
            node: /usr/bin/node
```

- *XML*

```xml
<!-- app/config/config.xml -->
<assetic:config>
    <assetic:filter
        name="coffee"
        bin="/usr/bin/coffee"
        node="/usr/bin/node" />
</assetic:config>
```

- *PHP*

```php
// app/config/config.php
$container->loadFromExtension('assetic', array(
    'filters' => array(
        'coffee' => array(
            'bin' => '/usr/bin/coffee',
            'node' => '/usr/bin/node',
        ),
    ),
));
```

Filtrando um único arquivo

Agora, você pode servir um único arquivo `CoffeeScript` como JavaScript a partir de seus templates:

- *Twig*

```
{% javascripts '@AcmeFooBundle/Resources/public/js/example.coffee'
    filter='coffee'
%}
<script src="{{ asset_url }} type="text/javascript"></script>
{% endjavascripts %}
```

- *PHP*

```
<?php foreach ($view['assetic']->javascripts(
    array('@AcmeFooBundle/Resources/public/js/example.coffee'),
    array('coffee')) as $url): ?>
<script src="<?php echo $view->escape($url) ?>" type="text/javascript"></script>
<?php endforeach; ?>
```

Isso é tudo o que é necessário para compilar este arquivo `CoffeeScript` e servir ele como JavaScript compilado.

Filtrando vários arquivos

Você também pode combinar vários arquivos `CoffeeScript` em um único arquivo de saída:

- *Twig*

```
{% javascripts '@AcmeFooBundle/Resources/public/js/example.coffee'
               '@AcmeFooBundle/Resources/public/js/another.coffee'
    filter='coffee'
%}
<script src="{{ asset_url }} type="text/javascript"></script>
{% endjavascripts %}
```

- *PHP*

```
<?php foreach ($view['assetic']->javascripts(
    array('@AcmeFooBundle/Resources/public/js/example.coffee',
          '@AcmeFooBundle/Resources/public/js/another.coffee'),
    array('coffee')) as $url): ?>
<script src="<?php echo $view->escape($url) ?>" type="text/javascript"></script>
<?php endforeach; ?>
```

Ambos os arquivos agora serão servidos como um único arquivo compilado em JavaScript regular.

Filtrando com base em uma extensão de arquivo

Uma das grandes vantagens de usar o `Assetic` é minimizar o número de arquivos `asset` para reduzir as solicitações HTTP. A fim de fazer seu pleno uso, seria bom combinar *todos* os seus arquivos JavaScript e `CoffeeScript` juntos, uma vez que, todos serão servidos como JavaScript. Infelizmente, apenas adicionar os arquivos JavaScript aos arquivos combinados como acima não funcionará, pois, os arquivos JavaScript regulares não sobreviverão a compilação do `CoffeeScript`.

Este problema pode ser evitado usando a opção `apply_to` na configuração, que permite especificar que filtro deverá ser sempre aplicado à determinadas extensões de arquivo. Neste caso, você pode especificar que o filtro `Coffee` será aplicado à todos os arquivos `.coffee`:

- *YAML*

```
# app/config/config.yml
assetic:
    filters:
        coffee:
            bin: /usr/bin/coffee
            node: /usr/bin/node
            apply_to: "\.coffee$"
```

- *XML*

```
<!-- app/config/config.xml -->
<assetic:config>
    <assetic:filter
        name="coffee"
        bin="/usr/bin/coffee"
        node="/usr/bin/node"
        apply_to="\.coffee$" />
</assetic:config>
```

- *PHP*

```
// app/config/config.php
$container->loadFromExtension('assetic', array(
    'filters' => array(
        'coffee' => array(
            'bin' => '/usr/bin/coffee',
            'node' => '/usr/bin/node',
            'apply_to' => '\.coffee$',
        ),
    ),
));
```

Com isso, você não precisa especificar o filtro `coffee` no template. Você também pode listar os arquivos JavaScript regulares, todos os quais serão combinados e renderizados como um único arquivo JavaScript (apenas com os arquivos `.coffee` sendo executados através do filtro `CoffeeScript`):

- *Twig*

```
{% javascripts '@AcmeFooBundle/Resources/public/js/example.coffee'
               '@AcmeFooBundle/Resources/public/js/another.coffee'
               '@AcmeFooBundle/Resources/public/js/regular.js'
%}
<script src="{{ asset_url }} type="text/javascript"></script>
{% endjavascripts %}
```

- *PHP*

```
<?php foreach ($view['assetic']->javascripts(
    array('@AcmeFooBundle/Resources/public/js/example.coffee',
          '@AcmeFooBundle/Resources/public/js/another.coffee',
          '@AcmeFooBundle/Resources/public/js/regular.js'),
    as $url): ?>
<script src="<?php echo $view->escape($url) ?>" type="text/javascript"></script>
<?php endforeach; ?>
```

3.1.5 Doctrine

Como Manipular o Upload de Arquivos com o Doctrine

Gerenciar o upload de arquivos utilizando entidades do Doctrine não é diferente de manusear qualquer outro upload de arquivo. Em outras palavras, você é livre para mover o arquivo em seu controlador após a manipulação do envio de um formulário. Para exemplos de como fazer isso, veja a página de `referência do tipo arquivo`.

Se você quiser, também pode integrar o upload de arquivo no ciclo de vida de sua entidade (ou seja, criação, atualização e remoção). Neste caso, como a sua entidade é criada, atualizada e removida pelo Doctrine, o tratamento do upload e da remoção de arquivos será realizado automaticamente (sem precisar fazer nada em seu controlador);

Para fazer este trabalho, você precisa cuidar de uma série de detalhes, que serão abordados neste artigo do cookbook.

Configuração Básica

Primeiro, crie uma classe Entity simples do Doctrine para você trabalhar:

```
// src/Acme/DemoBundle/Entity/Document.php
namespace Acme\DemoBundle\Entity;
```

```
use Doctrine\ORM\Mapping as ORM;
use Symfony\Component\Validator\Constraints as Assert;

/**
 * @ORM\Entity
 */
class Document
{
    /**
     * @ORM\Id
     * @ORM\Column(type="integer")
     * @ORM\GeneratedValue(strategy="AUTO")
     */
    public $id;

    /**
     * @ORM\Column(type="string", length=255)
     * @Assert\NotBlank
     */
    public $name;

    /**
     * @ORM\Column(type="string", length=255, nullable=true)
     */
    public $path;

    public function getAbsolutePath()
    {
        return null === $this->path
            ? null
            : $this->getUploadRootDir().'/'.$this->path;
    }

    public function getWebPath()
    {
        return null === $this->path
            ? null
            : $this->getUploadDir().'/'.$this->path;
    }

    protected function getUploadRootDir()
    {
        // the absolute directory path where uploaded
        // documents should be saved
        return __DIR__.'/../../../../web/'.$this->getUploadDir();
    }

    protected function getUploadDir()
    {
        // get rid of the __DIR__ so it doesn't screw up
        // when displaying uploaded doc/image in the view.
        return 'uploads/documents';
    }
}
```

A entidade `Document` tem um nome e ele é associado a um arquivo. A propriedade `path` armazena o caminho relativo para o arquivo e é persistida no banco de dados. O `getAbsolutePath()` é um método de conveniência que retorna o caminho absoluto para o arquivo enquanto o `getWebPath()` é um método de conveniência que retorna

o caminho web, que podem ser utilizados em um template para obter o link do arquivo que foi feito o upload.

Dica: Se não tiver feito isso, você deve ler primeiro a documentação sobre o tipo arquivo `file` para entender como funciona o processo básico de upload.

Nota: Se você estiver usando anotações para especificar as suas regras de validação (como mostrado neste exemplo), certifique-se de que tenha ativado a validação por anotação (veja *configuração de validação*).

Para lidar com o upload do arquivo no formulário, use um campo "virtual" `file`. Por exemplo, se você está construindo o seu formulário diretamente em um controlador, ele poderia parecer com o seguinte:

```
public function uploadAction()
{
    // ...

    $form = $this->createFormBuilder($document)
        ->add('name')
        ->add('file')
        ->getForm();

    // ...
}
```

Em seguida, crie essa propriedade em sua classe `Document` e adicione algumas regras de validação:

```
// src/Acme/DemoBundle/Entity/Document.php

// ...
class Document
{
    /**
     * @Assert\File(maxSize="6000000")
     */
    public $file;

    // ...
}
```

Nota: Como você está usando a constraint `File`, o Symfony2 irá "adivinhar" automaticamente que o campo do formulário é do tipo para upload de arquivos. É por isso que você não tem que defini-lo explicitamente ao criar o formulário acima (`->add('file')`).

O controlador a seguir mostra como lidar com todo o processo:

```
use Acme\DemoBundle\Entity\Document;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Template;
// ...

/**
 * @Template()
 */
public function uploadAction()
{
    $document = new Document();
```

```
    $form = $this->createFormBuilder($document)
        ->add('name')
        ->add('file')
        ->getForm()
    ;

    if ($this->getRequest()->isMethod('POST')) {
        $form->bind($this->getRequest());
        if ($form->isValid()) {
            $em = $this->getDoctrine()->getManager();

            $em->persist($document);
            $em->flush();

            $this->redirect($this->generateUrl(...));
        }
    }

    return array('form' => $form->createView());
}
```

Nota: Ao escrever o template, não esqueça de definir o atributo `enctype`:

```
<h1>Upload File</h1>

<form action="#" method="post" {{ form_enctype(form) }}>
    {{ form_widget(form) }}

    <input type="submit" value="Upload Document" />
</form>
```

O controlador anterior irá persistir automaticamente a entidade `Document` com o nome submetido, mas ele não fará nada a respeito do arquivo e a propriedade `path` ficará em branco.

Uma maneira fácil de lidar com o upload do arquivo é movê-lo pouco antes da entidade ser persistida e, em seguida, definir a propriedade `path` de acordo. Comece chamando o novo método `upload()` na classe `Document`, que você vai criar no momento para lidar com o upload do arquivo:

```
if ($form->isValid()) {
    $em = $this->getDoctrine()->getManager();

    $document->upload();

    $em->persist($document);
    $em->flush();

    $this->redirect(...);
}
```

O método `upload()` irá aproveitar o objeto `UploadedFile` , que é o retornado após um campo `file` ser submetido:

```
public function upload()
{
    // the file property can be empty if the field is not required
    if (null === $this->file) {
        return;
```

```
    }

    // use the original file name here but you should
    // sanitize it at least to avoid any security issues

    // move takes the target directory and then the
    // target filename to move to
    $this->file->move(
        $this->getUploadRootDir(),
        $this->file->getClientOriginalName()
    );

    // set the path property to the filename where you've saved the file
    $this->path = $this->file->getClientOriginalName();

    // clean up the file property as you won't need it anymore
    $this->file = null;
}
```

Utilizando Lifecycle Callbacks

Mesmo esta aplicação funcionando, ela sofre de uma grande falha: E se houver um problema quando a entidade for persistida? O arquivo já teria sido movido para seu local definitivo, apesar da propriedade `path` da entidade não ter sido persistida corretamente.

Para evitar esses problemas, você deve alterar a implementação de forma que as operações do banco de dados e a cópia do arquivo tornem-se atômicas: se há um problema persistindo a entidade ou se o arquivo não pode ser movido, então *nada* deve ser feito.

Para fazer isso, você precisa mover o arquivo no mesmo momento em que o Doctrine persistir a entidade no banco de dados. Isto pode ser feito lifecycle da entidade:

```
/**
 * @ORM\Entity
 * @ORM\HasLifecycleCallbacks
 */
class Document
{
}
```

Em seguida, refatore a classe `Document` para aproveitar esses callbacks:

```
use Symfony\Component\HttpFoundation\File\UploadedFile;

/**
 * @ORM\Entity
 * @ORM\HasLifecycleCallbacks
 */
class Document
{
    /**
     * @ORM\PrePersist()
     * @ORM\PreUpdate()
     */
    public function preUpload()
    {
        if (null !== $this->file) {
```

```
            // do whatever you want to generate a unique name
            $filename = sha1(uniqid(mt_rand(), true));
            $this->path = $filename.'.'.$this->file->guessExtension();
        }
    }

    /**
     * @ORM\PostPersist()
     * @ORM\PostUpdate()
     */
    public function upload()
    {
        if (null === $this->file) {
            return;
        }

        // if there is an error when moving the file, an exception will
        // be automatically thrown by move(). This will properly prevent
        // the entity from being persisted to the database on error
        $this->file->move($this->getUploadRootDir(), $this->path);

        unset($this->file);
    }

    /**
     * @ORM\PostRemove()
     */
    public function removeUpload()
    {
        if ($file = $this->getAbsolutePath()) {
            unlink($file);
        }
    }
}
```

A classe agora faz tudo o que você precisa: ela gera um nome de arquivo único antes de persistir, move o arquivo depois de persistir e remove o arquivo sempre que a entidade for excluída.

Agora que a cópia do arquivo é tratada atomicamente pela entidade, a chamada `$document->upload()` deve ser removida do controlador:

```
if ($form->isValid()) {
    $em = $this->getDoctrine()->getManager();

    $em->persist($document);
    $em->flush();

    $this->redirect(...);
}
```

Nota: Os callbacks dos eventos `@ORM\PrePersist()` e `@ORM\PostPersist()` são acionados antes e depois da entidade ser persistida no banco de dados. Por outro lado, a callback dos eventos `@ORM\PreUpdate()` e `@ORM\PostUpdate()` são chamadas quando a entidade é atualizada.

Cuidado: As callbacks `PreUpdate` e `PostUpdate` são acionadas somente se houver uma alteração em um dos campos de uma entidade que é persistida. Isto significa que, por padrão, se você modificar apenas a propriedade `$file`, esses eventos não serão disparados, pois a propriedade não é diretamente persistida via Doctrine. Uma solução seria a utilização de um campo `updated` que é persistido pelo Doctrine e modificá-lo manualmente quando alterar o arquivo.

Usando o `id` como nome do arquivo

Se você quiser usar o `id` como nome do arquivo, a implementação é ligeiramente diferente pois você precisa salvar a extensão na propriedade `path`, em vez do nome real:

```
use Symfony\Component\HttpFoundation\File\UploadedFile;

/**
 * @ORM\Entity
 * @ORM\HasLifecycleCallbacks
 */
class Document
{
    // a property used temporarily while deleting
    private $filenameForRemove;

    /**
     * @ORM\PrePersist()
     * @ORM\PreUpdate()
     */
    public function preUpload()
    {
        if (null !== $this->file) {
            $this->path = $this->file->guessExtension();
        }
    }

    /**
     * @ORM\PostPersist()
     * @ORM\PostUpdate()
     */
    public function upload()
    {
        if (null === $this->file) {
            return;
        }

        // you must throw an exception here if the file cannot be moved
        // so that the entity is not persisted to the database
        // which the UploadedFile move() method does
        $this->file->move(
            $this->getUploadRootDir(),
            $this->id.'.'.$this->file->guessExtension()
        );

        unset($this->file);
    }

    /**
     * @ORM\PreRemove()
```

```
     */
    public function storeFilenameForRemove()
    {
        $this->filenameForRemove = $this->getAbsolutePath();
    }

    /**
     * @ORM\PostRemove()
     */
    public function removeUpload()
    {
        if ($this->filenameForRemove) {
            unlink($this->filenameForRemove);
        }
    }

    public function getAbsolutePath()
    {
        return null === $this->path
            ? null
            : $this->getUploadRootDir().'/'.$this->id.'.'.$this->path;
    }
}
```

Você vai notar que, neste caso, é necessário um pouco mais de trabalho a fim de remover o arquivo. Antes que seja removido, você deve armazenar o caminho do arquivo (pois ele depende do id). Então, uma vez que o objeto foi totalmente removido do banco de dados, você pode apagar o arquivo com segurança (em `PostRemove`).

Como usar as extens?es do Doctrine: Timestampable, Sluggable, Translatable, etc.

O Doctrine2 ? muito flex?vel e a comunidade j? criou um s?rie de extens?es do Doctrine ?teis para ajudar voc? com tarefas comuns relacionadas a entidades.

Uma biblioteca em particular - a biblioteca DoctrineExtensions - fornece funcionalidade de integra??o para os comportamentos Sluggable, Translatable, Timestampable, Loggable, Tree e Sortable.

O uso de cada uma destas extens?es ? explicado no reposit?rio.

No entanto, para instalar/ativar cada extens?o voc? deve se registrar e ativar um Listener de Evento. Para fazer isso, voc? tem duas op??es:

1. Usar o StofDoctrineExtensionsBundle, que integra a biblioteca acima.
2. Implementar este servi?o diretamente seguindo a documenta??o para a integra??o com o Symfony2: Instalando extens?es Gedmo Doctrine2 no Symfony2

Como Registrar Ouvintes e Assinantes de Eventos

O Doctrine contém um valioso sistema de evento que dispara eventos quando quase tudo acontece dentro do sistema. Para você, isso significa que poderá criar serviços arbitrários e dizer ao Doctrine para notificar os objetos sempre que uma determinada ação (ex. `prePersist`) acontecer. Isto pode ser útil, por exemplo, para criar um índice de pesquisa independente sempre que um objeto em seu banco de dados for salvo.

O Doctrine define dois tipos de objetos que podem ouvir eventos do Doctrine: ouvintes e assinantes. Ambos são muito semelhantes, mas os ouvintes são um pouco mais simples. Para saber mais, consulte O Sistema de Eventos no site do Doctrine.

Configurando o Ouvinte/Assinante

Para registrar um serviço para agir como um ouvinte ou assinante de evento você só tem que usar a *tag* com o nome apropriado. Dependendo de seu caso de uso, você pode ligar um ouvinte em cada conexão DBAL e gerenciador de entidade ORM ou apenas em uma conexão específica DBAL e todos os gerenciadores de entidades que usam esta conexão.

- *YAML*

```yaml
doctrine:
    dbal:
        default_connection: default
        connections:
            default:
                driver: pdo_sqlite
                memory: true

services:
    my.listener:
        class: Acme\SearchBundle\EventListener\SearchIndexer
        tags:
            - { name: doctrine.event_listener, event: postPersist }
    my.listener2:
        class: Acme\SearchBundle\EventListener\SearchIndexer2
        tags:
            - { name: doctrine.event_listener, event: postPersist, connection: default }
    my.subscriber:
        class: Acme\SearchBundle\EventListener\SearchIndexerSubscriber
        tags:
            - { name: doctrine.event_subscriber, connection: default }
```

- *XML*

```xml
<?xml version="1.0" ?>
<container xmlns="http://symfony.com/schema/dic/services"
    xmlns:doctrine="http://symfony.com/schema/dic/doctrine">

    <doctrine:config>
        <doctrine:dbal default-connection="default">
            <doctrine:connection driver="pdo_sqlite" memory="true" />
        </doctrine:dbal>
    </doctrine:config>

    <services>
        <service id="my.listener" class="Acme\SearchBundle\EventListener\SearchIndexer">
            <tag name="doctrine.event_listener" event="postPersist" />
        </service>
        <service id="my.listener2" class="Acme\SearchBundle\EventListener\SearchIndexer2">
            <tag name="doctrine.event_listener" event="postPersist" connection="default" />
        </service>
        <service id="my.subscriber" class="Acme\SearchBundle\EventListener\SearchIndexerSubscrib
            <tag name="doctrine.event_subscriber" connection="default" />
        </service>
    </services>
</container>
```

Criando a Classe Ouvinte

No exemplo anterior, um serviço `my.listener` foi configurado como um ouvinte Doctrine no evento `postPersist`. A classe atrás desse serviço deve ter um método `postPersist`, que será chamado quando o evento é lançado:

```
// src/Acme/SearchBundle/EventListener/SearchIndexer.php
namespace Acme\SearchBundle\EventListener;

use Doctrine\ORM\Event\LifecycleEventArgs;
use Acme\StoreBundle\Entity\Product;

class SearchIndexer
{
    public function postPersist(LifecycleEventArgs $args)
    {
        $entity = $args->getEntity();
        $entityManager = $args->getEntityManager();

        // perhaps you only want to act on some "Product" entity
        if ($entity instanceof Product) {
            // do something with the Product
        }
    }
}
```

Em cada evento, você tem acesso a um objeto `LifecycleEventArgs`, que dá acesso tanto ao objeto entidade do evento quanto ao gerenciador de entidade em si.

Algo importante a notar é que um ouvinte estará ouvindo *todas* as entidades em sua aplicação. Então, se você está interessado apenas em lidar com um tipo específico de entidade (por exemplo, uma entidade `Product`, mas não uma entidade `BlogPost`), você deve verificar o nome da classe da entidade em seu método (como mostrado acima).

Como usar a Camada DBAL do Doctrine

Nota: Este artigo é sobre a camada DBAL do Doctrine. Normalmente, você vai trabalhar com a camada de alto nível ORM do Doctrine, que simplesmente usa o DBAL, nos bastidores, para comunicar-se com o banco de dados. Para ler mais sobre o ORM Doctrine, consulte "Bancos de Dados e Doctrine".

A Camada de Abstração de Banco de Dados Doctrine (DBAL) é uma camada de abstração que fica situada no topo do PDO e oferece uma API intuitiva e flexível para se comunicar com os bancos de dados relacionais mais populares. Em outras palavras, a biblioteca DBAL torna mais fácil a execução de consultas e de outras ações de banco de dados.

Dica: Leia a Documentação oficial do DBAL para aprender todos os detalhes e as capacidades da biblioteca DBAL do Doctrine.

Para começar, configure os parâmetros de conexão do banco de dados:

- *YAML*

```
    # app/config/config.yml
    doctrine:
        dbal:
```

```
            driver:   pdo_mysql
            dbname:   Symfony2
            user:     root
            password: null
            charset:  UTF8
```

- *XML*

```
    // app/config/config.xml
    <doctrine:config>
        <doctrine:dbal
            name="default"
            dbname="Symfony2"
            user="root"
            password="null"
            driver="pdo_mysql"
        />
    </doctrine:config>
```

- *PHP*

```
    // app/config/config.php
    $container->loadFromExtension('doctrine', array(
        'dbal' => array(
            'driver'    => 'pdo_mysql',
            'dbname'    => 'Symfony2',
            'user'      => 'root',
            'password'  => null,
        ),
    ));
```

Para ver todas as opções de configuração do DBAL, consulte reference-dbal-configuration.

Você pode acessar a conexão DBAL do Doctrine acessando o serviço `database_connection`:

```
class UserController extends Controller
{
    public function indexAction()
    {
        $conn = $this->get('database_connection');
        $users = $conn->fetchAll('SELECT * FROM users');

        // ...
    }
}
```

Registrando Tipos de Mapeamento Personalizados

Você pode registrar tipos de mapeamento personalizados através de configuração do symfony. Eles serão adicionados à todas as conexões configuradas. Para mais informações sobre tipos de mapeamento personalizados, leia a seção Tipos de Mapeamento Personalizados na documentação do Doctrine.

- *YAML*

```
    # app/config/config.yml
    doctrine:
        dbal:
            types:
```

```
                custom_first: Acme\HelloBundle\Type\CustomFirst
                custom_second: Acme\HelloBundle\Type\CustomSecond
```

- *XML*

```
<!-- app/config/config.xml -->
<container xmlns="http://symfony.com/schema/dic/services"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:doctrine="http://symfony.com/schema/dic/doctrine"
    xsi:schemaLocation="http://symfony.com/schema/dic/services http://symfony.com/schema/dic/ser
                        http://symfony.com/schema/dic/doctrine http://symfony.com/schema/dic/doc

    <doctrine:config>
        <doctrine:dbal>
            <doctrine:type name="custom_first" class="Acme\HelloBundle\Type\CustomFirst" />
            <doctrine:type name="custom_second" class="Acme\HelloBundle\Type\CustomSecond" />
        </doctrine:dbal>
    </doctrine:config>
</container>
```

- *PHP*

```
// app/config/config.php
$container->loadFromExtension('doctrine', array(
    'dbal' => array(
        'types' => array(
            'custom_first'  => 'Acme\HelloBundle\Type\CustomFirst',
            'custom_second' => 'Acme\HelloBundle\Type\CustomSecond',
        ),
    ),
));
```

Registrando Tipos de Mapeamento Personalizados no SchemaTool

O SchemaTool é usado para inspecionar o banco de dados para comparar o esquema. Para realizar esta tarefa, ele precisa saber que tipo de mapeamento precisa ser usado para cada um dos tipos do banco de dados. O registro de novos pode ser feito através de configuração.

Vamos mapear o tipo ENUM (por padrão não suportado pelo DBAL) para um tipo de mapeamento `string`:

- *YAML*

```
# app/config/config.yml
doctrine:
    dbal:
        connections:
            default:
                // Other connections parameters
                mapping_types:
                    enum: string
```

- *XML*

```
<!-- app/config/config.xml -->
<container xmlns="http://symfony.com/schema/dic/services"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:doctrine="http://symfony.com/schema/dic/doctrine"
    xsi:schemaLocation="http://symfony.com/schema/dic/services http://symfony.com/schema/dic/ser
```

```
                        http://symfony.com/schema/dic/doctrine http://symfony.com/schema/dic/doc

        <doctrine:config>
            <doctrine:dbal>
            <doctrine:dbal default-connection="default">
                <doctrine:connection>
                    <doctrine:mapping-type name="enum">string</doctrine:mapping-type>
                </doctrine:connection>
            </doctrine:dbal>
        </doctrine:config>
    </container>
```

- *PHP*

```
    // app/config/config.php
    $container->loadFromExtension('doctrine', array(
        'dbal' => array(
            'connections' => array(
                'default' => array(
                    'mapping_types' => array(
                        'enum'  => 'string',
                    ),
                ),
            ),
        ),
    ));
```

Como gerar Entidades de uma base de dados existente

Quando se começa a trabalhar em um novo projeto que usa banco de dados, duas diferentes situações são comuns. Na maioria dos casos, o modelo de banco de dados é projetado e construído do zero. Mas algumas vezes, você começará com um modelo de banco de dados existente e provavelmente não poderá alterá-lo. Felizmente, o Doctrine possui muitas ferramentas que podem te ajudar a gerar as classes model do seu de banco de dados já existente.

Nota: Como a Doctrine tools documentation diz, engenharia reversa é um processo único ao iniciar um projeto. Doctrine é capaz de converter aproximadamente 70-80% das instruções de mapeamento necessárias baseadas em campos, índices e chaves estrangeiras. Doctrine não pode descobrir associações inversas, tipos de herança, entidades com chaves estrangeiras como chaves primárias ou operações semânticas em associações como cascata ou eventos de ciclo de vida. Alguns ajustes adicionais nas entidades geradas são necessários posteriormente para adequar as especificidades de cada modelo de domínio.

Este tutorial asssume que você está utilizando uma simples aplicação de blog com as seguintes duas tabelas: `blog_post` e `blog_comment`. Um comentário gravado é ligado a um post gravado graças a uma chave estrangeira.

```
CREATE TABLE `blog_post` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT,
  `title` varchar(100) COLLATE utf8_unicode_ci NOT NULL,
  `content` longtext COLLATE utf8_unicode_ci NOT NULL,
  `created_at` datetime NOT NULL,
  PRIMARY KEY (`id`),
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8 COLLATE=utf8_unicode_ci;

CREATE TABLE `blog_comment` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT,
  `post_id` bigint(20) NOT NULL,
```

```
  `author` varchar(20) COLLATE utf8_unicode_ci NOT NULL,
  `content` longtext COLLATE utf8_unicode_ci NOT NULL,
  `created_at` datetime NOT NULL,
  PRIMARY KEY (`id`),
  KEY `blog_comment_post_id_idx` (`post_id`),
  CONSTRAINT `blog_post_id` FOREIGN KEY (`post_id`) REFERENCES `blog_post` (`id`) ON DELETE CASCADE
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8 COLLATE=utf8_unicode_ci;
```

Antes de começar a receita, verifique se seus parâmetros de conexão com banco de dados estão configurados corretamente no arquivo `app/config/parameters.yml` (ou onde quer que sua configuração de banco de dados é mantida) e que você tenha inicializado um bundle que irá receber sua futura classe de entidade. NEste tutorial, vamos supor que um `AcmeBlogBundle` existe e está localizado na pasta `src/Acme/BlogBundle`.

O primeiro passo para construir as classes de entidade de uma base da dados é solicitar que o Doctrine faça a introspecção do banco de dados e gere os arquivos de metadados. Os arquivos de metadados descrevem a classe de entidade baseados nos campos da tabela.

```
php app/console doctrine:mapping:convert xml ./src/Acme/BlogBundle/Resources/config/doctrine/metadata
```

Esta ferramenta de linha de comando pede para o Doctrine fazer a introspecção do banco de dados e gerar os arquivos de metadados na pasta `src/Acme/BlogBundle/Resources/config/doctrine/metadata/orm` do seu bundle.

Dica: Também é possivel gerar a classe de metadados no formato YAML alterando o primeiro argumento para *yml*.

O arquivo de metadados gerado `BlogPost.dcm.xml` é semelhante a isto:

```
<?xml version="1.0" encoding="utf-8"?>
<doctrine-mapping>
  <entity name="BlogPost" table="blog_post">
    <change-tracking-policy>DEFERRED_IMPLICIT</change-tracking-policy>
    <id name="id" type="bigint" column="id">
      <generator strategy="IDENTITY"/>
    </id>
    <field name="title" type="string" column="title" length="100"/>
    <field name="content" type="text" column="content"/>
    <field name="isPublished" type="boolean" column="is_published"/>
    <field name="createdAt" type="datetime" column="created_at"/>
    <field name="updatedAt" type="datetime" column="updated_at"/>
    <field name="slug" type="string" column="slug" length="255"/>
    <lifecycle-callbacks/>
  </entity>
</doctrine-mapping>
```

Uma vez que os arquivos de metados foram gerados, você pode pedir para Doctrine importar o esquema e construir as classes de entidade relacionadas com a execução dos dois comandos a seguir.

```
php app/console doctrine:mapping:import AcmeBlogBundle annotation
php app/console doctrine:generate:entities AcmeBlogBundle
```

O primeiro comando gera as classes de entidade com um mapeamento de anotações, msa você poderá alterar o argumento `annotation``para ``xml` ou `yml`. A classe de entidade `BlogComment` recém-criada é semelhante a isto:

```
<?php

// src/Acme/BlogBundle/Entity/BlogComment.php
```

```php
namespace Acme\BlogBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * Acme\BlogBundle\Entity\BlogComment
 *
 * @ORM\Table(name="blog_comment")
 * @ORM\Entity
 */
class BlogComment
{
    /**
     * @var bigint $id
     *
     * @ORM\Column(name="id", type="bigint", nullable=false)
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="IDENTITY")
     */
    private $id;

    /**
     * @var string $author
     *
     * @ORM\Column(name="author", type="string", length=100, nullable=false)
     */
    private $author;

    /**
     * @var text $content
     *
     * @ORM\Column(name="content", type="text", nullable=false)
     */
    private $content;

    /**
     * @var datetime $createdAt
     *
     * @ORM\Column(name="created_at", type="datetime", nullable=false)
     */
    private $createdAt;

    /**
     * @var BlogPost
     *
     * @ORM\ManyToOne(targetEntity="BlogPost")
     * @ORM\JoinColumn(name="post_id", referencedColumnName="id")
     */
    private $post;
}
```

Como você pode ver, Doctrine converte todos os campos da tabela para propriedades privadas e anotadas da classe. O mais impressionante é que ele descobre o relacionamento com a classe de entidade `BlogPost` baseada na restrição de chave estrangeira. Conseqüentemente, você pode encontrar uma propriedade privada `$post` mapeada com por uma entidade `BlogPost` na classe de entidade `BlogComment`.

O último comando gerou todos os getters e setters para todas as propriedades das duas classes de entidade `BlogPost` e `BlogComment`. As entidades geradas agora estão prontas para serem usadas. Divirta-se!

Como trabalhar com Múltiplos Gerenciadores de Entidade

Você pode usar múltiplos gerenciadores de entidades em uma aplicação Symfony2. Isto é necessário se você está usando bancos de dados diferentes ou algum vendor com conjuntos de entidades completamente diferentes. Em outras palavras, um gerenciador de entidades que conecta em um banco de dados manipulará algumas entidades enquanto um outro gerenciador de entidades que conecta a um outro banco de dados irá manupular as entidades restantes.

Nota: Usar múltiplos gerenciadores de entidade é muito fácil, mas mais avançado e geralmente não necessário. Certifique se você realmente precisa de múltiplos gerenciadores de entidades antes de adicionar esta camada de complexibilidade.

O código de configuração seguinte mostra como configurar dois gerenciadores de entidade:

- *YAML*

```
    doctrine:
        orm:
            default_entity_manager:   default
            entity_managers:
                default:
                    connection:       default
                    mappings:
                        AcmeDemoBundle: ~
                        AcmeStoreBundle: ~
                customer:
                    connection:       customer
                    mappings:
                        AcmeCustomerBundle: ~
```

Neste caso, você deve definir dois gerenciadores de entidade e chamá-los de `default` e `customer`. O gerenciador de entidade `default` manipula as entidades em `AcmeDemoBundle` e `AcmeStoreBundle`, enquanto o gerenciador de entidades `customer` manipula as entidades `AcmeCustomerBundle`.

Quando estiver trabalhando com múltiplos gerenciadores de entidade, você deve ser explícito sobre qual gerenciador de entidade você quer. Se você *omitir* o nome do gerenciador de entidade quando você atualizar o seu schema, será usado o padrão (ou seja, `default`):

```
# Play only with "default" mappings
php app/console doctrine:schema:update --force

# Play only with "customer" mappings
php app/console doctrine:schema:update --force --em=customer
```

Se você *omitir* o nome do gerenciador de entidade ao solicitar ele, o gerenciador de entidade padrão (ou seja, `default`) é retornado:

```
class UserController extends Controller
{
    public function indexAction()
    {
        // both return the "default" em
        $em = $this->get('doctrine')->getManager();
        $em = $this->get('doctrine')->getManager('default');

        $customerEm =  $this->get('doctrine')->getManager('customer');
    }
}
```

Agora você pode usar Doctrine exatamente da mesma forma que você fez antes - usando o gerenciador de entidade `default` para persistir e buscar as entidades que ele gerencia, e o gerenciador de entidade `customer` para persistir e buscar suas entidades.

O mesmo se aplica para chamadas de repositório:

```
class UserController extends Controller
{
    public function indexAction()
    {
        // Retrieves a repository managed by the "default" em
        $products = $this->get('doctrine')
                         ->getRepository('AcmeStoreBundle:Product')
                         ->findAll();

        // Explicit way to deal with the "default" em
        $products = $this->get('doctrine')
                         ->getRepository('AcmeStoreBundle:Product', 'default')
                         ->findAll();

        // Retrieves a repository managed by the "customer" em
        $customers = $this->get('doctrine')
                          ->getRepository('AcmeCustomerBundle:Customer', 'customer')
                          ->findAll();
    }
}
```

Como Registrar Funções DQL Personalizadas

O Doctrine permite especificar funções DQL personalizadas. Para mais informações sobre este assunto, leia o artigo "Funções DQL Definidas pelo Usuário" no cookbook do Doctrine.

No Symfony, você pode registrar suas funções DQL personalizadas da seguinte forma:

- *YAML*

```
# app/config/config.yml
doctrine:
    orm:
        # ...
        entity_managers:
            default:
                # ...
                dql:
                    string_functions:
                        test_string: Acme\HelloBundle\DQL\StringFunction
                        second_string: Acme\HelloBundle\DQL\SecondStringFunction
                    numeric_functions:
                        test_numeric: Acme\HelloBundle\DQL\NumericFunction
                    datetime_functions:
                        test_datetime: Acme\HelloBundle\DQL\DatetimeFunction
```

- *XML*

```
<!-- app/config/config.xml -->
<container xmlns="http://symfony.com/schema/dic/services"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:doctrine="http://symfony.com/schema/dic/doctrine"
    xsi:schemaLocation="http://symfony.com/schema/dic/services http://symfony.com/schema/dic/ser
```

```
                            http://symfony.com/schema/dic/doctrine http://symfony.com/schema/dic/doc

        <doctrine:config>
            <doctrine:orm>
                <!-- ... -->
                <doctrine:entity-manager name="default">
                    <!-- ... -->
                    <doctrine:dql>
                        <doctrine:string-function name="test_string>Acme\HelloBundle\DQL\StringFunct
                        <doctrine:string-function name="second_string>Acme\HelloBundle\DQL\SecondStr
                        <doctrine:numeric-function name="test_numeric>Acme\HelloBundle\DQL\NumericFu
                        <doctrine:datetime-function name="test_datetime>Acme\HelloBundle\DQL\Datetim
                    </doctrine:dql>
                </doctrine:entity-manager>
            </doctrine:orm>
        </doctrine:config>
    </container>
```

- *PHP*

```
    // app/config/config.php
    $container->loadFromExtension('doctrine', array(
        'orm' => array(
            ...,
            'entity_managers' => array(
                'default' => array(
                    ...,
                    'dql' => array(
                        'string_functions' => array(
                            'test_string'   => 'Acme\HelloBundle\DQL\StringFunction',
                            'second_string' => 'Acme\HelloBundle\DQL\SecondStringFunction',
                        ),
                        'numeric_functions' => array(
                            'test_numeric' => 'Acme\HelloBundle\DQL\NumericFunction',
                        ),
                        'datetime_functions' => array(
                            'test_datetime' => 'Acme\HelloBundle\DQL\DatetimeFunction',
                        ),
                    ),
                ),
            ),
        ),
    ));
```

Como Implementar um Formulário Simples de Registro

Alguns formulários possuem campos extras cujos valores não precisam ser armazenados no banco de dados. Por exemplo, você pode criar um formulário de registo com alguns campos extras (como um campo checkbox "termos de aceite") e incorporar o formulário que realmente armazena as informações da conta.

O modelo `User`

Você tem uma entidade simples `User` mapeada para o banco de dados:

```
// src/Acme/AccountBundle/Entity/User.php
namespace Acme\AccountBundle\Entity;
```

```
use Doctrine\ORM\Mapping as ORM;
use Symfony\Component\Validator\Constraints as Assert;
use Symfony\Bridge\Doctrine\Validator\Constraints\UniqueEntity;

/**
 * @ORM\Entity
 * @UniqueEntity(fields="email", message="Email already taken")
 */
class User
{
    /**
     * @ORM\Id
     * @ORM\Column(type="integer")
     * @ORM\GeneratedValue(strategy="AUTO")
     */
    protected $id;

    /**
     * @ORM\Column(type="string", length=255)
     * @Assert\NotBlank()
     * @Assert\Email()
     */
    protected $email;

    /**
     * @ORM\Column(type="string", length=255)
     * @Assert\NotBlank()
     */
    protected $plainPassword;

    public function getId()
    {
        return $this->id;
    }

    public function getEmail()
    {
        return $this->email;
    }

    public function setEmail($email)
    {
        $this->email = $email;
    }

    public function getPlainPassword()
    {
        return $this->plainPassword;
    }

    public function setPlainPassword($password)
    {
        $this->plainPassword = $password;
    }
}
```

Esta entidade `User` contém três campos, e dois deles (`email` e `plainPassword`) devem ser exibos no formulário. A propriedade e-mail deve ser única no banco de dados, isto é aplicado através da adição da validação no topo da

classe.

Nota: Se você quiser integrar este `User` com o sistema de segurança, você precisa implementar a *UserInterface* do componente de segurança.

Criando um Formulário para o Modelo

Em seguida, crie o formulário para o modelo `User`:

```
// src/Acme/AccountBundle/Form/Type/UserType.php
namespace Acme\AccountBundle\Form\Type;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolverInterface;

class UserType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder->add('email', 'email');
        $builder->add('plainPassword', 'repeated', array(
           'first_name' => 'password',
           'second_name' => 'confirm',
           'type' => 'password',
        ));
    }

    public function setDefaultOptions(OptionsResolverInterface $resolver)
    {
        $resolver->setDefaults(array(
            'data_class' => 'Acme\AccountBundle\Entity\User'
        ));
    }

    public function getName()
    {
        return 'user';
    }
}
```

Há apenas dois campos:`email` e `plainPassword` (repetido para confirmar a senha digitada). A opção `data_class` diz ao formulário o nome da classe de dados (ou seja, a sua entidade `User`).

Dica: Para explorar mais sobre o componente de formulário, leia Formulários.

Incorporando o Formulário do User no Formulário de Registro

O formulário que você vai usar para a página de registo não será o mesmo usado para modificar o `User` (ou seja, `UserType`). O formulário de registro conterá novos campos como o "aceitar os termos", cujo valor não será armazenado no banco de dados.

Comece criando uma classe simples que representa o "registro":

```
// src/Acme/AccountBundle/Form/Model/Registration.php
namespace Acme\AccountBundle\Form\Model;

use Symfony\Component\Validator\Constraints as Assert;

use Acme\AccountBundle\Entity\User;

class Registration
{
    /**
     * @Assert\Type(type="Acme\AccountBundle\Entity\User")
     */
    protected $user;

    /**
     * @Assert\NotBlank()
     * @Assert\True()
     */
    protected $termsAccepted;

    public function setUser(User $user)
    {
        $this->user = $user;
    }

    public function getUser()
    {
        return $this->user;
    }

    public function getTermsAccepted()
    {
        return $this->termsAccepted;
    }

    public function setTermsAccepted($termsAccepted)
    {
        $this->termsAccepted = (Boolean) $termsAccepted;
    }
}
```

Em seguida, crie o formulário para este modelo `Registration`:

```
// src/Acme/AccountBundle/Form/Type/RegistrationType.php
namespace Acme\AccountBundle\Form\Type;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;

class RegistrationType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder->add('user', new UserType());
        $builder->add(
            'terms',
            'checkbox',
            array('property_path' => 'termsAccepted')
        );
```

```
    }

    public function getName()
    {
        return 'registration';
    }
}
```

Você não precisa usar um método especial para incorporar o formulário `UserType`. Um formulário também é um campo - logo, você pode adicionar ele como qualquer outro campo, com a certeza de que a propriedade `Registration.user` irá manter uma instância da classe `User`.

Manuseando a Submissão do Formulário

Em seguida, você precisa de um controlador para lidar com o formulário. Comece criando um controlador simples para exibir o formulário de registro:

```
// src/Acme/AccountBundle/Controller/AccountController.php
namespace Acme\AccountBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Response;

use Acme\AccountBundle\Form\Type\RegistrationType;
use Acme\AccountBundle\Form\Model\Registration;

class AccountController extends Controller
{
    public function registerAction()
    {
        $form = $this->createForm(
            new RegistrationType(),
            new Registration()
        );

        return $this->render(
            'AcmeAccountBundle:Account:register.html.twig',
            array('form' => $form->createView())
        );
    }
}
```

e o seu template:

```
{# src/Acme/AccountBundle/Resources/views/Account/register.html.twig #}
<form action="{{ path('create') }}" method="post" {{ form_enctype(form) }}>
    {{ form_widget(form) }}

    <input type="submit" />
</form>
```

Por fim, adicione o controlador que lida com a submissão do formulário. Ele realiza a validação e salva os dados no banco de dados:

```
public function createAction()
{
    $em = $this->getDoctrine()->getEntityManager();
```

```
    $form = $this->createForm(new RegistrationType(), new Registration());

    $form->bind($this->getRequest());

    if ($form->isValid()) {
        $registration = $form->getData();

        $em->persist($registration->getUser());
        $em->flush();

        return $this->redirect(...);
    }

    return $this->render(
        'AcmeAccountBundle:Account:register.html.twig',
        array('form' => $form->createView())
    );
}
```

Pronto! O seu formulário agora valida e permite que você salve o objeto `User` no banco de dados. O checkbox extra `terms` na classe de modelo `Registration` é utilizado durante a validação, mas não é utilizado posteriormente quando salvamos o usuário no banco de dados.

3.1.6 Formulário

Como personalizar a Renderização de Formulários

O Symfony dispõe de uma grande variedade de maneiras para personalizar como um formulário é renderizado. Neste guia, você vai aprender a personalizar cada parte possível do seu formulário com o menor esforço, se você usa o Twig ou PHP como sua templating engine.

Noções Básicas sobre a Renderização de Formulários

Lembre-se que o widget HTML, a label e o erro de um campo do formulário podem ser facilmente renderizados usando a função Twig `form_row` ou o método helper PHP `row`:

- *Twig*

```
{{ form_row(form.age) }}
```

- *PHP*

```
<?php echo $view['form']->row($form['age']) }} ?>
```

Você também pode renderizar cada uma das três partes do campo individualmente:

- *Twig*

```
<div>
    {{ form_label(form.age) }}
    {{ form_errors(form.age) }}
    {{ form_widget(form.age) }}
</div>
```

- *PHP*

```
    <div>
        <?php echo $view['form']->label($form['age']) }} ?>
        <?php echo $view['form']->errors($form['age']) }} ?>
        <?php echo $view['form']->widget($form['age']) }} ?>
    </div>
```

Em ambos os casos, a label do formulário, os erros e o widget HTML são renderizados usando um conjunto de marcação que vem por padrão com o symfony. Por exemplo, ambos os templates acima seriam renderizados da seguinte forma:

```
<div>
    <label for="form_age">Age</label>
    <ul>
        <li>This field is required</li>
    </ul>
    <input type="number" id="form_age" name="form[age]" />
</div>
```

Para protótipos rápidos e para testar um formulário, você pode renderizar todo o formulário com apenas uma linha:

- *Twig*

```
    {{ form_widget(form) }}
```

- *PHP*

```
    <?php echo $view['form']->widget($form) }} ?>
```

O restante desta receita irá explicar como cada parte de marcação do formulário pode ser modificada em vários níveis diferentes. Para mais informações sobre a renderização do formulário em geral, consulte *Renderizando um formulário em um Template*.

O que são Temas de Formulário?

O Symfony utiliza fragmentos de formulário - um pequeno pedaço de um template que renderiza apenas uma parte de um formulário - para renderizar cada parte de um formulário - labels de campo, os erros, campos de texto `input`, tags `select`, etc

Os fragmentos são definidos como blocos no Twig e como arquivos de template no PHP.

Um *tema* é nada mais do que um conjunto de fragmentos que você deseja usar quando está renderizando um formulário. Em outras palavras, se você quiser personalizar uma parte de como um formulário é processado, você vai importar um *tema* que contém uma personalização dos fragmentos apropriados do formulário.

O Symfony vem com um tema padrão (form_div_layout.html.twig no Twig e `FrameworkBundle:Form` no PHP) que define cada fragmento necessário para renderizar cada parte de um formulário.

Na seção seguinte, você vai aprender a personalizar um tema, sobrescrevendo alguns ou todos os seus fragmentos.

Por exemplo, quando o widget de um campo do tipo `integer` é renderizado, um campo `input number` é gerado

- *Twig*

```
    {{ form_widget(form.age) }}
```

- *PHP*

```
    <?php echo $view['form']->widget($form['age']) ?>
```

renderiza:

```
<input type="number" id="form_age" name="form[age]" required="required" value="33" />
```

Internamente, o symfony usa o fragmento `integer_widget` para renderizar o campo. Isso ocorre porque o tipo de campo é `integer` e você está renderizando seu `widget` (em oposição a sua `label` ou `errors`).

No Twig ele seria por padrão o bloco `integer_widget` do template form_div_layout.html.twig.

No PHP ele seria o arquivo `integer_widget.html.php` localizado no diretório `FrameworkBundle/Resources/views/Form`.

A implementação padrão do fragmento `integer_widget` seria parecida com esta:

- *Twig*

```
{# form_div_layout.html.twig #}
{% block integer_widget %}
    {% set type = type|default('number') %}
    {{ block('field_widget') }}
{% endblock integer_widget %}
```

- *PHP*

```
<!-- integer_widget.html.php -->
<?php echo $view['form']->renderBlock('field_widget', array('type' => isset($type) ? $type : "nu
```

Como você pode ver, este próprio fragmento renderiza outro fragmento - `field_widget`:

- *Twig*

```
{# form_div_layout.html.twig #}
{% block field_widget %}
    {% set type = type|default('text') %}
    <input type="{{ type }}" {{ block('widget_attributes') }} value="{{ value }}" />
{% endblock field_widget %}
```

- *PHP*

```
<!-- FrameworkBundle/Resources/views/Form/field_widget.html.php -->
<input
    type="<?php echo isset($type) ? $view->escape($type) : "text" ?>"
    value="<?php echo $view->escape($value) ?>"
    <?php echo $view['form']->renderBlock('attributes') ?>
/>
```

O ponto é, os fragmentos ditam a saída HTML de cada parte de um formulário. Para personalizar a saída do formulário, você só precisa identificar e substituir o fragmento apropriado. O conjunto dessas personalizações de fragmentos de formulário é conhecida como "tema" de formulário. Ao renderizar um formulário, você pode escolher qual(ais) tema(s) de formulário deseja aplicar.

No Twig, um tema é um único arquivo de template e os fragmentos são os blocos definidos neste arquivo.

No PHP um tema é um diretório e os fragmentos são os arquivos de template individuais neste diretório.

Sabendo qual bloco personalizar

Neste exemplo, o nome do fragmento personalizado é `integer_widget` porque você quis sobrescrever o `widget` HTML para todos os tipos de campo `integer`. Se você precisa personalizar campos textarea, você iria personalizar o `textarea_widget`.
Como você pode ver, o nome do fragmento é uma combinação do tipo do campo e de qual parte do campo está sendo renderizada (ex.: `widget`, `label`, `errors`, `row`). Como tal, para personalizar a forma como os erros são renderizados apenas para campos input `text`, você deve personalizar o fragmento `text_errors`.
Mais comumente, no entanto, você vai querer personalizar a forma como os erros são exibidos através de *todos* os campos. Você pode fazer isso personalizando o fragmento `field_errors`. Isso aproveita a herança do tipo de campo. Especificamente, uma vez que o tipo `text` estende o tipo `field`, o componente de formulário vai procurar primeiro pelo fragmento de tipo específico (ex., `text_errors`) antes de voltar ao seu nome de fragmento pai, no caso dele não existir (ex., `field_errors`).
Para mais informações sobre este tópico, consulte *Nomeando os fragmentos do formulário*.

Tematizando os Formulários

Para ver o poder da tematização de formulários, suponha que você queira envolver cada campo input `number` com uma tag `div`. A chave para fazer isso é personalizar o fragmento `integer_widget`.

Tematização de Formulários no Twig

Ao personalizar o bloco de campo de formulário no Twig, você tem duas opções de *onde* o bloco de formulário personalizado pode residir:

Método	Prós	Contras
Dentro do mesmo template que o formulário	Rápido e fácil	Não pode ser reutilizado em outros templates
Dentro de um template separado	Pode ser reutilizado por muitos templates	Requer que um template extra seja criado

Ambos os métodos têm o mesmo efeito, mas são melhores em situações diferentes.

Método 1: Dentro do mesmo Template que o Formulário A maneira mais fácil de personalizar o bloco `integer_widget` é personalizá-lo diretamente no mesmo template que está renderizando o formulário.

```
{% extends '::base.html.twig' %}

{% form_theme form _self %}

{% block integer_widget %}
    <div class="integer_widget">
        {% set type = type|default('number') %}
        {{ block('field_widget') }}
    </div>
{% endblock %}

{% block content %}
    {# ... render the form #}

    {{ form_row(form.age) }}
{% endblock %}
```

Ao usar a tag especial `{% form_theme form _self %}`, o Twig procura dentro do mesmo template por qualquer blocos de formulários sobrescritos. Assumindo que o campo `form.age` é um tipo de campo `integer`, quando o widget é renderizado, o bloco `integer_widget` personalizado irá ser utilizado.

A desvantagem deste método é que o bloco de formulário personalizado não pode ser reutilizado ao renderizar outros formulários em outros templates. Em outras palavras, este método é mais útil ao fazer personalizações de formulários que são específicas para um único formulário em sua aplicação. Se você quiser reutilizar uma personalização em vários (ou todos) os formulários de sua aplicação, leia a próxima seção.

Método 2: Dentro de um Template Separado Você também pode optar por colocar o bloco de formulário `integer_widget` personalizado em um template totalmente separado. O código e o resultado final é o mesmo, mas agora você pode reutilizar a personalização de formulário em muitos templates:

```
{# src/Acme/DemoBundle/Resources/views/Form/fields.html.twig #}
{% block integer_widget %}
    <div class="integer_widget">
        {% set type = type|default('number') %}
        {{ block('field_widget') }}
    </div>
{% endblock %}
```

Agora que você já criou o bloco de formulário personalizado, você precisa dizer ao Symfony para usá-lo. Dentro do template onde você está renderizando o seu formulário, diga ao Symfony para usar o template através da tag `form_theme`:

```
{% form_theme form 'AcmeDemoBundle:Form:fields.html.twig' %}

{{ form_widget(form.age) }}
```

Quando o widget `form.age` é renderizado, o Symfony usará o bloco `integer_widget` do novo template e a tag `input` vai ser envolvida no elemento `div` especificado no bloco personalizado.

Tematizando Formulários em PHP

Ao usar o PHP como templating engine, o único método de personalizar um fragmento é criar um novo arquivo template - é semelhante ao segundo método utilizado pelo Twig.

O arquivo de template deve ser nomeado após o fragmento. Você deve criar um arquivo `integer_widget.html.php` para personalizar o fragmento `integer_widget`.

```
<!-- src/Acme/DemoBundle/Resources/views/Form/integer_widget.html.php -->
<div class="integer_widget">
    <?php echo $view['form']->renderBlock('field_widget', array('type' => isset($type) ? $type : "nur
</div>
```

Agora que você criou o template de formulário personalizado, você precisa dizer ao Symfony para usá-lo. Dentro do template onde você está renderizando o seu formulário, diga ao Symfony para usar o tema através do método helper `setTheme`:

```
<?php $view['form']->setTheme($form, array('AcmeDemoBundle:Form')) ;?>

<?php $view['form']->widget($form['age']) ?>
```

Quando o widget `form.age` é renderizado, o Symfony vai usar o template `integer_widget.html.php` personalizado e a tag `input` será envolvida pelo elemento `div`.

Referenciando Blocos de Formulário Base (específico para o Twig)

Até agora, para sobrescrever um bloco de formulário em particular, o melhor método é copiar o bloco padrão de form_div_layout.html.twig, colá-lo em um template diferente, e, em seguida, personalizá-lo. Em muitos casos, você pode evitar ter que fazer isso através da referência ao bloco base quando for personalizá-lo.

Isso é fácil de fazer, mas varia um pouco dependendo se as personalizações de seu bloco de formulário estão no mesmo template que o formulário ou em um template separado.

Referenciando Blocos no interior do mesmo Template que o Formulário Importe os blocos adicionando uma tag `use` no template onde você está renderizando o formulário:

```
{% use 'form_div_layout.html.twig' with integer_widget as base_integer_widget %}
```

Agora, quando os blocos do form_div_layout.html.twig são importados, o bloco `integer_widget` é chamado `base_integer_widget`. Isto significa que quando você redefinir o bloco `integer_widget`, você pode fazer referência a marcação padrão através do `base_integer_widget`:

```
{% block integer_widget %}
    <div class="integer_widget">
        {{ block('base_integer_widget') }}
    </div>
{% endblock %}
```

Referenciando Blocos Base a partir de um Template Externo Se as personalizações do formulário residirem dentro de um template externo, você pode fazer referência ao bloco base usando a função `parent()` do Twig:

```
{# src/Acme/DemoBundle/Resources/views/Form/fields.html.twig #}
{% extends 'form_div_layout.html.twig' %}

{% block integer_widget %}
    <div class="integer_widget">
        {{ parent() }}
    </div>
{% endblock %}
```

Nota: Não é possível fazer referência ao bloco base quando se utiliza o PHP como template engine. Você tem que copiar manualmente o conteúdo do bloco base para o seu novo arquivo de template.

Fazendo Personalizações para toda a Aplicação

Se você deseja que uma certa personalização de formulário seja global para a sua aplicação, você pode conseguir isso fazendo as personalizações de formulário em um template externo e depois importá-lo dentro da sua configuração da aplicação:

Twig Usando a seguinte configuração, quaisquer blocos de formulários personalizados dentro do template `AcmeDemoBundle:Form:fields.html.twig` serão usados globalmente quando um formulário é renderizado.

- *YAML*

```yaml
# app/config/config.yml
twig:
    form:
        resources:
            - 'AcmeDemoBundle:Form:fields.html.twig'
    # ...
```

- *XML*

```xml
<!-- app/config/config.xml -->
<twig:config ...>
        <twig:form>
            <resource>AcmeDemoBundle:Form:fields.html.twig</resource>
        </twig:form>
        <!-- ... -->
</twig:config>
```

- *PHP*

```php
// app/config/config.php
$container->loadFromExtension('twig', array(
    'form' => array(
        'resources' => array(
            'AcmeDemoBundle:Form:fields.html.twig',
        ),
    ),

    // ...
));
```

Por padrão, o Twig usa um layout *div* ao renderizar os formulários. Algumas pessoas, no entanto, podem preferir renderizar formulários em um layout de *tabela*. Para isso use o recurso `form_table_layout.html.twig` como layout:

- *YAML*

```yaml
# app/config/config.yml
twig:
    form:
        resources: ['form_table_layout.html.twig']
    # ...
```

- *XML*

```xml
<!-- app/config/config.xml -->
<twig:config ...>
        <twig:form>
            <resource>form_table_layout.html.twig</resource>
        </twig:form>
        <!-- ... -->
</twig:config>
```

- *PHP*

```php
// app/config/config.php
$container->loadFromExtension('twig', array(
    'form' => array(
        'resources' => array(
            'form_table_layout.html.twig',
        ),
```

```
        ),

        // ...
    ));
```

Se você quer fazer a alteração somente em um template, adicione a seguinte linha em seu arquivo de template em vez de adicionar o template como um recurso:

```
{% form_theme form 'form_table_layout.html.twig' %}
```

Note que a variável `form` no código acima é a variável de visão do formulário que você passou para o seu template.

PHP Usando a seguinte configuração, quaisquer fragmentos de formulários personalizados no interior do diretório `src/Acme/DemoBundle/Resources/views/Form` serão usados globalmente quando um formulário é renderizado.

- *YAML*

```
# app/config/config.yml
framework:
    templating:
        form:
            resources:
                - 'AcmeDemoBundle:Form'
    # ...
```

- *XML*

```
<!-- app/config/config.xml -->
<framework:config ...>
    <framework:templating>
        <framework:form>
            <resource>AcmeDemoBundle:Form</resource>
        </framework:form>
    </framework:templating>
    <!-- ... -->
</framework:config>
```

- *PHP*

```
// app/config/config.php
// PHP
$container->loadFromExtension('framework', array(
    'templating' => array(
        'form' => array(
            'resources' => array(
                'AcmeDemoBundle:Form',
            ),
        ),
     ),

     // ...
));
```

Por padrão, a engine PHP usa um layout *div* ao renderizar formulários. Algumas pessoas, no entanto, podem preferir renderizar formulários em um layout de *tabela*. Para isso use o recurso `FrameworkBundle:FormTable` como layout:

- *YAML*

```yaml
# app/config/config.yml
framework:
    templating:
        form:
            resources:
                - 'FrameworkBundle:FormTable'
```

- *XML*

```xml
<!-- app/config/config.xml -->
<framework:config ...>
    <framework:templating>
        <framework:form>
            <resource>FrameworkBundle:FormTable</resource>
        </framework:form>
    </framework:templating>
    <!-- ... -->
</framework:config>
```

- *PHP*

```php
// app/config/config.php
$container->loadFromExtension('framework', array(
    'templating' => array(
        'form' => array(
            'resources' => array(
                'FrameworkBundle:FormTable',
            ),
        ),
    ),

     // ...
));
```

Se você quer fazer a alteração somente em um template, adicione a seguinte linha em seu arquivo de template em vez de adicionar o template como um recurso:

```php
<?php $view['form']->setTheme($form, array('FrameworkBundle:FormTable')); ?>
```

Note que a variável `$form` no código acima é a variável de visão do formulário que você passou para o seu template.

Como personalizar um campo individual

Até agora, você viu as diferentes maneiras que pode personalizar a saída dos widgets de todos os tipos de campo texto. Você também pode personalizar campos individuais. Por exemplo, supondo que você tenha dois campos `text` - `first_name` e `last_name` - mas você só quer personalizar um dos campos. Isto pode ser feito pela personalização de um fragmento cujo nome é uma combinação do atributo id do campo e qual parte do campo está sendo personalizada. Por exemplo:

- *Twig*

```twig
{% form_theme form _self %}

{% block _product_name_widget %}
    <div class="text_widget">
        {{ block('field_widget') }}
    </div>
{% endblock %}
```

```
{{ form_widget(form.name) }}
```

- *PHP*

```
<!-- Main template -->
<?php echo $view['form']->setTheme($form, array('AcmeDemoBundle:Form')); ?>

<?php echo $view['form']->widget($form['name']); ?>

<!-- src/Acme/DemoBundle/Resources/views/Form/_product_name_widget.html.php -->

<div class="text_widget">
      echo $view['form']->renderBlock('field_widget') ?>
</div>
```

Aqui, o fragmento `_product_name_widget` define o template a ser usado para o campo cujo *id* é `product_name` (e o nome é `product[name]`).

Dica: A parte `product` do campo é o nome do formulário, que pode ser definido manualmente ou gerado automaticamente com base no nome do tipo de formulário (por exemplo, `ProductType` equivale ao `product`). Se você não tem certeza de qual é o nome de seu formulário, apenas visualize o código fonte do formulário gerado.

Você também pode sobrescrever a marcação de uma linha inteira de campo utilizando o mesmo método:

- *Twig*

```
{# _product_name_row.html.twig #}
{% form_theme form _self %}

{% block _product_name_row %}
    <div class="name_row">
        {{ form_label(form) }}
        {{ form_errors(form) }}
        {{ form_widget(form) }}
    </div>
{% endblock %}
```

- *PHP*

```
<!-- _product_name_row.html.php -->

<div class="name_row">
    <?php echo $view['form']->label($form) ?>
    <?php echo $view['form']->errors($form) ?>
    <?php echo $view['form']->widget($form) ?>
</div>
```

Outras Personalizações Comuns

Até agora, esta receita tem demonstrado diversas maneiras de personalizar um único pedaço de como um formulário é renderizado. A chave é personalizar um fragmento específico que corresponde à parte do formulário que você deseja controlar (veja *naming form blocks*).

Nas próximas seções, você vai ver como é possível fazer várias personalizações comuns de formulário. Para aplicar essas personalizações, utilize um dos métodos descritos na seção *Tematizando os Formulários*.

Personalizando Saída de Erro

Nota: O componente de formulário só lida com *como* os erros de validação são renderizados, e não com as mensagens de erro de validação. As mensagens de erro em si são determinadas pelas restrições de validação que você aplicou aos seus objetos. Para mais informações, consulte o capítulo sobre validation.

Há muitas formas diferentes para personalizar como os erros são renderizados quando um formulário é enviado com erros. As mensagens de erro para um campo são renderizadas quando você usa o helper `form_errors`:

- *Twig*

```
{{ form_errors(form.age) }}
```

- *PHP*

```
<?php echo $view['form']->errors($form['age']); ?>
```

Por padrão, os erros são renderizados dentro de uma lista não ordenada:

```
<ul>
    <li>This field is required</li>
</ul>
```

Para sobrecrever como os erros são renderizados para *todos* os campos, basta copiar, colar e personalizar o fragmento `field_errors`.

- *Twig*

```
{# fields_errors.html.twig #}
{% block field_errors %}
    {% spaceless %}
        {% if errors|length > 0 %}
        <ul class="error_list">
            {% for error in errors %}
                <li>{{ error.messageTemplate|trans(error.messageParameters, 'validators') }}</li
            {% endfor %}
        </ul>
        {% endif %}
    {% endspaceless %}
{% endblock field_errors %}
```

- *PHP*

```
<!-- fields_errors.html.php -->
<?php if ($errors): ?>
    <ul class="error_list">
        <?php foreach ($errors as $error): ?>
            <li><?php echo $view['translator']->trans(
                $error->getMessageTemplate(),
                $error->getMessageParameters(),
                'validators'
            ) ?></li>
        <?php endforeach; ?>
    </ul>
<?php endif ?>
```

Dica: Veja *Tematizando os Formulários* para saber como aplicar essa personalização.

Você também pode personalizar a saída de erro de apenas um tipo de campo específico. Por exemplo, certos erros que são mais globais para o seu formulário (ou seja, não específico para apenas um campo) são renderizados separadamente, geralmente no topo do seu formulário:

- *Twig*

```
{{ form_errors(form) }}
```

- *PHP*

```
<?php echo $view['form']->render($form); ?>
```

Para personalizar *apenas* a marcação usada para esses erros, siga as mesmas instruções acima, mas agora chame o bloco `form_errors` (Twig) / o arquivo `form_errors.html.php` (PHP). Agora, quando os erros para o tipo `form` são renderizados, o seu fragmento personalizado será usado em vez do padrão `field_errors`.

Personalizando a "Linha de Formulário" Quando você pode gerenciá-lo, a maneira mais fácil de renderizar um campo de formulário é através da função `form_row`, que renderiza a label, os erros e o widget HTML de um campo. Para personalizar a marcação usada para renderizar *todas* as linhas de campo do formulário, sobrescreva o fragmento `field_row`. Por exemplo, suponha que você deseja adicionar uma classe para o elemento `div` que envolve cada linha:

- *Twig*

```
{# field_row.html.twig #}
{% block field_row %}
    <div class="form_row">
        {{ form_label(form) }}
        {{ form_errors(form) }}
        {{ form_widget(form) }}
    </div>
{% endblock field_row %}
```

- *PHP*

```
<!-- field_row.html.php -->
<div class="form_row">
    <?php echo $view['form']->label($form) ?>
    <?php echo $view['form']->errors($form) ?>
    <?php echo $view['form']->widget($form) ?>
</div>
```

Dica: Veja *Tematizando os Formulários* para saber como aplicar essa personalização.

Adicionando um Asterisco para as Labels de Campo "Obrigatórias" Se você quiser indicar todos os seus campos obrigatórios com um asterisco (`*`), você pode fazer isso personalizando o fragmento `field_label`.

No Twig, se você estiver fazendo a personalização de formulário dentro do mesmo template que o seu formulário, modifique a tag `use` e adicione o seguinte:

```
{% use 'form_div_layout.html.twig' with field_label as base_field_label %}

{% block field_label %}
    {{ block('base_field_label') }}

    {% if required %}
```

```
        <span class="required" title="This field is required">*</span>
    {% endif %}
{% endblock %}
```

No Twig, se você estiver fazendo a personalização do formulário dentro de um template separado, use o seguinte:

```
{% extends 'form_div_layout.html.twig' %}

{% block field_label %}
    {{ parent() }}

    {% if required %}
        <span class="required" title="This field is required">*</span>
    {% endif %}
{% endblock %}
```

Ao usar o PHP como template engine você tem que copiar o conteúdo do template original:

```
<!-- field_label.html.php -->

<!-- original content -->
<label for="<?php echo $view->escape($id) ?>" <?php foreach($attr as $k => $v) { printf('%s="%s" ', $

<!-- customization -->
<?php if ($required) : ?>
    <span class="required" title="This field is required">*</span>
<?php endif ?>
```

Dica: Veja *Tematizando os Formulários* para saber como aplicar essa personalização.

Adicionando mensagens de "ajuda" Você também pode personalizar os widgets do formulário para ter uma mensagem opcional de "ajuda".

No Twig, se você está fazendo a personalização do formulário dentro do mesmo template que o seu formulário, modifique a tag `use` e adicione o seguinte:

```
{% use 'form_div_layout.html.twig' with field_widget as base_field_widget %}

{% block field_widget %}
    {{ block('base_field_widget') }}

    {% if help is defined %}
        <span class="help">{{ help }}</span>
    {% endif %}
{% endblock %}
```

No Twig, se você está fazendo a personalização do formulário dentro de um template separado, use o seguinte:

```
{% extends 'form_div_layout.html.twig' %}

{% block field_widget %}
    {{ parent() }}

    {% if help is defined %}
        <span class="help">{{ help }}</span>
    {% endif %}
{% endblock %}
```

Ao usar o PHP como template engine você tem que copiar o conteúdo do template original:

```
<!-- field_widget.html.php -->

<!-- Original content -->
<input
    type="<?php echo isset($type) ? $view->escape($type) : "text" ?>"
    value="<?php echo $view->escape($value) ?>"
    <?php echo $view['form']->renderBlock('attributes') ?>
/>

<!-- Customization -->
<?php if (isset($help)) : ?>
    <span class="help"><?php echo $view->escape($help) ?></span>
<?php endif ?>
```

Para renderizar uma mensagem de ajuda abaixo de um campo, passe em uma variável `help`:

- *Twig*

```
{{ form_widget(form.title, {'help': 'foobar'}) }}
```

- *PHP*

```
<?php echo $view['form']->widget($form['title'], array('help' => 'foobar')) ?>
```

Dica: Veja:ref:*cookbook-form-theming-methods* para saber como aplicar essa personalização.

Usando Variáveis de Formulário

A maioria das funções disponíveis para renderizar diferentes partes de um formulário (por exemplo, o widget de formulário, a label do formulário, os erros de formulário, etc) também permitem que você faça certas personalizações diretamente. Veja o exemplo a seguir:

- *Twig*

```
{# render a widget, but add a "foo" class to it #}
{{ form_widget(form.name, { 'attr': {'class': 'foo'} }) }}
```

- *PHP*

```
<!-- render a widget, but add a "foo" class to it -->
<?php echo $view['form']->widget($form['name'], array(
    'attr' => array(
        'class' => 'foo',
    ),
)) ?>
```

O array passado como segundo argumento contém "variáveis" de formulário. Para mais detalhes sobre este conceito no Twig, veja twig-reference-form-variables.

Como usar os Transformadores de Dados

Você frenquentemente encontrará a necessidade de transformar os dados inseridos pelo usuário em um formulário para um outro formato para uso em seu programa. Isto pode ser feito manualmente no seu controlador facilmente, mas, e se você pretende utilizar este formulário específico em locais diferentes?

Digamos que você tenha uma relação um-para-um da *Task* para o *Issue*, por exemplo, uma *Task* opcionalmente tem um *issue* ligado à ela. Adicionando uma caixa de listagem com todos os possíveis *issues*, eventualmente, pode levar a uma caixa de listagem muito longa onde é impossível encontrar algo. Em vez disso, você poderia adicionar uma caixa de texto, onde o usuário pode simplesmente informar o número do *issue*.

Você poderia tentar fazer isso no seu controlador, mas não é a melhor solução. Seria melhor se esse *issue* fosse automaticamente convertido em um objeto *Issue*. É onde os transformadores de dados entram em jogo.

Criando o Transformador

Primeiro, crie uma classe *IssueToNumberTransformer* - esta classe será responsável por converter de e para o número do *issue* e o objeto *Issue*:

```
// src/Acme/TaskBundle/Form/DataTransformer/IssueToNumberTransformer.php
namespace Acme\TaskBundle\Form\DataTransformer;

use Symfony\Component\Form\DataTransformerInterface;
use Symfony\Component\Form\Exception\TransformationFailedException;
use Doctrine\Common\Persistence\ObjectManager;
use Acme\TaskBundle\Entity\Issue;

class IssueToNumberTransformer implements DataTransformerInterface
{
    /**
     * @var ObjectManager
     */
    private $om;

    /**
     * @param ObjectManager $om
     */
    public function __construct(ObjectManager $om)
    {
        $this->om = $om;
    }

    /**
     * Transforms an object (issue) to a string (number).
     *
     * @param  Issue|null $issue
     * @return string
     */
    public function transform($issue)
    {
        if (null === $issue) {
            return "";
        }

        return $issue->getNumber();
    }

    /**
     * Transforms a string (number) to an object (issue).
     *
     * @param  string $number
     * @return Issue|null
     * @throws TransformationFailedException if object (issue) is not found.
```

```
     */
    public function reverseTransform($number)
    {
        if (!$number) {
            return null;
        }

        $issue = $this->om
            ->getRepository('AcmeTaskBundle:Issue')
            ->findOneBy(array('number' => $number))
        ;

        if (null === $issue) {
            throw new TransformationFailedException(sprintf(
                'An issue with number "%s" does not exist!',
                $number
            ));
        }

        return $issue;
    }
}
```

Dica: Se você deseja que um novo *issue* seja criado quando um número desconhecido é informado, você pode instanciá-lo ao invés de gerar uma `TransformationFailedException`.

Usando o Transformador

Agora que você já tem o transformador construído, você só precisa adicioná-lo ao seu campo *issue* de alguma forma.

Você também pode usar transformadores sem criar um novo tipo de formulário personalizado chamando `addModelTransformer` (ou `addViewTransformer` - ver *Transformadores de Modelo e Visão*)) em qualquer builder de campo:

```
    use Symfony\Component\Form\FormBuilderInterface;
    use Acme\TaskBundle\Form\DataTransformer\IssueToNumberTransformer;

    class TaskType extends AbstractType
    {
        public function buildForm(FormBuilderInterface $builder, array $options)
        {
            // ...

            // this assumes that the entity manager was passed in as an option
            $entityManager = $options['em'];
            $transformer = new IssueToNumberTransformer($entityManager);

            // add a normal text field, but add our transformer to it
            $builder->add(
                $builder->create('issue', 'text')
                    ->addModelTransformer($transformer)
            );
        }

        // ...
```

```
    }
```

Este exemplo requer que você passe no gerenciador de entidade como uma opção ao criar o seu formulário. Mais tarde, você vai aprender como criar um tipo de campo `issue` personalizado para evitar ter de fazer isso no seu controlador:

```
$taskForm = $this->createForm(new TaskType(), $task, array(
    'em' => $this->getDoctrine()->getEntityManager(),
));
```

Legal, está feito! O usuário poderá informar um número de *issue* no campo texto e ele será transformado novamente em um objeto *Issue*. Isto significa que, após um *bind* bem sucedido, o framework de Formulário passará um objeto *Issue* real para o `Task::setIssue()` em vez do número do *issue*.

Se o *issue* não for encontrado, um erro de formulário será criado para esse campo e sua mensagem de erro pode ser controlada com a opção do campo `invalid_message`.

Cuidado: Note que a adição de um transformador exige a utilização de uma sintaxe um pouco mais complicada ao adicionar o campo. O código seguinte está **errado**, já que o transformador será aplicado à todo o formulário, em vez de apenas este campo:

```
// ISTO ESTÁ ERRADO - O TRANSFORMADOR SERÁ APLICADA A TODO O FORMULÁRIO
// Veja o exemplo acima para o código correto
$builder->add('issue', 'text')
    ->addModelTransformer($transformer);
```

Transformadores de Modelo e Visão Novo na versão 2.1: Os nomes e método de transformadores foram alterados no Symfony 2.1. `prependNormTransformer` tornou-se `addModelTransformer` e `appendClientTransformer` tornou-se `addViewTransformer`.

No exemplo acima, o transformador foi utilizado como um transformador de "modelo". De fato, existem dois tipos diferentes de transformadores e três tipos diferentes de dados subjacentes.

Em qualquer formulário, os 3 tipos de dados possíveis são os seguintes:

1) **Dados do modelo** - Estes são os dados no formato usado em sua aplicação (ex., um objeto `Issue`). Se você chamar `Form::getData` ou `Form::setData`, você está lidando com os dados do "modelo".

2) **Dados Normalizados** - Esta é uma versão normalizada de seus dados, e é comumente o mesmo que os dados do "modelo" (apesar de não no nosso exemplo). Geralmente ele não é usado diretamente.

3) **Dados da Visão** - Este é o formato que é usado para preencher os campos do formulário. É também o formato no qual o usuário irá enviar os dados. Quando você chama `Form::bind($data)`, o `$data` está no formato de dados da "visão".

Os 2 tipos diferentes de transformadores ajudam a converter de e para cada um destes tipos de dados:

Transformadores de Modelo:

- `transform`: "model data" => "norm data"
- `reverseTransform`: "norm data" => "model data"

Transformadores de Visão:

- `transform`: "norm data" => "view data"
- `reverseTransform`: "view data" => "norm data"

O transformador que você vai precisar depende de sua situação.

Para utilizar o transformador de visão, chame `addViewTransformer`.

Então, por que nós usamos o transformador de modelo?

No nosso exemplo, o campo é um campo `text`, e nós sempre esperamos que um campo texto seja um formato escalar simples, nos formatos "normalizado" e "visão". Por esta razão, o transformador mais apropriado é o transformador de "modelo" (que converte de/para o formato *normalizado* - número de issue string - para o formato *modelo* - objeto *Issue*).

A diferença entre os transformadores é sutil e você deve sempre pensar sobre o que o dado "normalizado" para um campo deve realmente ser. Por exemplo, o dado "normalizado" para um campo `text` é uma string, mas é um objeto `DateTime` para um campo `date`.

Usando Transformadores em um tipo de campo personalizado

No exemplo acima, você aplicou o transformador para um campo `text` normal. Isto foi fácil, mas tem duas desvantagens:

1) Você precisa lembrar de aplicar o transformador sempre que você está adicionando um campo para números de *issue*

2) Você precisa se preocupar em sempre passar a opção `em` quando você está criando um formulário que usa o transformador.

Devido à isto, você pode optar por criar um tipo de campo personalizado. Primeiro, crie a classe do tipo de campo personalizado:

```
// src/Acme/TaskBundle/Form/Type/IssueSelectorType.php
namespace Acme\TaskBundle\Form\Type;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Acme\TaskBundle\Form\DataTransformer\IssueToNumberTransformer;
use Doctrine\Common\Persistence\ObjectManager;
use Symfony\Component\OptionsResolver\OptionsResolverInterface;

class IssueSelectorType extends AbstractType
{
    /**
     * @var ObjectManager
     */
    private $om;

    /**
     * @param ObjectManager $om
     */
    public function __construct(ObjectManager $om)
    {
        $this->om = $om;
    }

    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $transformer = new IssueToNumberTransformer($this->om);
        $builder->addModelTransformer($transformer);
    }

    public function setDefaultOptions(OptionsResolverInterface $resolver)
    {
        $resolver->setDefaults(array(
```

```
            'invalid_message' => 'The selected issue does not exist',
        ));
    }

    public function getParent()
    {
        return 'text';
    }

    public function getName()
    {
        return 'issue_selector';
    }
}
```

Em seguida, registre o seu tipo como um serviço e use a tag `form.type`, para que ele seja reconhecido como um tipo de campo personalizado:

- *YAML*

```
    services:
        acme_demo.type.issue_selector:
            class: Acme\TaskBundle\Form\Type\IssueSelectorType
            arguments: ["@doctrine.orm.entity_manager"]
            tags:
                - { name: form.type, alias: issue_selector }
```

- *XML*

```
    <service id="acme_demo.type.issue_selector" class="Acme\TaskBundle\Form\Type\IssueSelectorType">
        <argument type="service" id="doctrine.orm.entity_manager"/>
        <tag name="form.type" alias="issue_selector" />
    </service>
```

Agora, sempre que você precisa usar o seu tipo de campo especial `issue_selector`, é muito fácil:

```
// src/Acme/TaskBundle/Form/Type/TaskType.php
namespace Acme\TaskBundle\Form\Type;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;

class TaskType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('task')
            ->add('dueDate', null, array('widget' => 'single_text'));
            ->add('issue', 'issue_selector');
    }

    public function getName()
    {
        return 'task';
    }
}
```

Como Modificar Formulários dinamicamente usando Eventos de Formulário

Antes de saltar diretamente para a geração dinâmica de formulário, vamos fazer uma revisão rápida do como uma classe de formulário parece:

```
// src/Acme/DemoBundle/Form/Type/ProductType.php
namespace Acme\DemoBundle\Form\Type;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;

class ProductType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder->add('name');
        $builder->add('price');
    }

    public function getName()
    {
        return 'product';
    }
}
```

Nota: Se esta parte de código em particular ainda não lhe é familiar, você provavelmente terá que dar um passo para trás e primeiro rever o capítulo Formulários antes de prosseguir.

Vamos assumir, por um momento, que este formulário utiliza uma classe imaginária "Product" que possui apenas duas propriedades relevantes ("name" e "price"). O formulário gerado desta classe terá exatamente a mesma aparência, independentemente se um novo "Product" está sendo criado ou se um produto já existente está sendo editado (por exemplo, um produto obtido a partir do banco de dados).

Suponha agora, que você não deseja que o usuário possa alterar o valor de `name` uma vez que o objeto foi criado. Para fazer isso, você pode contar com o sistema de `Dispatcher de Evento` do Symfony para analisar os dados sobre o objeto e modificar o formulário com base nos dados do objeto "Product". Neste artigo, você vai aprender como adicionar este nível de flexibilidade aos seus formulários.

Adicionando um Assinante (Subscriber) de evento à uma Classe de Formulário

Assim, em vez de adicionar diretamente o widget "name" através da sua classe de formulário ProductType, vamos delegar a responsabilidade de criar esse campo específico para um Assinante de Evento:

```
// src/Acme/DemoBundle/Form/Type/ProductType.php
namespace Acme\DemoBundle\Form\Type;

use Symfony\Component\Form\AbstractType
use Symfony\Component\Form\FormBuilderInterface;
use Acme\DemoBundle\Form\EventListener\AddNameFieldSubscriber;

class ProductType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $subscriber = new AddNameFieldSubscriber($builder->getFormFactory());
```

```
        $builder->addEventSubscriber($subscriber);
        $builder->add('price');
    }

    public function getName()
    {
        return 'product';
    }
}
```

É passado o objeto FormFactory ao construtor do assinante de evento para que o seu novo assinante seja capaz de criar o widget de formulário, uma vez que ele é notificado do evento despachado durante a criação do formulário.

Dentro da Classe do Assinante de Evento

O objetivo é criar um campo "name" *apenas* se o objeto Product subjacente é novo (por exemplo, não tenha sido persistido no banco de dados). Com base nisso, o assinante pode parecer com o seguinte:

```
// src/Acme/DemoBundle/Form/EventListener/AddNameFieldSubscriber.php
namespace Acme\DemoBundle\Form\EventListener;

use Symfony\Component\Form\Event\DataEvent;
use Symfony\Component\Form\FormFactoryInterface;
use Symfony\Component\EventDispatcher\EventSubscriberInterface;
use Symfony\Component\Form\FormEvents;

class AddNameFieldSubscriber implements EventSubscriberInterface
{
    private $factory;

    public function __construct(FormFactoryInterface $factory)
    {
        $this->factory = $factory;
    }

    public static function getSubscribedEvents()
    {
        // Tells the dispatcher that you want to listen on the form.pre_set_data
        // event and that the preSetData method should be called.
        return array(FormEvents::PRE_SET_DATA => 'preSetData');
    }

    public function preSetData(DataEvent $event)
    {
        $data = $event->getData();
        $form = $event->getForm();

        // During form creation setData() is called with null as an argument
        // by the FormBuilder constructor. You're only concerned with when
        // setData is called with an actual Entity object in it (whether new
        // or fetched with Doctrine). This if statement lets you skip right
        // over the null condition.
        if (null === $data) {
            return;
        }

        // check if the product object is "new"
```

```
        if (!$data->getId()) {
            $form->add($this->factory->createNamed('name', 'text'));
        }
    }
}
```

Cuidado: É fácil entender mal o propósito do segmento `if (null === $data)` deste assinante de evento. Para entender plenamente o seu papel, você pode considerar também verificar a classe Form e prestar atenção especial onde o setData() é chamado no final do construtor, bem como o método setData() em si.

A linha `FormEvents::PRE_SET_DATA` resolve para a string `form.pre_set_data`. A classe FormEvents serve para propósito organizacional. É um local centralizado em que você pode encontrar todos os vários eventos disponíveis.

Enquanto este exemplo poderia ter usado o evento `form.set_data` ou até mesmo o `form.post_set_data` com a mesma eficácia, usando o `form.pre_set_data` você garante que os dados que estão sendo recuperados do objeto `Event` não foram de modo algum modificados por quaisquer outros assinantes ou ouvintes. Isto é porque o `form.pre_set_data` passa um objeto DataEvent em vez do objcto FilterDataEvent passado pelo evento `form.set_data`. O DataEvent, ao contrário de seu filho FilterDataEvent, não tem um método setData().

Nota: Você pode ver a lista completa de eventos de formulário através da classe FormEvents, encontrada no bundle de formulário.

Como embutir uma Coleção de Formulários

Neste artigo, você vai aprender como criar um formulário que incorpora uma coleção de muitos outros formulários. Isto pode ser útil, por exemplo, se você tem uma classe `Task` onde você deseja editar/criar/remover muitos objetos `Tag` relacionados a Task, dentro do mesmo formulário.

Nota: Neste artigo, é livremente assumido que você está usando o Doctrine para armazenar em seu banco de dados. Mas se você não está usando o Doctrine (por exemplo, Propel ou apenas uma conexão de banco de dados), tudo é muito semelhante. Há apenas algumas partes deste tutorial que realmente se preocupam com "persistência".

Se você *está* usando o Doctrine, você vai precisar adicionar os metadados do Doctrine, incluindo a definição de mapeamento da associação `ManyToMany` na propriedade `tags` da Task.

Vamos começar: suponha que cada `Task` pertence a vários objetos `Tags`. Comece criando uma classe simples `Task`:

```
// src/Acme/TaskBundle/Entity/Task.php
namespace Acme\TaskBundle\Entity;

use Doctrine\Common\Collections\ArrayCollection;

class Task
{
    protected $description;

    protected $tags;

    public function __construct()
    {
        $this->tags = new ArrayCollection();
    }
```

```
    public function getDescription()
    {
        return $this->description;
    }

    public function setDescription($description)
    {
        $this->description = $description;
    }

    public function getTags()
    {
        return $this->tags;
    }

    public function setTags(ArrayCollection $tags)
    {
        $this->tags = $tags;
    }
}
```

Nota: O `ArrayCollection` é específico do Doctrine e é basicamente o mesmo que usar um `array` (mas deve ser um `ArrayCollection` se você está usando o Doctrine).

Agora, crie uma classe `Tag`. Como você viu acima, uma `Task` pode ter muitos objetos `Tag`:

```
// src/Acme/TaskBundle/Entity/Tag.php
namespace Acme\TaskBundle\Entity;

class Tag
{
    public $name;
}
```

Dica: A propriedade `name` é pública aqui, mas ela pode facilmente ser protegida ou privada (então seriam necessários os métodos `getName` e `setName`).

Agora, vamos para os formulários. Crie uma classe de formulário para que um objeto `Tag` possa ser modificado pelo usuário:

```
// src/Acme/TaskBundle/Form/Type/TagType.php
namespace Acme\TaskBundle\Form\Type;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolverInterface;

class TagType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder->add('name');
    }

    public function setDefaultOptions(OptionsResolverInterface $resolver)
```

```
    {
        $resolver->setDefaults(array(
            'data_class' => 'Acme\TaskBundle\Entity\Tag',
        ));
    }

    public function getName()
    {
        return 'tag';
    }
}
```

Com isso, você tem o suficiente para renderizar um formulário tag. Mas, uma vez que o objetivo final é permitir que as tags de uma `Task` sejam modificadas dentro do próprio formulário da task, crie um formulário para a classe `Task`.

Observe que você embutiu uma coleção de formulários `TagType` usando o tipo de campo `collection`:

```
// src/Acme/TaskBundle/Form/Type/TaskType.php
namespace Acme\TaskBundle\Form\Type;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolverInterface;

class TaskType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder->add('description');

        $builder->add('tags', 'collection', array('type' => new TagType()));
    }

    public function setDefaultOptions(OptionsResolverInterface $resolver)
    {
        $resolver->setDefaults(array(
            'data_class' => 'Acme\TaskBundle\Entity\Task',
        ));
    }

    public function getName()
    {
        return 'task';
    }
}
```

Em seu controlador, você irá agora inicializar uma nova instância do `TaskType`:

```
// src/Acme/TaskBundle/Controller/TaskController.php
namespace Acme\TaskBundle\Controller;

use Acme\TaskBundle\Entity\Task;
use Acme\TaskBundle\Entity\Tag;
use Acme\TaskBundle\Form\Type\TaskType;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class TaskController extends Controller
{
```

```
    public function newAction(Request $request)
    {
        $task = new Task();

        // dummy code - this is here just so that the Task has some tags
        // otherwise, this isn't an interesting example
        $tag1 = new Tag();
        $tag1->name = 'tag1';
        $task->getTags()->add($tag1);
        $tag2 = new Tag();
        $tag2->name = 'tag2';
        $task->getTags()->add($tag2);
        // end dummy code

        $form = $this->createForm(new TaskType(), $task);

        // process the form on POST
        if ($request->isMethod('POST')) {
            $form->bind($request);
            if ($form->isValid()) {
                // ... maybe do some form processing, like saving the Task and Tag objects
            }
        }

        return $this->render('AcmeTaskBundle:Task:new.html.twig', array(
            'form' => $form->createView(),
        ));
    }
}
```

O template correspondente agora é capaz de renderizar tanto o campo `description` para o formulário da task, quanto todos os formulários `TagType` para quaisquer tags que já estão relacionadas com esta `Task`. No controlador acima, foi adicionado algum código fictício para que você possa ver isso em ação (uma vez que uma `Task` não tem nenhuma tag quando ela é criada pela primeira vez).

- *Twig*

```
{# src/Acme/TaskBundle/Resources/views/Task/new.html.twig #}

{# ... #}

<form action="..." method="POST" {{ form_enctype(form) }}>
    {# render the task's only field: description #}
    {{ form_row(form.description) }}

    <h3>Tags</h3>
    <ul class="tags">
        {# iterate over each existing tag and render its only field: name #}
        {% for tag in form.tags %}
            <li>{{ form_row(tag.name) }}</li>
        {% endfor %}
    </ul>

    {{ form_rest(form) }}
    {# ... #}
</form>
```

- *PHP*

```
<!-- src/Acme/TaskBundle/Resources/views/Task/new.html.php -->

<!-- ... -->

<form action="..." method="POST" ...>
    <h3>Tags</h3>
    <ul class="tags">
        <?php foreach($form['tags'] as $tag): ?>
            <li><?php echo $view['form']->row($tag['name']) ?></li>
        <?php endforeach; ?>
    </ul>

    <?php echo $view['form']->rest($form) ?>
</form>

<!-- ... -->
```

Quando o usuário submeter o formulário, os dados submetidos para os campos `Tags` são usados para construir um ArrayCollection de objetos `Tag`, o qual é então definido no campo `tag` da instância `Task`.

A coleção `Tags` é acessível naturalmente via `$task->getTags()` e pode ser persistida no banco de dados ou utilizada da forma que você precisar.

Até agora, isso funciona muito bem, mas não permite que você adicione dinamicamente novas tags ou exclua as tags existentes. Então, enquanto a edição de tags existentes irá funcionar perfeitamente, o usuário não pode, ainda, adicionar quaisquer tags novas.

Cuidado: Neste artigo, você embutiu apenas uma coleção, mas você não está limitado a apenas isto. Você também pode incorporar coleção aninhada com a quantidade de níveis abaixo que desejar. Mas, se você usar o Xdebug em sua configuração de desenvolvimento, você pode receber erro `Maximum function nesting level of '100' reached, aborting!`. Isto ocorre devido a configuração do PHP `xdebug.max_nesting_level`, que tem como padrão `100`.
Esta diretiva limita recursão para 100 chamadas, o que pode não ser o suficiente para renderizar o formulário no template se você renderizar todo o formulário de uma vez (por exemplo, usando `form_widget(form)`). Para corrigir isso, você pode definir esta diretiva para um valor maior (através do arquivo ini do PHP ou via `ini_set`, por exemplo em `app/autoload.php`) ou renderizar cada campo do formulário manualmente usando `form_row`.

Permitindo "novas" tags com o "prototype"

Permitir ao usuário adicionar dinamicamente novas tags significa que você vai precisar usar algum JavaScript. Anteriormente, você adicionou duas tags ao seu formulário no controlador. Agora, para permitir ao usuário adicionar a quantidade de formulários tag que precisar diretamente no navegador, vamos utilizar um pouco de JavaScript.

A primeira coisa que você precisa fazer é tornar a coleção de formulário ciente de que ela vai receber um número desconhecido de tags. Até agora, você adicionou duas tags e o tipo formulário espera receber exatamente duas, caso contrário, um erro será lançado: `Este formulário não deve conter campos extras`. Para tornar isto flexível, adicione a opção `allow_add` no seu campo de coleção:

```
// src/Acme/TaskBundle/Form/Type/TaskType.php

// ...

use Symfony\Component\Form\FormBuilderInterface;

public function buildForm(FormBuilderInterface $builder, array $options)
```

```
{
    $builder->add('description');

    $builder->add('tags', 'collection', array(
        'type'         => new TagType(),
        'allow_add'    => true,
        'by_reference' => false,
    ));
}
```

Note que `'by_reference' => false` também foi adicionado. Normalmente, o framework de formulário irá modificar as tags em um objeto *Task sem* realmente nunca chamar *setTags*. Definindo by_reference para *false*, o *setTags* será chamado. Você verá que isto será importante mais tarde.

Além de dizer ao campo para aceitar qualquer número de objetos submetidos, o `allow_add` também disponibiliza para você uma variável "prototype". Este "prototype" é um "template" que contém todo o HTML para poder renderizar quaisquer formulários "tag" novos. Para renderizá-lo, faça a seguinte alteração no seu template:

- *Twig*

```
<ul class="tags" data-prototype="{{ form_widget(form.tags.vars.prototype)|e }}">
    ...
</ul>
```

- *PHP*

```
<ul class="tags" data-prototype="<?php echo $view->escape($view['form']->row($form['tags']->getV
    ...
</ul>
```

Nota: Se você renderizar todo o seu sub-formulário "tags" de uma vez (por exemplo `form_row(form.tags)`), então o prototype está automaticamente disponível na `div` externa, no atributo `data-prototype`, semelhante ao que você vê acima.

Dica: O `form.tags.vars.prototype` é um elemento de formulário com o aspecto semelhante aos elementos individuais `form_widget(tag)` dentro do seu laço `for`. Isso significa que você pode chamar `form_widget`, `form_row` ou `form_label` nele. Você pode até mesmo optar por renderizar apenas um de seus campos (por exemplo, o campo `name`):

```
{{ form_widget(form.tags.vars.prototype.name)|e }}
```

Na página renderizada, o resultado será algo parecido com o seguinte:

```
<ul class="tags" data-prototype="&lt;div&gt;&lt;label class=&quot; required&quot;&gt;__name__&lt;/lab
```

O objetivo desta seção será usar JavaScript para ler este atributo e dinamicamente adicionar novos formulários tag quando o usuário clicar no link "Adicionar uma tag". Para tornar as coisas simples, este exemplo usa jQuery e assume que você o incluiu em algum lugar na sua página.

Adicione uma tag `script` em algum lugar na sua página para que você possa começar a escrever um pouco de JavaScript.

Primeiro, adicione um link no final da lista "tags" via JavaScript. Segundo, faça o bind do evento "click" desse link para que você possa adicionar um novo formulário de tag (`addTagForm` será exibido em seguida):

```
// Get the ul that holds the collection of tags
var collectionHolder = $('ul.tags');

// setup an "add a tag" link
var $addTagLink = $('<a href="#" class="add_tag_link">Add a tag</a>');
var $newLinkLi = $('<li></li>').append($addTagLink);

jQuery(document).ready(function() {
    // add the "add a tag" anchor and li to the tags ul
    collectionHolder.append($newLinkLi);

    // count the current form inputs we have (e.g. 2), use that as the new
    // index when inserting a new item (e.g. 2)
    collectionHolder.data('index', collectionHolder.find(':input').length);

    $addTagLink.on('click', function(e) {
        // prevent the link from creating a "#" on the URL
        e.preventDefault();

        // add a new tag form (see next code block)
        addTagForm(collectionHolder, $newLinkLi);
    });
});
```

O trabalho da função `addTagForm` será usar o atributo `data-prototype` para adicionar dinamicamente um novo formulário quando é clicado neste link. O HTML `data-prototype` contém o elemento de entrada `text` com um nome de `task[tags][__name__][name]` e com o id `task_tags___name___name`. O nome `__name__` é um pequeno "placeholder", que você vai substituir por um número único, incrementado (por exemplo: `task[tags][3][name]`).

Novo na versão 2.1: O placeholder foi alterado de `$$name$$` para `__name__` no Symfony 2.1

O código real necessário para fazer todo este trabalho pode variar um pouco, mas aqui está um exemplo:

```
function addTagForm(collectionHolder, $newLinkLi) {
    // Get the data-prototype explained earlier
    var prototype = collectionHolder.data('prototype');

    // get the new index
    var index = collectionHolder.data('index');

    // Replace '__name__' in the prototype's HTML to
    // instead be a number based on the current collection's length.
    var newForm = prototype.replace(/__name__/g, collectionHolder.children().length);

    // increase the index with one for the next item
    collectionHolder.data('index', index + 1);

    // Display the form in the page in an li, before the "Add a tag" link li
    var $newFormLi = $('<li></li>').append(newForm);
    $newLinkLi.before($newFormLi);
}
```

Nota: É melhor separar o seu javascript em arquivos JavaScript do que escrevê-lo dentro do HTML como foi feito aqui.

Agora, cada vez que um usuário clicar no link `Adicionar uma tag`, um novo sub-formulário vai aparecer na

página. Quando o formulário é submetido, todos os novos formulários de tag serão convertidos em novos objetos `Tag` e adicionados à propriedade `tags` do objeto `Task`.

Doctrine: Relações em Cascata e salvando o lado "Inverso"

Para obter as novas tags para salvar no Doctrine, é preciso considerar algumas coisas a mais. Em primeiro lugar, a menos que você iterar sobre todos os novos objetos `Tag` e chamar `$em->persist($tag)` em cada um, você receberá um erro do Doctrine:

> Uma nova entidade foi encontrada através da relação *AcmeTaskBundleEntityTask#tags* que não foi configurada para operações de persistir em cascata para a entidade...

Para corrigir isso, você pode optar pela operação de persistir em "cascata" automaticamente a partir do objeto `Task` para todas as tags relacionadas. Para fazer isso, adicione a opção `cascade` em seu metadado `ManyToMany`:

- *Annotations*

```
// src/Acme/TaskBundle/Entity/Task.php

// ...

/**
 * @ORM\ManyToMany(targetEntity="Tag", cascade={"persist"})
 */
protected $tags;
```

- *YAML*

```
# src/Acme/TaskBundle/Resources/config/doctrine/Task.orm.yml
Acme\TaskBundle\Entity\Task:
    type: entity
    # ...
    oneToMany:
        tags:
            targetEntity: Tag
            cascade:      [persist]
```

- *XML*

```
<!-- src/Acme/TaskBundle/Resources/config/doctrine/Task.orm.xml -->
<doctrine-mapping xmlns="http://doctrine-project.org/schemas/orm/doctrine-mapping"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://doctrine-project.org/schemas/orm/doctrine-mapping
                    http://doctrine-project.org/schemas/orm/doctrine-mapping.xsd">

    <entity name="Acme\TaskBundle\Entity\Task" ...>
        <!-- ... -->
        <one-to-many field="tags" target-entity="Tag">
            <cascade>
                <cascade-persist />
            </cascade>
        </one-to-many>
    </entity>
</doctrine-mapping>
```

Um segundo problema potencial aborda o Lado Proprietário e Lado Inverso dos relacionamentos do Doctrine. Neste exemplo, se o lado "proprietário" da relação é "Task", então a persistência irá funcionar bem pois as tags são devidamente adicionadas à Task. No entanto, se o lado proprietário é a "Tag", então você vai ter um pouco mais de trabalho para garantir que o lado correto da relação será modificado.

O truque é ter certeza de que uma única "Task" é definida em cada "Tag". Uma maneira fácil de fazer isso é adicionar alguma lógica extra ao `setTags()`, que é chamada pelo framework de formulário desde que by_reference esteja definido como `false`:

```
// src/Acme/TaskBundle/Entity/Task.php

// ...

public function setTags(ArrayCollection $tags)
{
    foreach ($tags as $tag) {
        $tag->addTask($this);
    }
```

Permitindo que as tags sejam removidas

O passo seguinte é permitir a remoção de um item em particular na coleção. A solução é similar a que permite que as tags sejam adicionadas.

Comece adicionando a opção `allow_delete` no tipo do formulário:

```
// src/Acme/TaskBundle/Form/Type/TaskType.php

// ...

public function buildForm(FormBuilder $builder, array $options)
{
    $builder->add('description');

    $builder->add('tags', 'collection', array(
        'type'         => new TagType(),
        'allow_add'    => true,
        'allow_delete' => true,
        'by_reference' => false,
    ));
}
```

Modificações nos Templates A opção `allow_delete` tem uma consequência: se um item de uma coleção não for enviado na submissão, o dado relacionado é removido da coleção no servidor. A solução é, portanto, remover o elemento de formulário do DOM.

Primeiro, adicione um link "excluir esta tag" para cada formulário de tag:

```
jQuery(document).ready(function() {
    // add a delete link to all of the existing tag form li elements
    collectionHolder.find('li').each(function() {
        addTagFormDeleteLink($(this));
    });

    // ... the rest of the block from above
});

function addTagForm() {
    // ...

    // add a delete link to the new form
    addTagFormDeleteLink($newFormLi);
}
```

A função `addTagFormDeleteLink` será parecida com a seguinte:

```
function addTagFormDeleteLink($tagFormLi) {
    var $removeFormA = $('<a href="#">delete this tag</a>');
    $tagFormLi.append($removeFormA);

    $removeFormA.on('click', function(e) {
        // prevent the link from creating a "#" on the URL
        e.preventDefault();

        // remove the li for the tag form
        $tagFormLi.remove();
```

```
    });
}
```

Quando um formulário de tag é removido do DOM e submetido, o objeto `Tag` removido não será incluído na coleção passada ao `setTags`. Dependendo de sua camada de persistência, isto pode ou não ser o suficiente para remover efetivamente a relação entre os objetos `Tag` e `Task`.

Doctrine: Garantir a persistência de dados

Ao remover objetos dessa forma, você pode precisar fazer um pouco mais de trabalho para garantir que a relação entre a Task e Tag seja removida adequadamente.
No Doctrine, você tem dois lados da relação: o lado proprietário e o lado inverso. Normalmente, neste caso, você vai ter uma relação ManyToMany e as tags excluídas desaparecerão e será persistido corretamente (a adição de novas tags também funciona sem esforço).
Mas, se você tem uma relação `OneToMany` ou uma `ManyToMany` com um `mappedBy` na entidade Task (significando que Task é o lado "inverso"), você vai ter mais trabalho para que as tags removidas persistam corretamente.
Neste caso, você pode modificar o controlador para remover a relação na tag removida. Isso pressupõe que você tenha algum `editAction` que está lidando com a "atualização" da sua Task:

```
// src/Acme/TaskBundle/Controller/TaskController.php

// ...

public function editAction($id, Request $request)
{
    $em = $this->getDoctrine()->getManager();
    $task = $em->getRepository('AcmeTaskBundle:Task')->find($id);

    if (!$task) {
        throw $this->createNotFoundException('No task found for is '.$id);
    }

    $originalTags = array();

    // Create an array of the current Tag objects in the database
    foreach ($task->getTags() as $tag) $originalTags[] = $tag;

    $editForm = $this->createForm(new TaskType(), $task);

    if ($request->isMethod('POST')) {
        $editForm->bind($this->getRequest());

        if ($editForm->isValid()) {

            // filter $originalTags to contain tags no longer present
            foreach ($task->getTags() as $tag) {
                foreach ($originalTags as $key => $toDel) {
                    if ($toDel->getId() === $tag->getId()) {
                        unset($originalTags[$key]);
                    }
                }
            }

            // remove the relationship between the tag and the Task
            foreach ($originalTags as $tag) {
                // remove the Task from the Tag
                $tag->getTasks()->removeElement($task);

                // if it were a ManyToOne relationship, remove the relationship like thi
                // $tag->setTask(null);

                $em->persist($tag);

                // if you wanted to delete the Tag entirely, you can also do that
                // $em->remove($tag);
            }

            $em->persist($task);
            $em->flush();

            // redirect back to some edit page
```

Como Criar um Tipo de Campo de Formulário Personalizado

O Symfony vem com vários tipos de campo disponíveis para a construção de formulários. No entanto, existem situações em que você pode desejar criar um tipo de campo de formulário personalizado para um propósito específico. Esta receita assume que você precisa de uma definição de campo que possui o gênero de uma pessoa, com base no campo choice existente. Esta seção explica como o campo é definido, como você pode personalizar o seu layout e, finalmente, como registrá-lo para uso em sua aplicação.

Definindo o Tipo de Campo

A fim de criar o tipo de campo personalizado, primeiro você precisa criar a classe que o representa. Nesta situação, a classe que contém o tipo de campo será chamada *GenderType* e o arquivo será armazenado no local padrão para campos de formulário, que é `<BundleName>\Form\Type`. Verifique se o campo estende a classe `AbstractType`:

```
// src/Acme/DemoBundle/Form/Type/GenderType.php
namespace Acme\DemoBundle\Form\Type;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\OptionsResolver\OptionsResolverInterface;

class GenderType extends AbstractType
{
    public function setDefaultOptions(OptionsResolverInterface $resolver)
    {
        $resolver->setDefaults(array(
            'choices' => array(
                'm' => 'Male',
                'f' => 'Female',
            )
        ));
    }

    public function getParent()
    {
        return 'choice';
    }

    public function getName()
    {
        return 'gender';
    }
}
```

Dica: A localização deste arquivo não é importante - o diretório `Form\Type` é apenas uma convenção.

Aqui, o valor de retorno da função `getParent` indica que você está estendendo o tipo de campo `choice`. Isto significa que, por padrão, você herda toda a lógica e renderização deste tipo de campo. Para ver um pouco da lógica, confira a classe ChoiceType. Existem três métodos que são particularmente importantes:

- `buildForm()` - Cada tipo de campo possui um método `buildForm`, que é onde você configura e constrói qualquer campo(s). Note que este é o mesmo método que você usa para configurar *seus* formulários, e ele funciona da mesma forma aqui.
- `buildView()` - Este método é usado para definir quaisquer variáveis extras que você precisa ao renderizar o seu campo em um template. Por exemplo, no ChoiceType, uma variável `multiple` é definida e utilizada

no template para setar (ou não) o atributo `multiple` no campo `select`. Veja *Criando um template para o Campo* para mais detalhes.

- `setDefaultOptions()` - Define opções para o seu tipo do formulário, que podem ser usadas no `buildForm()` e no `buildView()`. Há várias opções comuns a todos os campos (veja `/reference/forms/types/form`), mas você pode criar quaisquer outras que você precisar aqui.

Dica: Se você está criando um campo que consiste de muitos campos, então não se esqueça de definir o seu tipo "pai" como `form` ou algo que estenda `form`. Além disso, se você precisar modificar a "visão" de qualquer um dos tipos filho a partir de seu tipo pai, use o método `finishView()`.

O método `getName()` retorna um identificador que deve ser único na sua aplicação. Isto é usado em vários lugares, como ao personalizar a forma que o seu tipo de formulário será renderizado.

O objetivo deste campo foi estender o tipo choice para ativar a seleção de um gênero. Isto é alcançado através do ajuste das `choices` para uma lista de gêneros possíveis.

Criando um Template para o Campo

Cada tipo de campo é renderizado por um fragmento de template, o qual é determinado, em parte, pelo valor do seu método `getName()`. Para maiores informações, visite *O que são Temas de Formulário?*.

Neste caso, uma vez que o campo pai é `choice`, você não *precisa* fazer qualquer trabalho pois o tipo de campo personalizado será automaticamente renderizado como um tipo `choice`. Mas, para o propósito deste exemplo, vamos supor que, quando o seu campo é "expandido" (ou seja, botões de opção ou caixas de seleção em vez de um campo de seleção), você quer sempre renderizá-lo em um elemento `ul`. Em seu template tema de formulário (veja o link acima para mais detalhes), crie um bloco `gender_widget` para lidar com isso:

```
{# src/Acme/DemoBundle/Resources/views/Form/fields.html.twig #}
{% block gender_widget %}
    {% spaceless %}
        {% if expanded %}
            <ul {{ block('widget_container_attributes') }}>
            {% for child in form %}
                <li>
                    {{ form_widget(child) }}
                    {{ form_label(child) }}
                </li>
            {% endfor %}
            </ul>
        {% else %}
            {# just let the choice widget render the select tag #}
            {{ block('choice_widget') }}
        {% endif %}
    {% endspaceless %}
{% endblock %}
```

Nota: Certifique-se que é usado o prefixo widget correto. Neste exemplo, o nome deve ser `gender_widget`, de acordo com o valor retornado pelo `getName`. Além disso, o arquivo de configuração principal deve apontar para o template de formulário personalizado, assim, ele será usado ao renderizar todos os formulários.

```
# app/config/config.yml
twig:
    form:
```

```
        resources:
            - 'AcmeDemoBundle:Form:fields.html.twig'
```

Usando o Tipo de Campo

Agora você pode usar o seu tipo de campo personalizado imediatamente, simplesmente criando uma nova instância do tipo em um de seus formulários:

```
// src/Acme/DemoBundle/Form/Type/AuthorType.php
namespace Acme\DemoBundle\Form\Type;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;

class AuthorType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder->add('gender_code', new GenderType(), array(
            'empty_value' => 'Choose a gender',
        ));
    }
}
```

Mas isso só funciona porque o `GenderType()` é muito simples. E se os códigos do gênero foram armazenados em configuração ou num banco de dados? A próxima seção explica como os tipos de campos mais complexos resolvem este problema.

Criando o seu Tipo de Campo como um Serviço

Até agora, este artigo assumiu que você tem um tipo de campo personalizado bem simples. Mas se você precisar acessar a configuração, uma conexão de banco de dados ou algum outro serviço, então, você vai querer registrar o seu tipo personalizado como um serviço. Por exemplo, suponha que você está armazenando os parâmetros de gênero em configuração:

- *YAML*

```
    # app/config/config.yml
    parameters:
        genders:
            m: Male
            f: Female
```

- *XML*

```
    <!-- app/config/config.xml -->
    <parameters>
        <parameter key="genders" type="collection">
            <parameter key="m">Male</parameter>
            <parameter key="f">Female</parameter>
        </parameter>
    </parameters>
```

Para usar o parâmetro, defina o seu tipo de campo personalizado como um serviço, injetando o valor do parâmetro `genders` como o primeiro argumento para a sua função recém-criada `__construct`:

- *YAML*

```yaml
# src/Acme/DemoBundle/Resources/config/services.yml
services:
    acme_demo.form.type.gender:
        class: Acme\DemoBundle\Form\Type\GenderType
        arguments:
            - "%genders%"
        tags:
            - { name: form.type, alias: gender }
```

- *XML*

```xml
<!-- src/Acme/DemoBundle/Resources/config/services.xml -->
<service id="acme_demo.form.type.gender" class="Acme\DemoBundle\Form\Type\GenderType">
    <argument>%genders%</argument>
    <tag name="form.type" alias="gender" />
</service>
```

Dica: Certifique-se que o arquivo de serviços está sendo importado. Para mais detalhes consulte *Importando configuração com imports*.

Certifique-se também que o atributo `alias` da tag corresponde ao valor retornado pelo método `getName` definido anteriormente. Você vai ver a importância disto logo que usar o tipo de campo personalizado. Mas, primeiro, adicione um método `__construct` para o `GenderType`, o qual recebe a configuração do gênero:

```php
// src/Acme/DemoBundle/Form/Type/GenderType.php
namespace Acme\DemoBundle\Form\Type;

use Symfony\Component\OptionsResolver\OptionsResolverInterface;

// ...

class GenderType extends AbstractType
{
    private $genderChoices;

    public function __construct(array $genderChoices)
    {
        $this->genderChoices = $genderChoices;
    }

    public function setDefaultOptions(OptionsResolverInterface $resolver)
    {
        $resolver->setDefaults(array(
            'choices' => $this->genderChoices,
        ));
    }

    // ...
}
```

Ótimo! O `GenderType` é alimentado agora por parâmetros de configuração e registrado como um serviço. Além disso, devido a você ter usado o alias `form.type` na sua configuração, a utilização do campo é muito mais fácil agora:

```
// src/Acme/DemoBundle/Form/Type/AuthorType.php
namespace Acme\DemoBundle\Form\Type;

use Symfony\Component\Form\FormBuilderInterface;

// ...

class AuthorType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder->add('gender_code', 'gender', array(
            'empty_value' => 'Choose a gender',
        ));
    }
}
```

Observe que em vez de criar uma nova instância, você pode apenas referir-se à ela pelo alias usado na sua configuração do serviço, `gender`. Divirta-se!

Como criar uma Extensão do Tipo de Formulário

Tipos de campo de formulário personalizados são ótimos quando você precisa de tipos de campo para um propósito específico, por exemplo para selecionar o gênero ou uma entrada de número de IVA.

Mas, às vezes, você realmente não precisa adicionar novos tipos de campo - você quer adicionar funcionalidades aos tipos existentes. É onde o tipo de formulário é usado.

As extensões do tipo de formulário têm dois casos de uso principais:

1. Você quer adicionar **uma funcionalidade genérica para vários tipos** (como a adição de um texto de "ajuda" para cada tipo de campo);
2. Você quer adicionar **uma funcionalidade específica para um único tipo** (tal como a adição de uma funcionalidade "download" para o tipo de campo "file").

Em ambos os casos, é possível alcançar o seu objetivo com a renderização personalizada de formulário. Mas, usar extensões do tipo de formulário pode ser mais limpo (limitando a quantidade de lógica de negócios nos templates) e mais flexível (você pode adicionar várias extensões do tipo a um único tipo de formulário.

Extensões do tipo de formulário podem alcançar mais do que os tipos de campos personalizados podem fazer, mas, em vez de serem tipos de campos próprios, **elas conectam em tipos existentes**.

Imagine que você gerencia uma entidade `Media`, e que cada mídia está associada a um arquivo. Seu formulário `Media` usa um tipo file, mas, ao editar a entidade, você gostaria de ver sua imagem processada automaticamente ao lado do campo arquivo.

Você poderia, naturalmente, fazer isso personalizando a forma como este campo é renderizado em um template. Mas as extensões do tipo de formulário permitem que você faça isso de uma forma DRY agradável.

Definindo a Extensão do Tipo de Formulário

Sua primeira tarefa será criar a classe da extensão do tipo de formulário. Vamos chamá-la `ImageTypeExtension`. Por padrão, as extensões de formulário geralmente residem no diretório `Form\Extension` de um de seus bundles.

Ao criar uma extensão de tipo de formulário, você pode implementar a interface `FormTypeExtensionInterface` ou estender a classe `AbstractTypeExtension`. Na maioria dos casos, é mais fácil estender a classe abstrata:

```
// src/Acme/DemoBundle/Form/Extension/ImageTypeExtension.php
namespace Acme\DemoBundle\Form\Extension;

use Symfony\Component\Form\AbstractTypeExtension;

class ImageTypeExtension extends AbstractTypeExtension
{
    /**
     * Returns the name of the type being extended.
     *
     * @return string The name of the type being extended
     */
    public function getExtendedType()
    {
        return 'file';
    }
}
```

O único método que você **deve** implementar é o `getExtendedType`. Ele é usado para indicar o nome do tipo de formulário que será estendido pela sua extensão.

Dica: O valor que você retorna no método `getExtendedType` corresponde ao valor retornado pelo método `getName` na classe do tipo de formulário que você deseja estender.

Além da função‘‘getExtendedType‘‘, provavelmente você vai querer sobrescrever um dos seguintes métodos:

- `buildForm()`
- `buildView()`
- `setDefaultOptions()`
- `finishView()`

Para mais informações sobre o que esses métodos fazem, você pode consultar o artigo do cookbook Criando Tipos de Campo Personalizados.

Registrando a sua Extensão do Tipo de Formulário como um Serviço

O próximo passo tornar o Symfony ciente da sua extensão. Tudo o que você precisa fazer é declará-la como um serviço usando a tag `form.type_extension`.

- *YAML*

```
services:
    acme_demo_bundle.image_type_extension:
        class: Acme\DemoBundle\Form\Extension\ImageTypeExtension
        tags:
            - { name: form.type_extension, alias: file }
```

- *XML*

```
<service id="acme_demo_bundle.image_type_extension"
    class="Acme\DemoBundle\Form\Extension\ImageTypeExtension"
>
    <tag name="form.type_extension" alias="file" />
</service>
```

- *PHP*

```
$container
    ->register(
        'acme_demo_bundle.image_type_extension',
        'Acme\DemoBundle\Form\Extension\ImageTypeExtension'
    )
    ->addTag('form.type_extension', array('alias' => 'file'));
```

A chave `alias` da tag é o tipo de campo que essa extensão deve ser aplicada. No seu caso, como você deseja estender o tipo de `file`, você vai usar o `file` como um alias.

Adicionando a lógica de negócio da extensão

O objetivo da sua extensão é exibir imagens agradáveis ao lado de campos arquivo (quando o modelo subjacente contém imagens). Para esta finalidade, vamos supor que você usa uma abordagem semelhante à descrita em Como manusear o upload de arquivos com o Doctrine: você tem um modelo Media com uma propriedade file (correspondente ao campo de arquivo, no formulário) e uma propriedade path (correspondendo ao caminho da imagem, no banco de dados):

```
// src/Acme/DemoBundle/Entity/Media.php
namespace Acme\DemoBundle\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Media
{
    // ...

    /**
     * @var string The path - typically stored in the database
     */
    private $path;

    /**
     * @var \Symfony\Component\HttpFoundation\File\UploadedFile
     * @Assert\File(maxSize="2M")
     */
    public $file;

    // ...

    /**
     * Get the image url
     *
     * @return null|string
     */
    public function getWebPath()
    {
        // ... $webPath being the full image url, to be used in templates

        return $webPath;
    }
}
```

Sua classe da extensão do tipo de formulário terá que fazer duas coisas, a fim de estender o tipo de formulário `file`:

1. Sobrescrever o método `setDefaultOptions`, a fim de adicionar uma opção image_path;

2. Sobrescrever os métodos `buildForm` e `buildView` a fim de passar a url da imagem para a visão.

A lógica é a seguinte: quando adicionar um campo de formulário do tipo `file`, você poderá especificar uma nova opção: ‘‘ image_path ‘‘. Esta opção irá dizer ao campo arquivo como obter o endereço real da imagem, a fim de exibí-la na visão:

```
// src/Acme/DemoBundle/Form/Extension/ImageTypeExtension.php
namespace Acme\DemoBundle\Form\Extension;

use Symfony\Component\Form\AbstractTypeExtension;
use Symfony\Component\Form\FormView;
use Symfony\Component\Form\FormInterface;
use Symfony\Component\Form\Util\PropertyPath;
use Symfony\Component\OptionsResolver\OptionsResolverInterface;

class ImageTypeExtension extends AbstractTypeExtension
{
    /**
     * Returns the name of the type being extended.
     *
     * @return string The name of the type being extended
     */
    public function getExtendedType()
    {
        return 'file';
    }

    /**
     * Add the image_path option
     *
     * @param OptionsResolverInterface $resolver
     */
    public function setDefaultOptions(OptionsResolverInterface $resolver)
    {
        $resolver->setOptional(array('image_path'));
    }

    /**
     * Pass the image url to the view
     *
     * @param FormView $view
     * @param FormInterface $form
     * @param array $options
     */
    public function buildView(FormView $view, FormInterface $form, array $options)
    {
        if (array_key_exists('image_path', $options)) {
            $parentData = $form->getParent()->getData();

            if (null !== $parentData) {
                $propertyPath = new PropertyPath($options['image_path']);
                $imageUrl = $propertyPath->getValue($parentData);
            } else {
                 $imageUrl = null;
            }

            // set an "image_url" variable that will be available when rendering this field
            $view->set('image_url', $imageUrl);
        }
```

```
    }

}
```

Sobrescrevendo o Fragmento de Template do Widget File

Cada tipo de campo é renderizado por um fragmento de template. Esses fragmentos de template podem ser sobrescritos, para personalizar a renderização formulário. Para mais informações você pode consultar o artigo *O que são Temas de Formulário?*.

Em sua classe de extensão, você adicionou uma nova variável (`image_url`), mas você ainda precisa aproveitar esta nova variável em seus templates. Especificamente, você precisa sobrescrever o bloco `file_widget`:

- *Twig*

```
{# src/Acme/DemoBundle/Resources/views/Form/fields.html.twig #}
{% extends 'form_div_layout.html.twig' %}

{% block file_widget %}
    {% spaceless %}

    {{ block('form_widget') }}
    {% if image_url is not null %}
        <img src="{{ asset(image_url) }}"/>
    {% endif %}

    {% endspaceless %}
{% endblock %}
```

- *PHP*

```
<!-- src/Acme/DemoBundle/Resources/views/Form/file_widget.html.php -->
<?php echo $view['form']->widget($form) ?>
<?php if (null !== $image_url): ?>
    <img src="<?php echo $view['assets']->getUrl($image_url) ?>"/>
<?php endif ?>
```

Nota: Você precisará mudar o seu arquivo de configuração ou especificar explicitamente como você quer o tema do seu formulário, para que o Symfony utilize o seu bloco sobrecrito. Veja *O que são Temas de Formulário?* para mais informações.

Usando a Extensão do Tipo de Formulário

A partir de agora, ao adicionar um campo do tipo `file` no seu formulário, você pode especificar uma opção `image_path` que será usada para exibir uma imagem próxima ao campo arquivo. Por exemplo:

```
// src/Acme/DemoBundle/Form/Type/MediaType.php
namespace Acme\DemoBundle\Form\Type;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;

class MediaType extends AbstractType
{
```

```
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('name', 'text')
            ->add('file', 'file', array('image_path' => 'webPath'));
    }

    public function getName()
    {
        return 'media';
    }
}
```

Ao exibir o formulário, se o modelo subjacente já foi associado com uma imagem, você vai vê-la ao lado do campo arquivo.

Como usar a opção de campo de formulário `Virtual`

A opção de campo de formulário `virtual` pode ser muito útil quando você possui alguns campos duplicados em entidades diferentes.

Por exemplo, imagine que você tem duas entidades:`Company` e `Customer`:

```
// src/Acme/HelloBundle/Entity/Company.php
namespace Acme\HelloBundle\Entity;

class Company
{
    private $name;
    private $website;

    private $address;
    private $zipcode;
    private $city;
    private $country;
}
```

```
// src/Acme/HelloBundle/Entity/Customer.php
namespace Acme\HelloBundle\Entity;

class Customer
{
    private $firstName;
    private $lastName;

    private $address;
    private $zipcode;
    private $city;
    private $country;
}
```

Como pode-se ver, as entidades possuem alguns campos iguais: `address`, `zipcode`, `city` e `country`.

Agora, você deseja construir dois formulários: um para `Company` e outro para `Customer`.

Comece criando classes simples de tipo de formulário para `CompanyType` e `CustomerType`:

```
// src/Acme/HelloBundle/Form/Type/CompanyType.php
namespace Acme\HelloBundle\Form\Type;

use Symfony\Component\Form\FormBuilderInterface;

class CompanyType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('name', 'text')
            ->add('website', 'text');
    }
}
```

```
// src/Acme/HelloBundle/Form/Type/CustomerType.php
namespace Acme\HelloBundle\Form\Type;

use Symfony\Component\Form\FormBuilderInterface;

class CustomerType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('firstName', 'text')
            ->add('lastName', 'text');
    }
}
```

Agora, temos que lidar com os quatro campos duplicados. Aqui está um formulário (simples) para localidade (`Location`):

```
// src/Acme/HelloBundle/Form/Type/LocationType.php
namespace Acme\HelloBundle\Form\Type;

use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolverInterface;

class LocationType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('address', 'textarea')
            ->add('zipcode', 'text')
            ->add('city', 'text')
            ->add('country', 'text');
    }

    public function setDefaultOptions(OptionsResolverInterface $resolver)
    {
        $resolver->setDefaults(array(
            'virtual' => true
        ));
    }

    public function getName()
```

```
    {
        return 'location';
    }
}
```

Nós não temos *realmente* um campo de localidade em cada uma das nossas entidades, de modo que não podemos ligar diretamente `LocationType` ao nosso `CompanyType` ou `CustomerType`. Mas, com certeza, queremos um tipo de formulário próprio para lidar com a localidade (lembre-se, DRY!).

A opção de campo de formulário `virtual` é a solução.

Podemos definir a opção `'virtual' => true` no método `setDefaultOptions()` da `LocationType` e começar a usá-lo diretamente nos dois tipos de formulários originais.

Verifique o resultado:

```
// CompanyType
public function buildForm(FormBuilderInterface $builder, array $options)
{
    $builder->add('foo', new LocationType(), array(
        'data_class' => 'Acme\HelloBundle\Entity\Company'
    ));
}
```

```
// CustomerType
public function buildForm(FormBuilderInterface $builder, array $options)
{
    $builder->add('bar', new LocationType(), array(
        'data_class' => 'Acme\HelloBundle\Entity\Customer'
    ));
}
```

Com a opção virtual definida para false (comportamento padrão) , o Componente de Formulário espera que cada objeto subjacente tenha uma propriedade `foo` (ou `bar`) que é algum objeto ou array que contém os quatro campos da localidade. Claro, não temos este objeto/array em nossas entidades e nós não queremos isso!

Com a opção virtual definida como true, o componente de Formulário ignora a propriedade `foo` (ou `bar`), e, em vez disso, aplica "gets" e "sets" aos quatro campos de localidade diretamente no objeto subjacente!

Nota: Ao invés de definir a opção `virtual` dentro de `LocationType`, você pode (assim como com todas as outras opções) também passá-la como uma opção de array no terceiro argumento de `$builder->add()`.

Como configurar Dados Vazios para uma Classe de Formulário

A opção `empty_data` permite que você especifique um conjunto de dados vazios para a sua classe de formulário. Este conjunto de dados vazios será usado se você fez o bind do seu formulário, mas não chamou o `setData()` nele ou não passou dados quando criou ele. Por exemplo:

```
public function indexAction()
{
    $blog = // ...

    // $blog is passed in as the data, so the empty_data option is not needed
    $form = $this->createForm(new BlogType(), $blog);

    // no data is passed in, so empty_data is used to get the "starting data"
```

```
    $form = $this->createForm(new BlogType());
}
```

Por padrão, o `empty_data` é setado como `null`. Ou, se você especificou a opção `data_class` para a sua classe de formulário, ele será, por padrão, uma nova instância dessa classe. Essa instância será criada chamando o construtor sem argumentos.

Se você quiser sobrescrever esse comportamento padrão, existem duas formas de fazer isso.

Opção 1: Instanciar uma nova Classe

Uma razão para usar esta opção é se você quer usar um construtor que possui argumentos. Lembre-se, a opção `data_class` padrão chama o construtor sem argumentos:

```
// src/Acme/DemoBundle/Form/Type/BlogType.php
// ...

use Symfony\Component\Form\AbstractType;
use Acme\DemoBundle\Entity\Blog;
use Symfony\Component\OptionsResolver\OptionsResolverInterface;

class BlogType extends AbstractType
{
    private $someDependency;

    public function __construct($someDependency)
    {
        $this->someDependency = $someDependency;
    }
    // ...

    public function setDefaultOptions(OptionsResolverInterface $resolver)
    {
        $resolver->setDefaults(array(
            'empty_data' => new Blog($this->someDependency),
        ));
    }
}
```

Você pode instanciar sua classe como desejar. Neste exemplo, nós passamos algumas dependências para o `BlogType` ao instanciá-lo, então, use isso para instanciar o objeto `Blog`. O ponto é, você pode setar o `empty_data` para o objeto "novo" que você deseja usar.

Opção 2: Fornecer uma Closure

Usar uma closure é o método preferido, uma vez que irá criar o objeto apenas se for necessário.

A closure deve aceitar uma instância `FormInterface` como seu primeiro argumento:

```
use Symfony\Component\OptionsResolver\OptionsResolverInterface;
use Symfony\Component\Form\FormInterface;
// ...

public function setDefaultOptions(OptionsResolverInterface $resolver)
{
    $resolver->setDefaults(array(
```

```
        'empty_data' => function (FormInterface $form) {
            return new Blog($form->get('title')->getData());
        },
    ));
}
```

3.1.7 Validação

Como criar uma Constraint de Validação Personalizada

Você pode criar uma constraint personalizada estendendo uma classe base de constraint `Constraint`. Como exemplo vamos criar um validador simples que verifica se uma string contém apenas caracteres alfanuméricos.

Criando a Classe Constraint

Primeiro você precisa criar uma classe de Constraint e estender `Constraint`:

```
// src/Acme/DemoBundle/Validator/constraints/ContainsAlphanumeric.php
namespace Acme\DemoBundle\Validator\Constraints;

use Symfony\Component\Validator\Constraint;

/**
 * @Annotation
 */
class ContainsAlphanumeric extends Constraint
{
    public $message = 'The string "%string%" contains an illegal character: it can only contain lette
}
```

Nota: A annotation `@Annotation` é necessária para esta nova constraint para torná-la disponível para uso em classes através de annotations. Opções para a sua constraint são representadas como propriedades públicas na classe constraint.

Criando o Validador em si

Como você pode ver, uma classe de constraint é muito curta. A validação real é realizada por uma outra classe "validadora de constraint". A classe validadora de constraint é especificada pelo método de constraint `validatedBy()`, que inclui alguma lógica padrão simples:

```
// in the base Symfony\Component\Validator\Constraint class
public function validatedBy()
{
    return get_class($this).'Validator';
}
```

Em outras palavras, se você criar uma `Constraint` personalizada (Ex. `MyConstraint`), o Symfony2 automaticamente irá procurar a outra classe `MyConstraintValidator` quando realmente executar a validação.

A classe validadora também é simples, e só contém um método necessário: `validate`:

```
// src/Acme/DemoBundle/Validator/Constraints/ContainsAlphanumericValidator.php
namespace Acme\DemoBundle\Validator\Constraints;

use Symfony\Component\Validator\Constraint;
use Symfony\Component\Validator\ConstraintValidator;

class ContainsAlphanumericValidator extends ConstraintValidator
{
    public function validate($value, Constraint $constraint)
    {
        if (!preg_match('/^[a-zA-Za0-9]+$/', $value, $matches)) {
            $this->context->addViolation($constraint->message, array('%string%' => $value));
        }
    }
}
```

Nota: O método `validate` não retorna um valor, em vez disso, ele acrescenta violações à propriedade `context` do validador com uma chamada do método `addViolation` se existem falhas de validação. Portanto, um valor pode ser considerado como sendo válido, desde que não cause violações adicionadas ao contexto. O primeiro parâmetro da chamada `addViolation` é a mensagem de erro para usar para aquela violação.

Novo na versão 2.1: O método `isValid` foi renomeado para `validate` no Symfony 2.1. O método `setMessage` também ficou obsoleto, em favor da chamada `addViolation` do contexto.

Usando o novo Validador

Usar validadores personalizados é muito fácil, assim como os fornecidos pelo Symfony2 em si:

- *YAML*

```
# src/Acme/BlogBundle/Resources/config/validation.yml
Acme\DemoBundle\Entity\AcmeEntity:
    properties:
        name:
            - NotBlank: ~
            - Acme\DemoBundle\Validator\Constraints\ContainsAlphanumeric: ~
```

- *Annotations*

```
// src/Acme/DemoBundle/Entity/AcmeEntity.php
use Symfony\Component\Validator\Constraints as Assert;
use Acme\DemoBundle\Validator\Constraints as AcmeAssert;

class AcmeEntity
{
    // ...

    /**
     * @Assert\NotBlank
     * @AcmeAssert\ContainsAlphanumeric
     */
    protected $name;

    // ...
}
```

- *XML*

```
<!-- src/Acme/DemoBundle/Resources/config/validation.xml -->
<?xml version="1.0" encoding="UTF-8" ?>
<constraint-mapping xmlns="http://symfony.com/schema/dic/constraint-mapping"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/dic/constraint-mapping http://symfony.com/sche

    <class name="Acme\DemoBundle\Entity\AcmeEntity">
        <property name="name">
            <constraint name="NotBlank" />
            <constraint name="Acme\DemoBundle\Validator\Constraints\ContainsAlphanumeric" />
        </property>
    </class>
</constraint-mapping>
```

- *PHP*

```
// src/Acme/DemoBundle/Entity/AcmeEntity.php
use Symfony\Component\Validator\Mapping\ClassMetadata;
use Symfony\Component\Validator\Constraints\NotBlank;
use Acme\DemoBundle\Validator\Constraints\ContainsAlphanumeric;

class AcmeEntity
{
    public $name;

    public static function loadValidatorMetadata(ClassMetadata $metadata)
    {
        $metadata->addPropertyConstraint('name', new NotBlank());
        $metadata->addPropertyConstraint('name', new ContainsAlphanumeric());
    }
}
```

Se a sua constraint contém opções, então elas devem ser propriedades públicas na classe Constraint personalizada que você criou anteriormente. Essas opções podem ser configuradas como opções nas constraints do núcleo do Symfony.

Validadores de Constraints com Dependências

Se o seu validador de restrição possui dependências, como uma conexão de banco de dados, ela terá que ser configurada como um serviço no container de injeção de dependência. Este serviço deve incluir a tag `validator.constraint_validator` e um atributo `alias`:

- *YAML*

```
services:
    validator.unique.your_validator_name:
        class: Fully\Qualified\Validator\Class\Name
        tags:
            - { name: validator.constraint_validator, alias: alias_name }
```

- *XML*

```
<service id="validator.unique.your_validator_name" class="Fully\Qualified\Validator\Class\Name">
    <argument type="service" id="doctrine.orm.default_entity_manager" />
    <tag name="validator.constraint_validator" alias="alias_name" />
</service>
```

- *PHP*

```
    $container
        ->register('validator.unique.your_validator_name', 'Fully\Qualified\Validator\Class\Name')
        ->addTag('validator.constraint_validator', array('alias' => 'alias_name'));
```

Sua classe de constraint pode agora usar este alias para referenciar o validador apropriado:

```
public function validatedBy()
{
    return 'alias_name';
}
```

Como mencionado acima, o Symfony2 irá procurar automaticamente por uma classe chamada após a constraint, com `Validator` acrescentado. Se o seu validador constraint está definido como um serviço, é importante que você sobrescreva o método `validatedBy()` para retornar o alias utilizado na definição de seu serviço, caso contrário, o Symfony2 não vai usar o serviço do validador de constraint, e, em vez disso, irá instanciar a classe, sem quaisquer dependências injetadas.

Classe Constraint Validadora

Junto da validação de uma propriedade de classe, uma constraint pode ter um escopo de classe, fornecendo um alvo:

```
public function getTargets()
{
    return self::CLASS_CONSTRAINT;
}
```

Com isso, o método validador `validate()` obtém um objeto como seu primeiro argumento:

```
class ProtocolClassValidator extends ConstraintValidator
{
    public function validate($protocol, Constraint $constraint)
    {
        if ($protocol->getFoo() != $protocol->getBar()) {
            $this->context->addViolationAtSubPath('foo', $constraint->message, array(), null);
        }
    }
}
```

Note que a classe constraint validadora é aplicada na classe em si, e não à propriedade:

- *YAML*

```
    # src/Acme/BlogBundle/Resources/config/validation.yml
    Acme\DemoBundle\Entity\AcmeEntity:
        constraints:
            - ContainsAlphanumeric
```

- *Annotations*

```
    /**
     * @AcmeAssert\ContainsAlphanumeric
     */
    class AcmeEntity
    {
        // ...
    }
```

- *XML*

```
    <!-- src/Acme/BlogBundle/Resources/config/validation.xml -->
    <class name="Acme\DemoBundle\Entity\AcmeEntity">
        <constraint name="ContainsAlphanumeric" />
    </class>
```

3.1.8 Configuração

Como Dominar e Criar novos Ambientes

Cada aplicação é a combinação de código e um conjunto de configurações que dita como o código deve funcionar. A configuração pode: definir o banco de dados a ser utilizado, se algo deve ou não ser armazenado em cache, ou como deve ser a verbosidade do log. No Symfony2, a idéia de "ambientes" é a idéia de que a mesma base de código pode ser executada usando várias configurações diferentes. Por exemplo, o ambiente `dev` deve usar uma configuração que faça com que o desenvolvimento seja fácil e amigável, enquanto o ambiente `prod` deve usar um conjunto de configurações otimizada para a velocidade.

Ambientes Diferentes, Diferentes Arquivos de Configuração

Uma aplicação típica do Symfony2 começa com três ambientes: `dev`, `prod` e `test`. Como discutido, cada "ambiente" simplesmente representa uma forma de executar o mesmo código com configurações diferentes. Não deve ser nenhuma surpresa, então, que cada ambiente carrega seu arquivo de configuração individual. Se você estiver usando o formato de configuração YAML, os seguintes arquivos são usados:

- para o ambiente `dev`: `app/config/config_dev.yml`
- para o ambiente `prod`: `app/config/config_prod.yml`
- para o ambiente `test`: `app/config/config_test.yml`

Isso funciona por meio de uma convenção simples que é usada, por padrão, dentro da classe `AppKernel`:

```
// app/AppKernel.php

// ...

class AppKernel extends Kernel
{
    // ...

    public function registerContainerConfiguration(LoaderInterface $loader)
    {
        $loader->load(__DIR__.'/config/config_'.$this->getEnvironment().'.yml');
    }
}
```

Como você pode ver, quando o Symfony2 é carregado, ele usa um dado ambiente para determinar qual arquivo de configuração deve carregar. Isto alcança o objetivo de vários ambientes de uma forma elegante, poderosa e transparente.

É claro que, na realidade, cada ambiente só difere um pouco dos outros. Geralmente, todos os ambientes irão compartilhar uma grande base de configuração comum. Abrindo o arquivo de configuração "dev", você pode ver como isso é feito fácil e transparentemente:

- *YAML*

```
    imports:
        - { resource: config.yml }
    # ...
```

- *XML*

```
<imports>
    <import resource="config.xml" />
</imports>
<!-- ... -->
```

- *PHP*

```
$loader->import('config.php');
// ...
```

Para compartilhar as configurações comuns, cada arquivo de configuração de ambiente simplesmente primeiro importa de um arquivo de configuração central (`config.yml`). O restante do arquivo pode então desviar-se da configuração padrão sobrescrevendo os parâmetros individuais. Por exemplo, por padrão, a barra de ferramentas `web_profiler` está desativada. No entanto, no ambiente `dev`, a barra de ferramentas é ativada, modificando o valor padrão no arquivo de configuração `dev`:

- *YAML*

```
# app/config/config_dev.yml
imports:
    - { resource: config.yml }

web_profiler:
    toolbar: true
    # ...
```

- *XML*

```
<!-- app/config/config_dev.xml -->
<imports>
    <import resource="config.xml" />
</imports>

<webprofiler:config
    toolbar="true"
    ... />
```

- *PHP*

```
// app/config/config_dev.php
$loader->import('config.php');

$container->loadFromExtension('web_profiler', array(
    'toolbar' => true,
    ...,
));
```

A execução de uma Aplicação em Ambientes Diferentes

Para executar a aplicação em cada ambiente, carregue a aplicação usando o front controller `app.php` (para o ambiente `prod`) ou o `app_dev.php` (para o ambiente `dev`):

```
http://localhost/app.php      -> *prod* environment
http://localhost/app_dev.php  -> *dev* environment
```

Nota: As URLs fornecidas assumem que o seu servidor web está configurado para usar o diretório `web/` da aplicação como sua raiz. Leia mais em Installing Symfony2.

Se você abrir um desses arquivos, verá rapidamente que o ambiente utilizado por cada um é definido explicitamente:

```
1 <?php
2
3 require_once __DIR__.'/../app/bootstrap_cache.php';
4 require_once __DIR__.'/../app/AppCache.php';
5
6 use Symfony\Component\HttpFoundation\Request;
7
8 $kernel = new AppCache(new AppKernel('prod', false));
9 $kernel->handle(Request::createFromGlobals())->send();
```

Como você pode ver, a chave `prod` especifica que esse ambiente será executado no ambiente `prod`. Uma aplicação Symfony2 pode ser executada em qualquer ambiente usando este código e alterando a string de ambiente.

Nota: O ambiente `test` é utilizado quando escrevemos testes funcionais e não é acessível diretamente no navegador através de um front controller. Em outras palavras, ao contrário dos outros ambientes, não há um arquivo front controller `app_test.php`.

Modo *Depuração*

Importante, mas sem relação com o tema de *ambientes* é a chave `false` na linha 8 do front controller acima. Esta especifica se a aplicação deve ou não ser executada em "modo de depuração". Independentemente do ambiente, uma aplicação Symfony2 pode ser executada com o modo de depuração definido como `true` ou `false`. Isso afeta muitas coisas na aplicação, como se os erros devem ou não ser apresentados ou se os arquivos de cache são dinamicamente reconstruídos em cada pedido. Apesar de não ser uma exigência, o modo de depuração é geralmente definido para `true` nos ambientes `dev` e `test` e `false` para o ambiente `prod`. Internamente, o valor do modo de depuração torna-se o parâmetro `kernel.debug` utilizado dentro do service container. Se você olhar dentro do arquivo de configuração da aplicação, verá o parâmetro utilizado, por exemplo, para ligar ou desligar o log ao usar o Doctrine DBAL:

- *YAML*

```
doctrine:
    dbal:
        logging:  "%kernel.debug%"
        # ...
```

- *XML*

```
<doctrine:dbal logging="%kernel.debug%" ... />
```

- *PHP*

```
$container->loadFromExtension('doctrine', array(
    'dbal' => array(
        'logging'  => '%kernel.debug%',
        ...,
    ),
    ...
));
```

Criação de um Novo Ambiente

Por padrão, uma aplicação Symfony2 tem que lidar com três ambientes na maioria dos casos. Claro, desde que um ambiente nada mais é do que uma string que corresponde a um conjunto de configuração, a criação de um novo ambiente é bastante fácil.

Suponha, por exemplo, que antes da implantação, você precisa realizar um benchmark da sua aplicação. Uma forma de realizar o benchmark da aplicação é usar configurações próximas da produção, mas com o `web_profiler` do Symfony2 habilitado. Isto permite ao Symfony2 registrar as informações sobre sua aplicação enquanto o realiza o benchmark.

A melhor forma de realizar isto é através de um novo ambiente chamado, por exemplo, `benchmark`. Comece por criar um novo arquivo de configuração:

- *YAML*

```
# app/config/config_benchmark.yml
imports:
    - { resource: config_prod.yml }

framework:
    profiler: { only_exceptions: false }
```

- *XML*

```
<!-- app/config/config_benchmark.xml -->
<imports>
    <import resource="config_prod.xml" />
</imports>

<framework:config>
    <framework:profiler only-exceptions="false" />
</framework:config>
```

- *PHP*

```
// app/config/config_benchmark.php
$loader->import('config_prod.php')

$container->loadFromExtension('framework', array(
    'profiler' => array('only-exceptions' => false),
));
```

E com esta simples adição, a aplicação agora suporta um novo ambiente chamado `benchmark`.

Este novo arquivo de configuração importa a configuração do ambiente `prod` e modifica ela. Isso garante que o novo ambiente é idêntico ao ao ambiente `prod`, exceto por quaisquer alterações explicitamente feitas aqui.

Já que você deseja que este ambiente seja acessível através de um navegador, você também deve criar um front controller para ele. Copie o arquivo `web/app.php` para `web/app_benchmark.php` e edite o ambiente para `benchmark`:

```
<?php

require_once __DIR__.'/../app/bootstrap.php';
require_once __DIR__.'/../app/AppKernel.php';

use Symfony\Component\HttpFoundation\Request;
```

```
$kernel = new AppKernel('benchmark', false);
$kernel->handle(Request::createFromGlobals())->send();
```

O novo ambiente está agora acessível através:

```
http://localhost/app_benchmark.php
```

Nota: Alguns ambientes, como o ambiente `dev`, nunca são destinados ao acesso em qualquer servidor implantado para o público em geral. Isto é porque certos ambientes, para fins de depuração, podem fornecer muita informação sobre a aplicação ou a infra-estrutura subjacente. Para ter certeza que estes ambientes não são acessíveis, o front controller é normalmente protegido de endereços IP externos através do seguinte código no topo do controlador:

```
    if (!in_array(@$_SERVER['REMOTE_ADDR'], array('127.0.0.1', '::1'))) {
        die('You are not allowed to access this file. Check '.basename(__FILE__).' for more informat
    }
```

Ambientes e o Diretório de Cache

O Symfony2 aproveita o cache de muitas maneiras: para a configuração da aplicação, configuração de roteamento, templates Twig e mais é realizado o cache para objetos PHP armazenados em arquivos no sistema de arquivos.

Por padrão, esses arquivos em cache são armazenados, em grande parte, no diretório `app/cache`. No entanto, cada ambiente armazena seu próprio conjunto de arquivos:

```
app/cache/dev   - diretório cache para o ambiente *dev*
app/cache/prod  - diretório cache para o ambiente *prod*
```

Às vezes, quando depurando, pode ser útil inspecionar um arquivo em cache para compreender como algo está funcionando. Ao fazê-lo, lembre-se de olhar no diretório do ambiente que você está usando (mais comumente `dev` enquanto estiver desenvolvendo e depurando). Embora possa variar, o diretório `app/cache/dev` inclui o seguinte:

- `appDevDebugProjectContainer.php` - o "container de serviço" em cache que representa a configuração da aplicação em cache;
- `appdevUrlGenerator.php` - a classe PHP gerada a partir da configuração de roteamento e usada ao gerar URLs;
- `appdevUrlMatcher.php` - a classe PHP usada para correspondência de rota - olhe aqui para ver a lógica de expressões regulares compiladas usadas para correspondência das URLs de entrada para diferentes rotas;
- `twig/` - Este diretório contém todos templates Twig em cache.

Nota: Você pode facilmente mudar a localização do diretório e o seu nome. Para mais informações leia o artigo Como Substituir a Estrutura de Diretório Padrão do Symfony.

Investigando mais Detalhadamente

Leia o artigo em Como definir Parâmetros Externos no Container de Serviços.

Como Substituir a Estrutura de Diretório Padrão do Symfony

O Symfony vem automaticamente com uma estrutura de diretórios padrão. Você pode facilmente substituir essa estrutura de diretório criando a seu própria. A estrutura de diretório padrão é:

```
app/
    cache/
    config/
    logs/
    ...
src/
    ...
vendor/
    ...
web/
    app.php
    ...
```

Substituir o diretório `cache`

Você pode substituir o diretório cache, sobrescrevendo o método `getCacheDir` na classe `AppKernel` de sua aplicação:

```
// app/AppKernel.php

// ...
class AppKernel extends Kernel
{
    // ...

    public function getCacheDir()
    {
        return $this->rootDir.'/'.$this->environment.'/cache/';
    }
}
```

`$this->rootDir` é o caminho absoluto para o diretório `app` e `$this->environment` é o ambiente atual (ou seja, `dev`). Neste caso você alterou a localização do diretório cache para `app/{environment}/cache`.

Cuidado: Você deve manter o diretório `cache` diferente para cada ambiente, caso contrário, algum comportamento inesperado pode acontecer. Cada ambiente gera seus próprios arquivos de configuração em cache, e assim, cada um precisa de seu próprio diretório para armazenar os arquivos de cache.

Substituir o diretório `logs`

O processo para substituir o diretório `logs` é o mesmo do diretório `cache`, a única diferença é que você precisa sobrescrever o método `getLogDir`:

```
// app/AppKernel.php

// ...
class AppKernel extends Kernel
{
    // ...
```

```
    public function getLogDir()
    {
        return $this->rootDir.'/'.$this->environment.'/logs/';
    }
}
```

Aqui você alterou o local do diretório para `app/{environment}/logs`.

Substituir o diretório `web`

Se você precisa renomear ou mover o seu diretório `web`, a única coisa que você precisa garantir é que o caminho para o diretório `app` ainda está correto em seus front controllers `app.php` e `app_dev.php`. Se você simplesmente renomear o diretório, então está tudo ok. Mas se você moveu de alguma forma, pode precisar modificar os caminhos dentro desses arquivos:

```
require_once __DIR__.'/../Symfony/app/bootstrap.php.cache';
require_once __DIR__.'/../Symfony/app/AppKernel.php';
```

Dica: Alguns hosts compartilhados tem um diretório raiz web `public_html`. Renomeando seu diretório web de `web` para `public_html` é uma maneira de fazer funcionar o seu projeto Symfony em seu servidor compartilhado. Outra forma é implantar sua aplicação em um diretório fora do raiz web, excluir seu diretório `public_html` e, então, substituí-lo por um link simbólico para o `web` em seu projeto.

Nota: Se você utiliza o AsseticBundle precisará configurar o seguinte, para que ele possa usar o diretório `web` correto:

```
# app/config/config.yml

# ...
assetic:
    # ...
    read_from: "%kernel.root_dir%/../../public_html"
```

Agora você só precisa realizar o dump dos assets novamente e sua aplicação deve funcionar:

```
$ php app/console assetic:dump --env=prod --no-debug
```

Como definir Parâmetros Externos no Container de Serviços

No capítulo Como Dominar e Criar novos Ambientes, você aprendeu como gerenciar a configuração da sua aplicação. Às vezes, armazenar certas credenciais fora do código do seu projeto pode beneficiar a sua aplicação. A configuração do banco de dados é um exemplo. A flexibilidade do container de serviço do Symfony permite à você fazer isso facilmente.

Variáveis de Ambiente

O Symfony vai pegar qualquer variável de ambiente com o prefixo `SYMFONY__` e setá-la como um parâmetro no container de serviço. Sublinhados duplos são substituídos por um ponto, pois o ponto não é um caracter válido no nome de uma variável de ambiente.

Por exemplo, se você está usando o Apache, as variáveis de ambiente podem ser definidas utilizando a seguinte configuração `VirtualHost`:

```
<VirtualHost *:80>
    ServerName       Symfony2
    DocumentRoot     "/path/to/symfony_2_app/web"
    DirectoryIndex   index.php index.html
    SetEnv           SYMFONY__DATABASE__USER user
    SetEnv           SYMFONY__DATABASE__PASSWORD secret

    <Directory "/path/to/symfony_2_app/web">
        AllowOverride All
        Allow from All
    </Directory>
</VirtualHost>
```

Nota: O exemplo acima é para uma configuração Apache, usando a diretiva SetEnv. No entanto, isso vai funcionar para qualquer servidor web que suporte a definição de variáveis de ambiente.

Além disso, para que o seu console funcione (que não usa Apache), você deve exportar estas como variáveis shell. Em um sistema Unix, você pode executar o seguinte:

```
$ export SYMFONY__DATABASE__USER=user
$ export SYMFONY__DATABASE__PASSWORD=secret
```

Agora que você declarou uma variável de ambiente, ela estará presente na variável global `$_SERVER` do PHP. O Symfony, então, automaticamente define todas as variáveis ‘‘$_SERVER‘‘ prefixadas com `SYMFONY__` como parâmetros no container de serviços.

Agora, você pode referenciar estes parâmetros em qualquer local onde precisar deles.

- *YAML*

```
doctrine:
    dbal:
        driver    pdo_mysql
        dbname:   symfony2_project
        user:     "%database.user%"
        password: "%database.password%"
```

- *XML*

```
<!-- xmlns:doctrine="http://symfony.com/schema/dic/doctrine" -->
<!-- xsi:schemaLocation="http://symfony.com/schema/dic/doctrine http://symfony.com/schema/dic/do

<doctrine:config>
    <doctrine:dbal
        driver="pdo_mysql"
        dbname="symfony2_project"
        user="%database.user%"
        password="%database.password%"
    />
</doctrine:config>
```

- *PHP*

```
$container->loadFromExtension('doctrine', array('dbal' => array(
    'driver'   => 'pdo_mysql',
    'dbname'   => 'symfony2_project',
    'user'     => '%database.user%',
```

```
        'password' => '%database.password%',
    ));
```

Constantes

O container também possui suporte para definir constantes do PHP como parâmetros. Para aproveitar esse recurso, mapeie o nome da sua constante para uma chave de parâmetro , e defina o tipo como `constant`.

```
    <?xml version="1.0" encoding="UTF-8"?>

    <container xmlns="http://symfony.com/schema/dic/services"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

        <parameters>
            <parameter key="global.constant.value" type="constant">GLOBAL_CONSTANT</parameter>
            <parameter key="my_class.constant.value" type="constant">My_Class::CONSTANT_NAME</parame
        </parameters>
    </container>
```

Nota: Isso funciona somente para a configuração XML. Se você *não* está usando XML, simplesmente importe um arquivo XML para aproveitar essa funcionalidade:

```
# app/config/config.yml
imports:
    - { resource: parameters.xml }
```

Configurações Diversas

A diretiva `imports` pode ser usada para puxar os parâmetros armazenados em outro lugar. Importando um arquivo PHP lhe dá a flexibilidade para adicionar o que for necessário no container. O seguinte importa um arquivo chamado `parameters.php`.

- *YAML*

```
    # app/config/config.yml
    imports:
        - { resource: parameters.php }
```

- *XML*

```
    <!-- app/config/config.xml -->
    <imports>
        <import resource="parameters.php" />
    </imports>
```

- *PHP*

```
    // app/config/config.php
    $loader->import('parameters.php');
```

Nota: Um arquivo de recursos pode ser um de muitos tipos. PHP, XML, YAML, INI e recursos de closure são todos suportados pela directiva `imports`.

No `parameters.php`, diga ao container de serviço os parâmetros que você deseja definir. Isto é útil quando alguma configuração importante está em um formato fora do padrão. O exemplo abaixo inclui a configuração de um banco de dados do Drupal no container de serviço do Symfony.

```
// app/config/parameters.php
include_once('/path/to/drupal/sites/default/settings.php');
$container->setParameter('drupal.database.url', $db_url);
```

Como usar o PdoSessionStorage para armazenar as Sessões no Banco de Dados

O armazenamento de sessão padrão do Symfony2 grava as informações da sessão em arquivo(s). A maioria dos sites de médio à grande porte utilizam um banco de dados para armazenar os valores da sessão, em vez de arquivos, pois os bancos de dados são mais fáceis de usar e escalam em um ambiente multi-servidor.

O Symfony2 tem uma solução interna para o armazenamento de sessão em banco de dados chamado `PdoSessionHandler`. Para usá-lo, você só precisa alterar alguns parâmetros no `config.yml` (ou o formato de configuração de sua escolha):

Novo na versão 2.1: No Symfony2.1 a classe e o namespace foram ligeiramente modificados. Você pode agora encontrar as classes de armazenamento de sessão no namespace *Session\Storage*: `Symfony\Component\HttpFoundation\Session\Storage`. Observe também que no Symfony2.1 você deve configurar o `handler_id` e não o `storage_id` como no Symfony2.0. Abaixo, você vai perceber que o `%session.storage.options%` não é mais usado.

- *YAML*

```
# app/config/config.yml
framework:
    session:
        # ...
        handler_id:     session.handler.pdo

parameters:
    pdo.db_options:
        db_table:    session
        db_id_col:   session_id
        db_data_col: session_value
        db_time_col: session_time

services:
    pdo:
        class: PDO
        arguments:
            dsn:      "mysql:dbname=mydatabase"
            user:     myuser
            password: mypassword

    session.handler.pdo:
        class:     Symfony\Component\HttpFoundation\Session\Storage\Handler\PdoSessionHandler
        arguments: [@pdo, %pdo.db_options%]
```

- *XML*

```
<!-- app/config/config.xml -->
<framework:config>
    <framework:session handler-id="session.handler.pdo" lifetime="3600" auto-start="true"/>
</framework:config>
```

```
    <parameters>
        <parameter key="pdo.db_options" type="collection">
            <parameter key="db_table">session</parameter>
            <parameter key="db_id_col">session_id</parameter>
            <parameter key="db_data_col">session_value</parameter>
            <parameter key="db_time_col">session_time</parameter>
        </parameter>
    </parameters>

    <services>
        <service id="pdo" class="PDO">
            <argument>mysql:dbname=mydatabase</argument>
            <argument>myuser</argument>
            <argument>mypassword</argument>
        </service>

        <service id="session.handler.pdo" class="Symfony\Component\HttpFoundation\Session\Storage\Ha
            <argument type="service" id="pdo" />
            <argument>%pdo.db_options%</argument>
        </service>
    </services>
```

- *PHP*

```
    // app/config/config.php
    use Symfony\Component\DependencyInjection\Definition;
    use Symfony\Component\DependencyInjection\Reference;

    $container->loadFromExtension('framework', array(
        // ...
        'session' => array(
            ...,
            'handler_id' => 'session.handler.pdo',
        ),
    ));

    $container->setParameter('pdo.db_options', array(
        'db_table'      => 'session',
        'db_id_col'     => 'session_id',
        'db_data_col'   => 'session_value',
        'db_time_col'   => 'session_time',
    ));

    $pdoDefinition = new Definition('PDO', array(
        'mysql:dbname=mydatabase',
        'myuser',
        'mypassword',
    ));
    $container->setDefinition('pdo', $pdoDefinition);

    $storageDefinition = new Definition('Symfony\Component\HttpFoundation\Session\Storage\Handler\Pd
        new Reference('pdo'),
        '%pdo.db_options%',
    ));
    $container->setDefinition('session.handler.pdo', $storageDefinition);
```

- `db_table`: O nome da tabela de sessão no seu banco de dados
- `db_id_col`: O nome da coluna id na sua tabela de sessão (VARCHAR (255) ou maior)

- `db_data_col`: O nome da coluna value na sua tabela de sessão (TEXT ou CLOB)
- `db_time_col`: O nome da coluna time em sua tabela de sessão (INTEGER)

Compartilhando suas Informações de Conexão do Banco de Dados

Com a configuração fornecida, as configurações de conexão do banco de dados são definidas somente para a conexão de armazenamento de sessão. Isto está OK quando você usa um banco de dados separado para os dados da sessão.

Mas, se você gostaria de armazenar os dados da sessão no mesmo banco de dados que o resto dos dados do seu projeto, você pode usar as definições de conexão do parameter.ini referenciando os parâmetros relacionados ao banco de dados definidos lá:

- *YAML*

```
pdo:
    class: PDO
    arguments:
        - "mysql:dbname=%database_name%"
        - %database_user%
        - %database_password%
```

- *XML*

```
<service id="pdo" class="PDO">
    <argument>mysql:dbname=%database_name%</argument>
    <argument>%database_user%</argument>
    <argument>%database_password%</argument>
</service>
```

- *PHP*

```
$pdoDefinition = new Definition('PDO', array(
    'mysql:dbname=%database_name%',
    '%database_user%',
    '%database_password%',
));
```

Exemplo de Instruções SQL

MySQL A instrução SQL para criar a tabela de banco de dados necessária pode ser semelhante a seguinte (MySQL):

```
CREATE TABLE `session` (
    `session_id` varchar(255) NOT NULL,
    `session_value` text NOT NULL,
    `session_time` int(11) NOT NULL,
    PRIMARY KEY (`session_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

PostgreSQL Para o PostgreSQL, a declaração deve ficar assim:

```
CREATE TABLE session (
    session_id character varying(255) NOT NULL,
    session_value text NOT NULL,
    session_time integer NOT NULL,
    CONSTRAINT session_pkey PRIMARY KEY (session_id)
);
```

Como usar o Apache Router

O Symfony2, mesmo sendo rápido logo após a sua instalação, ainda fornece várias formas de aumentar essa velocidade com poucos ajustes. Uma dessas formas é deixar o apache lidar com as rotas diretamente, em vez de usar o Symfony2 para esta tarefa.

Alterando os Parâmetros de Configuração do Router

Para fazer o dump das rotas do Apache precisamos primeiro ajustar alguns parâmetros de configuração e dizer ao Symfony2 para usar o `ApacheUrlMatcher` em vez do padrão:

```
# app/config/config_prod.yml
parameters:
    router.options.matcher.cache_class: ~ # disable router cache
    router.options.matcher_class: Symfony\Component\Routing\Matcher\ApacheUrlMatcher
```

Dica: Note que o `ApacheUrlMatcher` estende `UrlMatcher` assim, mesmo se você não regerar as regras url_rewrite, tudo vai funcionar (porque no final do `ApacheUrlMatcher::match()` é realizada uma chamada para o `parent::match()`).

Gerando regras do mod_rewrite

Para testar se está funcionando, vamos criar uma rota bem básica para o bundle demo:

```
# app/config/routing.yml
hello:
    pattern:  /hello/{name}
    defaults: { _controller: AcmeDemoBundle:Demo:hello }
```

Agora vamos gerar regras **url_rewrite**:

```
$ php app/console router:dump-apache -e=prod --no-debug
```

Que deve produzir aproximadamente o seguinte:

```
# skip "real" requests
RewriteCond %{REQUEST_FILENAME} -f
RewriteRule .* - [QSA,L]

# hello
RewriteCond %{REQUEST_URI} ^/hello/([^/]+?)$
RewriteRule .* app.php [QSA,L,E=_ROUTING__route:hello,E=_ROUTING_name:%1,E=_ROUTING__controller:AcmeI
```

Agora você pode reescrever o *web/.htaccess* para usar as novas regras, portanto, com o nosso exemplo, ele deve parecer com o seguinte:

```
<IfModule mod_rewrite.c>
    RewriteEngine On

    # skip "real" requests
    RewriteCond %{REQUEST_FILENAME} -f
    RewriteRule .* - [QSA,L]

    # hello
```

```
    RewriteCond %{REQUEST_URI} ^/hello/([^/]+?)$
    RewriteRule .* app.php [QSA,L,E=_ROUTING__route:hello,E=_ROUTING_name:%1,E=_ROUTING__controller:A
</IfModule>
```

Nota: O procedimento acima deve ser feito cada vez que você adicionar/alterar uma rota se deseja aproveitar o máximo desta configuração.

É isso! Você agora está pronto para usar as regras do Apache Route.

Ajustes adicionais

Para economizar um pouco de tempo de processamento, altere as ocorrências de `Request` para `ApacheRequest` no `web/app.php`:

```
// web/app.php

require_once __DIR__.'/../app/bootstrap.php.cache';
require_once __DIR__.'/../app/AppKernel.php';
//require_once __DIR__.'/../app/AppCache.php';

use Symfony\Component\HttpFoundation\ApacheRequest;

$kernel = new AppKernel('prod', false);
$kernel->loadClassCache();
//$kernel = new AppCache($kernel);
$kernel->handle(ApacheRequest::createFromGlobals())->send();
```

3.1.9 Bundles

Como usar Melhores Práticas para a Estruturação dos Bundles

Um Bundle é um diretório que tem uma estrutura bem definida e pode hospedar qualquer coisa desde classes até controladores e recursos web. Mesmo os bundles sendo muito flexíveis, você deve seguir algumas das melhores práticas se deseja distribuí-los.

Nome do Bundle

Um bundle é também um namespace PHP. O namespace deve seguir os padrões técnicos de interoperabilidade para namespaces do PHP 5.3 e nomes de classes: ele inicia com o segmento do vendor, seguido por zero ou mais segmentos de categoria, e termina com o nome curto do namespace, que deve terminar com o sufixo `Bundle`.

Um namespace torna-se um bundle assim que você adiciona uma classe bundle à ele. O nome da classe do bundle deve seguir estas regras simples:

- Usar apenas caracteres alfanuméricos e sublinhados;
- Usar o nome em CamelCase;
- Usar um nome descritivo e curto (não mais que 2 palavras);
- Prefixar o nome com a concatenação do vendor (e, opcionalmente, os namespaces da categoria);
- Adicionar o sufixo `Bundle` ao nome.

Aqui estão alguns namespaces de bundle e nomes de classes válidos:

Namespace	Nome da Classe do Bundle
`Acme\Bundle\BlogBundle`	`AcmeBlogBundle`
`Acme\Bundle\Social\BlogBundle`	`AcmeSocialBlogBundle`
`Acme\BlogBundle`	`AcmeBlogBundle`

Por convenção, o método `getName()` da classe bundle deve retornar o nome da classe.

Nota: Se você compartilhar publicamente seu bundle, você deve usar o nome da classe do bundle como o nome do repositório (`AcmeBlogBundle` e não `BlogBundle` por exemplo).

Nota: Os Bundles do núcleo do Symfony2 não prefixam a classe Bundle com `Symfony` e sempre adicionam um subnamespace `Bundle`, por exemplo: `FrameworkBundle`.

Cada bundle tem um alias, que é a versão curta e em letras minúsculas do nome do bundle usando sublinhados (por exemplo, `acme_hello` para `AcmeHelloBundle`, ou `acme_social_blog` para `Acme\Social\BlogBundle`). Estes alias são usados para definir exclusividade dentro de um bundle (veja abaixo alguns exemplos de uso).

Estrutura de Diretórios

A estrutura básica de diretório de um bundle `HelloBundle` deve parecer com a seguinte:

```
XXX/...
    HelloBundle/
        HelloBundle.php
        Controller/
        Resources/
            meta/
                LICENSE
            config/
            doc/
                index.rst
            translations/
            views/
            public/
        Tests/
```

O(s) diretório(s) `XXX` refletem a estrutura do namespace do bundle.

Os seguintes arquivos são obrigatórios:

- `HelloBundle.php`;
- `Resources/meta/LICENSE`: A licença completa para o código;
- `Resources/doc/index.rst`: O arquivo raiz para a documentação do Bundle.

Nota: Estas convenções garantem que ferramentas automatizadas possam contar com esta estrutura padrão para trabalhar.

A profundidade dos sub-diretórios deve ser mantida ao mínimo para as classes e arquivos mais utilizados (2 níveis, no máximo). Mais níveis podem ser definidos para arquivos não-estratégicos ou menos utilizados.

O diretório bundle é somente leitura. Se você precisa gravar arquivos temporários, armazene-os sob o diretório `cache/` ou `log/` da aplicação host. Ferramentas podem gerar arquivos na estrutura de diretório do bundle, mas apenas se os arquivos gerados serão parte do repositório.

As classes e arquivos seguintes possuem um local específico:

Tipo	Diretório
Comandos	`Command/`
Controladores	`Controller/`
Extensões do Service Container	`DependencyInjection/`
Listeners de Evento	`EventListener/`
Configuração	`Resources/config/`
Recursos Web	`Resources/public/`
Arquivos de Tradução	`Resources/translations/`
Templates	`Resources/views/`
Testes Unitários e Funcionais	`Tests/`

Classes

A estrutura de diretórios do bundle é usada como hierarquia de namespace. Por exemplo, um controlador `HelloController` é armazenado em `Bundle/HelloBundle/Controller/HelloController.php` e o nome totalmente qualificado da classe é `Bundle\HelloBundle\Controller\HelloController`.

Todas as classes e arquivos devem seguir os padrões de codificação do Symfony2.

Algumas classes deve ser vistas como facades e devem ser o mais curtas possível, como Commands, Helpers, Listeners e Controllers.

As classes que se conectam ao Dispatcher de Eventos devem ter o sufixo `Listener`.

Classes de exceções (Exceptions) devem ser armazenadas em um sub-namespace `Exception`.

Vendors

Um bundle não deve incorporar bibliotecas PHP de terceiros. Em vez disso, ele deve contar com o autoloading padrão do Symfony2.

Um bundle não deve incorporar bibliotecas de terceiros escritas em JavaScript, CSS ou qualquer outra linguagem.

Testes

Um bundle deve vir com um conjunto de testes escritos com o PHPUnit e armazenados sob o diretório `Tests/`. Os testes devem seguir os seguintes princípios:

- O conjunto de testes deve ser executável com um simples comando `phpunit`, executado a partir de uma aplicação de exemplo;
- Os testes funcionais só devem ser usados para testar a resposta de saída e algumas informações de perfis, caso você tiver;
- A cobertura de código deve cobrir pelo menos 95% da base de código.

Nota: Um conjunto de testes não deve conter scripts `AllTests.php`, mas deve contar com a existência de um arquivo `phpunit.xml.dist`.

Documentação

Todas as classes e funções devem vir com PHPDoc completo.

Documentação extensa também deve ser fornecida no formato `reStructuredText`, sob o diretório `Resources/doc/`; o arquivo `Resources/doc/index.rst` é o único arquivo obrigatório e deve ser o ponto de entrada para a documentação.

Controladores

Como melhor prática, os controladores em um bundle que se destina a ser distribuído não deve estender a classe base `Controller`. Ao invés disso, eles podem implementar `ContainerAwareInterface` ou estender `ContainerAware`.

Nota: Se você verificar os métodos do `Controller`, vai ver que eles são apenas atalhos para facilitar a curva de aprendizado.

Roteamento

Se o bundle oferece rotas, elas devem ser prefixadas com o alias do bundle. Por exemplo, para o AcmeBlogBundle, todas as rotas devem ser prefixadas com `acme_blog_`.

Templates

Se um bundle fornece templates, eles devem usar o Twig. Um bundle não deve fornecer um layout principal, exceto se ele fornece uma aplicação completa.

Arquivos de Tradução

Se um bundle fornece mensagens de tradução, elas devem ser definidas no formato XLIFF; o domínio deve ser nomeado após o nome do bundle (`bundle.hello`).

Um bundle não deve sobrescrever as mensagens existentes de outro bundle.

Configuração

Para proporcionar maior flexibilidade, um bundle pode fornecer definições configuráveis usando os mecanismos embutidos do Symfony2.

Definições de configuração simples contam com a entrada padrão `parameters` da configuração do Symfony2. Os parâmetros do Symfony2 são simples pares chave/valor, um valor pode ser qualquer valor válido em PHP. Cada nome de parâmetro deve começar com o alias do bundle, embora esta seja apenas uma sugestão de melhor prática. O resto do nome do parâmetro usará um ponto (`.`) para separar partes diferentes (por exemplo, `acme_hello.email.from`).

O usuário final pode fornecer valores em qualquer arquivo de configuração:

- *YAML*

```
# app/config/config.yml
parameters:
    acme_hello.email.from: fabien@example.com
```

- *XML*

```
<!-- app/config/config.xml -->
<parameters>
    <parameter key="acme_hello.email.from">fabien@example.com</parameter>
</parameters>
```

- *PHP*

```
// app/config/config.php
$container->setParameter('acme_hello.email.from', 'fabien@example.com');
```

- *INI*

```
; app/config/config.ini
[parameters]
acme_hello.email.from = fabien@example.com
```

Recupere os parâmetros de configuração no seu código a partir do container:

```
$container->getParameter('acme_hello.email.from');
```

Mesmo esse mecanismo sendo bastante simples, você é altamente encorajado a usar a configuração semântica descrita no cookbook.

Nota: Se você estiver definindo serviços, eles também devem ser prefixados com o alias do bundle.

Aprenda mais no Cookbook

- Como expor uma Configuração Semântica para um Bundle

Como usar herança para substituir partes de um Bundle

Ao trabalhar com bundles de terceiros, você frequentemente precisará substituir um arquivo dele por um próprio seu para personalizar seu comportamento ou aparência. Symfony possui uma maneira bem conveniente de personalizar controllers, templates, Traduções e outros arquivos do diretório `Resources/` de um bundle.

Por exemplo, suponha que você está instalando FOSUserBundle, mas você quer que a template `layout.html.twig` o um dos seus controllers seja aqueles que você personalizou e colocou no seu bundle. No exemplo seguinte, estamos assumindo que você já tenha o bundle `AcmeUserBundle` e coloque os arquivos personalizados nele. O primeiro passo é registrar o bundle `FOSUserBundle` como pai do seu bundle:

```
// src/Acme/UserBundle/AcmeUserBundle.php
namespace Acme\UserBundle;

use Symfony\Component\HttpKernel\Bundle\Bundle;

class AcmeUserBundle extends Bundle
{
    public function getParent()
    {
        return 'FOSUserBundle';
    }
}
```

Esta simples alteração permitirá que substitua vários partes de `FOSUserBundle` simplesmente criando um arquivo com o mesmo nome.

Substituindo controladores

Suponha que você queira adicionar alguma funcionalidade a ação `registerAction` do controlador `RegistrationController` que está dentro do bundle `FOSUserBundle`. Para fazê-lo, basta criar o seu próprio `RegistrationController.php`, crie um método que substitua o do bundle original e mude sua funcionalidade como mostrado a seguir.

```
// src/Acme/UserBundle/Controller/RegistrationController.php
namespace Acme\UserBundle\Controller;

use FOS\UserBundle\Controller\RegistrationController as BaseController;

class RegistrationController extends BaseController
{
    public function registerAction()
    {
        $response = parent::registerAction();

        // do custom stuff

        return $response;
    }
}
```

Dica: Dependendo do tipo de personalização que precisa fazer no controlador, você pode substituir completamente o método com lógica própria sem nem mesmo chamar `parent::registerAction()`.

Nota: Substituir controladores desta maneira somente funciona se o bundle referencia o controlador utilizando sintaxe padrão `FOSUserBundle:Registration:register` nas rotas e nos templates. Esta é a sintaxe recomendada.

Substituindo recursos: Templates, Rotas, Traduções, Validação, etc

A maioria dos recursos também podem ser substituídos, simplesmente criando um arquivo no mesmo caminho relativo que estiver no bundle pai.

Por exemplo, é muito comum precisar substituir o arquivo de template `layout.html.twig` do bundle `FOSUserBundle` para utilizar o layout base de sua própria aplicação. Uma vez que o arquivo fica no caminho `Resources/views/layout.html.twig` dentro do bundle `FOSUserBundle` você consegue criar seu próprio arquivo no mesmo lugar relativo dentro do seu bundler (por exemplo, `Resources/views/layout.html.twig` do bundle `FOSUserBundle`). O Symfony vai ignorar o arquivo dentro do `FOSUserBundle` e utilizar o seu no lugar.

O mesmo vale para arquivos de rotas, configuração de Validação e outros recursos.

Nota: A substituição de recursos só funciona quando você se refere a recursos utilizando a sintaxe recomendada `@FosUserBundle/Resources/config/routing/security.xml`. Se você se referir a recursos sem o atalho @FosUserBundle, eles não serão substituídos.

Nota: Arquivos de traduções não funcionam da maneira descrita acima. Todos os ficheiros traduzidos serão adicionados em um conjunto de "pools" organizados por dominios. Symfony abrirá os ficheiros de tradução dos bundles primeiro na ordem que eles são inicializados e então do seu diretorio 'app/Resource'. Se houver dois ficheiros da mesma tradução estiver especificados por diferentes 'Resources', o ficheiro de tradução que for aberto por ultimo será o utilizado.

Como Sobrescrever qualquer parte de um Bundle

Este documento é uma referência rápida sobre como sobrescrever diferentes partes de bundles de terceiros.

Templates

Para obter informações sobre como sobrescrever templates, consulte * *Sobrepondo Templates de Pacote*. * Como usar herança para substituir partes de um Bundle

Roteamento

O roteamento nunca é automaticamente importado no Symfony2. Se você quiser incluir as rotas de qualquer bundle, elas devem ser manualmente importadas em algum lugar na sua aplicação (ex.: `app/config/routing.yml`).

A maneira mais fácil para "sobrescrever" o roteamento de um bundle é nunca importá-lo . Em vez de importar o roteamento de um bundle de terceiros, simplesmente copie o arquivo de roteamento em sua aplicação, modifique-o e importe-o no lugar.

Controladores

Assumindo que o bundle de terceiro envolvido usa controladores não-serviços (que é quase sempre o caso), você pode facilmente sobrescrever os controladores através de herança do bundle: Para mais informações, consulte Como usar herança para substituir partes de um Bundle.

Serviços e Configuração

Existem duas opções para sobrescrever/estender um serviço. Primeiro, você pode definir o parâmetro que contêm o nome do serviço da classe para a sua própria classe, definindo ele em `app/config/config.yml`. Isto, naturalmente, só é possível se o nome da classe está definido como um parâmetro na configuração de serviço do bundle que contém o serviço. Por exemplo, para sobrescrever a classe usada pelo serviço `translator` do Symfony, você poderia sobrescrever o parâmetro `translator.class`. Para saber exatamente qual parâmetro deve-se sobrescrever, poderá ser necessária alguma pesquisa. Para o tradutor, o parâmetro é definido e usado no arquivo `Resources/config/translation.xml` do FrameworkBundle:

- *YAML*

```
# app/config/config.yml
parameters:
    translator.class:      Acme\HelloBundle\Translation\Translator
```

- *XML*

```
<!-- app/config/config.xml -->
<parameters>
    <parameter key="translator.class">Acme\HelloBundle\Translation\Translator</parameter>
</parameters>
```

- *PHP*

```
// app/config/config.php
$container->setParameter('translator.class', 'Acme\HelloBundle\Translation\Translator');
```

Em segundo lugar, se a classe não está disponível como um parâmetro, você quer ter a certeza que a classe será sempre sobrescrita quando seu bundle for utilizado, ou quando você precisa modificar algo além do nome da classe, você deve usar um compiler pass:

```
// src/Acme/FooBundle/DependencyInjection/Compiler/OverrideServiceCompilerPass.php
namespace Acme\DemoBundle\DependencyInjection\Compiler;

use Symfony\Component\DependencyInjection\Compiler\CompilerPassInterface;
use Symfony\Component\DependencyInjection\ContainerBuilder;

class OverrideServiceCompilerPass implements CompilerPassInterface
{
    public function process(ContainerBuilder $container)
    {
        $definition = $container->getDefinition('original-service-id');
        $definition->setClass('Acme\DemoBundle\YourService');
    }
}
```

Neste exemplo, buscamos a definição de serviço do serviço original, e definimos seu nome de classe para a nossa própria classe.

Veja `/cookbook/service_container/compiler_passes` para obter informações sobre como usar compiler passes. Se você quer fazer algo além de apenas sobrescrever a classe - como adicionar uma chamada de método - você só pode usar o método compiler pass.

Entidades e Mapeamento de Entidade

Em andamento...

Formulários

A fim de sobrescrever um tipo de formulário (form type), ele tem que ser registrado como um serviço (o que significa que tem a tag definida como "form.type"). Você pode, então, sobrescrevê-lo como faria com qualquer serviço, como foi explicado em *Serviços e Configuração*. Isto, é claro, somente funcionará se o tipo é referido por seu alias, em vez de ser instanciado, ex.:

```
$builder->add('name', 'custom_type');
```

em vez de:

```
$builder->add('name', new CustomType());
```

Validação de Metadados

Em andamento..

Traduções

Em andamento...

Como expor uma Configuração Semântica para um Bundle

Se você abrir o arquivo de configuração da sua aplicação (geralmente `app/config/config.yml`), verá um número de diferentes "namespaces" de configurações, como `framework`, `twig` e `doctrine`. Cada um deles configura um bundle específico, permitindo configurar as coisas a um alto nível e deixar o bundle fazer todas as modificações complexas, de baixo nível resultantes.

Por exemplo, o código a seguir diz ao `FrameworkBundle` para habilitar a integração do formulário, a qual envolve a definição de alguns serviços, bem como a integração de outros componentes relacionados:

- *YAML*

```
framework:
    # ...
    form:             true
```

- *XML*

```
<framework:config>
    <framework:form />
</framework:config>
```

- *PHP*

```
$container->loadFromExtension('framework', array(
    // ...
    'form'            => true,
    // ...
));
```

Quando você cria um bundle, você tem duas opções para lidar com a configuração:

1. **Configuração de Serviço Normal** (*fácil*):

   Você pode especificar seus serviços em um arquivo de configuração (por exemplo `services.yml`) que reside em seu bundle e, então, importá-lo a partir da configuração principal da sua aplicação. Isto é realmente fácil, rápido e totalmente eficaz. Se você fazer uso de *parâmetros*, então, você ainda tem a flexibilidade para personalizar seu bundle a partir da configuração da sua aplicação. Veja "*Importando configuração com imports*" para mais detalhes.

2. **Expondo Configuração Semântica** (*avançado*):

   Esta é a maneira como a configuração é feita com os bundles do núcleo (como descrito acima). A idéia básica é que, em vez de o usuário sobrescrever parâmetros individuais, você deixa ele configurar apenas algumas opções criadas . Como desenvolvedor do bundle, você então faz o parse desta configuração e carrega os serviços dentro de uma classe "Extension". Com este método, você não vai precisar importar quaisquer recursos de configuração a partir da configuração principal da sua aplicação: a classe de Extensão (Extension) pode lidar com tudo isso.

A segunda opção - que você vai aprender neste artigo - é muito mais flexível, mas também requer mais tempo para configurar. Se você está se perguntando qual método deve usar, provavelmente é uma boa idéia começar com o método 1, e depois mudar para o 2 mais tarde, se você precisar.

O segundo método tem várias vantagens específicas:

- Muito mais poderoso do que simplesmente definir parâmetros: um valor de opção específico pode acionar a criação de muitas definições de serviço;
- Possibilidade de ter hierarquia de configuração
- Smart merging quando possuir vários arquivos de configuração (por exemplo `config_dev.yml` e `config.yml`) sobrescrevendo a configuração um do outro;
- Validação de configuração (se você usar uma *Classe de Configuração*);
- Auto-completar da IDE quando você criar um XSD e os desenvolvedores usarem XML.

Sobresecrevendo parâmetros do bundle

Se um Bundle fornecer uma classe Extension, então, você geralmente *não* deve sobrescrever quaisquer parâmetros do container de serviço daquele bundle. A idéia é que, se uma classe Extension estiver presente, cada definição que deve ser configurável deve estar presente na configuração disponibilizada por esta classe. Em outras palavras, a classe Extension as definições de configuração públicas suportadas para as quais a compatibilidade com versões anteriores será mantida.

Criando uma Classe Extension

Se você optar por expor uma configuração semântica para seu bundle, você vai precisar primeiro criar uma nova classe "Extension", que irá lidar com o processo. Esta classe deve residir no diretório `DependencyInjection` de seu bundle e o seu nome deve ser construído substituindo o sufixo `Bundle` do nome da classe Bundle por `Extension`. Por exemplo, a classe Extension do `AcmeHelloBundle` seria chamada `AcmeHelloExtension`:

```
// Acme/HelloBundle/DependencyInjection/AcmeHelloExtension.php
namespace Acme\HelloBundle\DependencyInjection;

use Symfony\Component\HttpKernel\DependencyInjection\Extension;
use Symfony\Component\DependencyInjection\ContainerBuilder;

class AcmeHelloExtension extends Extension
{
    public function load(array $configs, ContainerBuilder $container)
    {
        // ... where all of the heavy logic is done
    }

    public function getXsdValidationBasePath()
    {
        return __DIR__.'/../Resources/config/';
    }

    public function getNamespace()
    {
        return 'http://www.example.com/symfony/schema/';
    }
}
```

Nota: Os métodos `getXsdValidationBasePath` e `getNamespace` são necessários apenas se o bundle fornece XSDs opcionais para a configuração.

A presença da classe anterior significa que agora você pode definir um namespace de configuração `acme_hello` em qualquer arquivo de configuração. O namespace `acme_hello` é construído a partir do nome da classe de extensão, removendo a palavra `Extension` e, em seguida, deixando o resto do nome todo em letras minúsculas e com underscores. Em outras palavras, `AcmeHelloExtension` torna-se `acme_hello`.

Você pode começar a especificar a configuração sob este namespace imediatamente:

- *YAML*

```yaml
# app/config/config.yml
acme_hello: ~
```

- *XML*

```xml
<!-- app/config/config.xml -->
<?xml version="1.0" ?>

<container xmlns="http://symfony.com/schema/dic/services"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:acme_hello="http://www.example.com/symfony/schema/"
    xsi:schemaLocation="http://www.example.com/symfony/schema/ http://www.example.com/symfony/sc

   <acme_hello:config />

   <!-- ... -->
</container>
```

- *PHP*

```php
// app/config/config.php
$container->loadFromExtension('acme_hello', array());
```

Dica: Se você seguir as convenções de nomenclatura estabelecidas acima, então, o método `load()` de seu código de extensão é sempre chamado, uma vez que seu bundle está registrado no Kernel. Em outras palavras, mesmo se o usuário não fornecer qualquer configuração (ou seja, a entrada `acme_hello` nem mesmo aparecer), o método `load()` será chamado e passado um array `$configs` vazio. Você ainda pode fornecer alguns padrões para seu bundle se desejar.

Fazendo o parse do array `$configs`

Sempre que um usuário inclui o namespace `acme_hello` em um arquivo de configuração, a configuração abaixo dele é adicionada à um array de configurações e passado para o método `load()` de sua extensão (o Symfony2 automaticamente converte XML e YAML para um array).

Assuma a seguinte configuração:

- *YAML*

```yaml
# app/config/config.yml
acme_hello:
    foo: fooValue
    bar: barValue
```

- *XML*

```
<!-- app/config/config.xml -->
<?xml version="1.0" ?>

<container xmlns="http://symfony.com/schema/dic/services"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:acme_hello="http://www.example.com/symfony/schema/"
    xsi:schemaLocation="http://www.example.com/symfony/schema/ http://www.example.com/symfony/sc

    <acme_hello:config foo="fooValue">
        <acme_hello:bar>barValue</acme_hello:bar>
    </acme_hello:config>

</container>
```

- *PHP*

```
// app/config/config.php
$container->loadFromExtension('acme_hello', array(
    'foo' => 'fooValue',
    'bar' => 'barValue',
));
```

O array passado para seu método `load()` ficará parecido com o seguinte:

```
array(
    array(
        'foo' => 'fooValue',
        'bar' => 'barValue',
    )
)
```

Observe que este é um *array de arrays*, e não apenas um único array simples de valores de configuração. Isso é intencional. Por exemplo, se `acme_hello` aparece em outro arquivo de configuração - digamos `config_dev.yml` - com valores diferentes abaixo dele, então, o array de entrada poderia ser assim:

```
array(
    array(
        'foo' => 'fooValue',
        'bar' => 'barValue',
    ),
    array(
        'foo' => 'fooDevValue',
        'baz' => 'newConfigEntry',
    ),
)
```

A ordem dos dois arrays depende de qual é definido primeiro.

É o seu trabalho, então, decidir como deve ser o merge dessas configurações . Você pode, por exemplo, ter valores posteriores sobrescrevendo os valores anteriores ou de alguma forma fazer o merge deles.

Mais tarde, na seção sobre a *Classe de Configuração* , você vai aprender uma forma verdadeiramente robusta para lidar com isso. Mas, por ora, você pode apenas fazer o merge manualmente:

```
public function load(array $configs, ContainerBuilder $container)
{
    $config = array();
    foreach ($configs as $subConfig) {
        $config = array_merge($config, $subConfig);
```

```
    }

    // ... now use the flat $config array
}
```

Cuidado: Certifique-se que a técnica de merge acima faz sentido para o seu bundle. Este é apenas um exemplo, e você deve ter cuidado para não usá-lo cegamente.

Usando o Método `load()`

Dentro do `load()` a variável `$container` refere-se a um container que apenas sabe sobre essa configuração de namespace (ou seja, não contêm informação de serviço carregada a partir de outros bundles). O objetivo do método `load()` é manipular o container, adicionando e configurando quaisquer métodos ou serviços necessários ao seu bundle.

Carregando Recursos de Configuração Externos Algo comum de se fazer é carregar um arquivo de configuração externo que pode conter a maioria dos serviços necessários para o seu bundle. Por exemplo, suponha que você tem um arquivo `services.xml` que contém muitas das configurações de serviços do seu bundle:

```
use Symfony\Component\DependencyInjection\Loader\XmlFileLoader;
use Symfony\Component\Config\FileLocator;

public function load(array $configs, ContainerBuilder $container)
{
    // ... prepare your $config variable

    $loader = new XmlFileLoader($container, new FileLocator(__DIR__.'/../Resources/config'));
    $loader->load('services.xml');
}
```

Você pode até fazer isso condicionalmente, com base em um dos valores de configuração. Por exemplo, supondo que você quer carregar um conjunto de serviços somente se a opção `enabled` for passada e definida com true:

```
public function load(array $configs, ContainerBuilder $container)
{
    // ... prepare your $config variable

    $loader = new XmlFileLoader($container, new FileLocator(__DIR__.'/../Resources/config'));

    if (isset($config['enabled']) && $config['enabled']) {
        $loader->load('services.xml');
    }
}
```

Configurando Serviços e Definindo Parâmetros Uma vez que você já carregou alguma configuração de serviço, você pode precisar modificar a configuração com base em alguns dos valores de entrada. Por exemplo, supondo que existe um serviço cujo primeiro argumento é alguma string "type" que ele irá usar internamente. Você gostaria que isto fosse facilmente configurado pelo usuário do bundle, então, em seu arquivo de configuração do serviço (ex. `services.xml`), você define este serviço e usa um parâmetro em branco - `acme_hello.my_service_type` - como seu primeiro argumento:

```
<!-- src/Acme/HelloBundle/Resources/config/services.xml -->
<container xmlns="http://symfony.com/schema/dic/services"
```

```
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/dic/services http://symfony.com/schema/dic/services

    <parameters>
        <parameter key="acme_hello.my_service_type" />
    </parameters>

    <services>
        <service id="acme_hello.my_service" class="Acme\HelloBundle\MyService">
            <argument>%acme_hello.my_service_type%</argument>
        </service>
    </services>
</container>
```

Mas por que definir um parâmetro vazio e então passá-lo ao seu serviço? A resposta é que você vai definir este parâmetro em sua classe de extensão, com base nos valores de configuração de entrada. Suponha, por exemplo, que você quer permitir ao usuário definir esta opção *type* sob uma chave chamada `my_type`. Para fazer isso, adicione o seguinte ao método `load()`:

```
public function load(array $configs, ContainerBuilder $container)
{
    // ... prepare your $config variable

    $loader = new XmlFileLoader($container, new FileLocator(__DIR__.'/../Resources/config'));
    $loader->load('services.xml');

    if (!isset($config['my_type'])) {
        throw new \InvalidArgumentException('The "my_type" option must be set');
    }

    $container->setParameter('acme_hello.my_service_type', $config['my_type']);
}
```

Agora, o usuário pode efetivamente configurar o serviço especificando o valor de configuração `my_type`:

- *YAML*

```
    # app/config/config.yml
    acme_hello:
        my_type: foo
        # ...
```

- *XML*

```
    <!-- app/config/config.xml -->
    <?xml version="1.0" ?>

    <container xmlns="http://symfony.com/schema/dic/services"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:acme_hello="http://www.example.com/symfony/schema/"
        xsi:schemaLocation="http://www.example.com/symfony/schema/ http://www.example.com/symfony/sc

        <acme_hello:config my_type="foo">
            <!-- ... -->
        </acme_hello:config>

    </container>
```

- *PHP*

```
    // app/config/config.php
    $container->loadFromExtension('acme_hello', array(
        'my_type' => 'foo',
        // ...
    ));
```

Parâmetros Globais Quando estiver configurando o container, esteja ciente de que você tem os seguintes parâmetros globais disponíveis para uso:

- `kernel.name`
- `kernel.environment`
- `kernel.debug`
- `kernel.root_dir`
- `kernel.cache_dir`
- `kernel.logs_dir`
- `kernel.bundle_dirs`
- `kernel.bundles`
- `kernel.charset`

Cuidado: Todos os nomes de parâmetros e serviços, começando com um _ são reservados para o framework, e os novos não devem ser definidos por bundles.

Validação e Merging com uma Classe de Configuração

Até agora, você já fez o merge de seus arrays de configuração manualmente e está verificando a presença de valores de configuração manualmente usando a função `isset()` do PHP. Um sistema opcional de *Configuração* está também disponível que pode ajudar com merge, validação, valores padrão e normalização de formato.

Nota: Normalização de formato refere-se ao fato de que certos formatos - em grande parte XML - resultam em arrays de configuração ligeiramente diferentes e que estes arrays precisam ser "normalizados" para corresponder com todo o resto.

Para tirar vantagem deste sistema, você vai criar uma classe `Configuration` e construir uma árvore que define a sua configuração nesta classe:

```
// src/Acme/HelloBundle/DependencyInjection/Configuration.php
namespace Acme\HelloBundle\DependencyInjection;

use Symfony\Component\Config\Definition\Builder\TreeBuilder;
use Symfony\Component\Config\Definition\ConfigurationInterface;

class Configuration implements ConfigurationInterface
{
    public function getConfigTreeBuilder()
    {
        $treeBuilder = new TreeBuilder();
        $rootNode = $treeBuilder->root('acme_hello');
```

```
        $rootNode
            ->children()
            ->scalarNode('my_type')->defaultValue('bar')->end()
            ->end();

        return $treeBuilder;
    }
```

Este é um exemplo *muito* simples, mas agora você pode usar essa classe em seu método `load()` para o merge da sua configuração e forçar a validação. Se outras opções que não sejam `my_type` forem passadas, o usuário será notificado com uma exceção de que uma opção não suportada foi passada:

```
public function load(array $configs, ContainerBuilder $container)
{
    $configuration = new Configuration();

    $config = $this->processConfiguration($configuration, $configs);

    // ...
}
```

O método `processConfiguration()` usa a árvore de configuração que você definiu na classe `Configuration` para validar, normalizar e fazer o merge de todos os arrays de configuração em conjunto.

A classe "Configuration' pode ser muito mais complicada do que mostramos aqui, suportando array nodes, "prototype" nodes, validação avançada, normalização específica de XML e merge avançado. Você pode ler mais sobre isso na `documentação do Componente de Configuração`. Você também pode vê-lo em ação verificando algumas das classes de Configuração do núcleo, tais como a Configuração do FrameworkBundle ou a Configuração do TwigBundle.

Dump de Configuração Padrão Novo na versão 2.1: O comando `config:dump-reference` foi adicionado no Symfony 2.1

O comando `config:dump-reference` permite que a configuração padrão de um bundle seja impressa no console em YAML.

Enquanto a configuração do bundle está localizada no local padrão (`YourBundle\DependencyInjection\Configuration`) e não tem um `__constructor()` ele vai funcionar automaticamente. Se você tem algo diferente a sua classe `Extension` terá que sobrescrever o método `Extension::getConfiguration()`. Fazendo ele retornar uma instância de sua `Configuration`.

Comentários e exemplos podem ser adicionados aos nós de configuração utilizando os métodos `->info()` e `->example()`:

```
// src/Acme/HelloBundle/DependencyExtension/Configuration.php
namespace Acme\HelloBundle\DependencyInjection;

use Symfony\Component\Config\Definition\Builder\TreeBuilder;
use Symfony\Component\Config\Definition\ConfigurationInterface;

class Configuration implements ConfigurationInterface
{
    public function getConfigTreeBuilder()
    {
        $treeBuilder = new TreeBuilder();
        $rootNode = $treeBuilder->root('acme_hello');

        $rootNode
```

```
            ->children()
                ->scalarNode('my_type')
                    ->defaultValue('bar')
                    ->info('what my_type configures')
                    ->example('example setting')
                ->end()
            ->end()
        ;

        return $treeBuilder;
    }
```

Este texto aparece como comentários yaml na saída do comando `config:dump-reference`.

Convenções de Extensão

Ao criar uma extensão, siga estas convenções simples:

- A extensão deve ser armazenada no sub-namespace `DependencyInjection`;
- A extensão deve ser nomeada após o nome do bundle e com o sufixo `Extension` (`AcmeHelloExtension` para `AcmeHelloBundle`);
- A extensão deve fornecer um esquema XSD.

Se você seguir estas convenções simples, suas extensões serão registradas automaticamente pelo Symfony2. Se não, sobrescreva o método Bundle `build()` em seu bundle:

```
// ...
use Acme\HelloBundle\DependencyInjection\UnconventionalExtensionClass;

class AcmeHelloBundle extends Bundle
{
    public function build(ContainerBuilder $container)
    {
        parent::build($container);

        // register extensions that do not follow the conventions manually
        $container->registerExtension(new UnconventionalExtensionClass());
    }
}
```

Neste caso, a classe de extensão também deve implementar um método `getAlias()` e retornar um alias exclusivo nomeado após o bundle (por exemplo, `acme_hello`). Isto é necessário porque o nome da classe não segue os padrões terminando em `Extension`.

Além disso, o método `load()` de sua extensão será *apenas* chamado se o usuário especificar o alias `acme_hello` em pelo menos um arquivo de configuração . Mais uma vez, isso é porque a classe de extensão não segue os padrões acima referidos, de modo que, nada acontece automaticamente.

3.1.10 Email

Como enviar um e-mail

Enviar e-mails é uma tarefa clássica para qualquer aplicação web e, possui complicações especiais e potenciais armadilhas. Em vez de recriar a roda, uma solução para enviar e-mails é usar o `SwiftmailerBundle`, que aproveita o poder da biblioteca Swiftmailer.

Nota: Não esqueça de ativar o bundle em seu kernel antes de usá-lo:

```
public function registerBundles()
{
    $bundles = array(
        // ...
        new Symfony\Bundle\SwiftmailerBundle\SwiftmailerBundle(),
    );

    // ...
}
```

Configuração

Antes de usar o Swiftmailer, não esqueça de incluir a sua configuração. O único parâmetro de configuração obrigatório é o `transport`:

- *YAML*

```
# app/config/config.yml
swiftmailer:
    transport:  smtp
    encryption: ssl
    auth_mode:  login
    host:       smtp.gmail.com
    username:   your_username
    password:   your_password
```

- *XML*

```
<!-- app/config/config.xml -->

<!--
xmlns:swiftmailer="http://symfony.com/schema/dic/swiftmailer"
http://symfony.com/schema/dic/swiftmailer http://symfony.com/schema/dic/swiftmailer/swiftmailer-
-->

<swiftmailer:config
    transport="smtp"
    encryption="ssl"
    auth-mode="login"
    host="smtp.gmail.com"
    username="your_username"
    password="your_password" />
```

- *PHP*

```
// app/config/config.php
$container->loadFromExtension('swiftmailer', array(
    'transport'  => "smtp",
    'encryption' => "ssl",
    'auth_mode'  => "login",
    'host'       => "smtp.gmail.com",
    'username'   => "your_username",
    'password'   => "your_password",
));
```

A maioria das configurações do Swiftmailer lidam com a forma como as mensagens devem ser entregues.

Os seguintes atributos de configuração estão disponíveis:

- `transport` (`smtp`, `mail`, `sendmail`, ou `gmail`)
- `username`
- `password`
- `host`
- `port`
- `encryption` (`tls`, ou `ssl`)
- `auth_mode` (`plain`, `login`, ou `cram-md5`)
- `spool`
  - `type` (como será o queue de mensagens, são suportados `file` ou `memory`, veja Como fazer Spool de E-mail)
  - `path` (onde armazenar as mensagens)
- `delivery_address` (um endereço de email para onde serão enviados TODOS os e-mails)
- `disable_delivery` (defina como true para desabilitar completamente a entrega)

Enviando e-mails

A biblioteca Swiftmailer funciona através da criação, configuração e, então, o envio de objetos `Swift_Message`. O "mailer" é responsável pela entrega da mensagem e é acessível através do serviço `mailer`. No geral, o envio de um e-mail é bastante simples:

```
public function indexAction($name)
{
    $message = \Swift_Message::newInstance()
        ->setSubject('Hello Email')
        ->setFrom('send@example.com')
        ->setTo('recipient@example.com')
        ->setBody($this->renderView('HelloBundle:Hello:email.txt.twig', array('name' => $name)))
    ;
    $this->get('mailer')->send($message);

    return $this->render(...);
}
```

Para manter as coisas desacopladas, o corpo do e-mail foi armazenado em um template e renderizado através do método `renderView()`.

O objeto `$message` suporta mais opções, como, a inclusão de anexos, a adição de conteúdo HTML, e muito mais. Felizmente, o Swiftmailer cobre o tópico Criação de Mensagens em grande detalhe na sua documentação.

Dica: Vários outros artigos cookbook relacionados ao envio de e-mails estão disponíveis no Symfony2:

- Como usar o Gmail para enviar E-mails
- Como Trabalhar com E-mails Durante o Desenvolvimento
- Como fazer Spool de E-mail

Como usar o Gmail para enviar E-mails

Durante o desenvolvimento, em vez de usar um servidor SMTP regular para enviar e-mails, você pode descobrir que usar o Gmail é mais fácil e prático. O bundle Swiftmailer torna esta tarefa realmente fácil.

Dica: Em vez de usar a sua conta do Gmail normal, é, com certeza, recomendado que você crie uma conta especial para este propósito.

No arquivo de configuração de desenvolvimento, altere a definição `transport` para `gmail` e defina o `username` e `password` com as credenciais do Google:

- *YAML*

```
# app/config/config_dev.yml
swiftmailer:
    transport: gmail
    username:  your_gmail_username
    password:  your_gmail_password
```

- *XML*

```
<!-- app/config/config_dev.xml -->

<!--
    xmlns:swiftmailer="http://symfony.com/schema/dic/swiftmailer"
    http://symfony.com/schema/dic/swiftmailer http://symfony.com/schema/dic/swiftmailer/swiftmai
-->

<swiftmailer:config
    transport="gmail"
    username="your_gmail_username"
    password="your_gmail_password" />
```

- *PHP*

```
// app/config/config_dev.php
$container->loadFromExtension('swiftmailer', array(
    'transport' => "gmail",
    'username'  => "your_gmail_username",
    'password'  => "your_gmail_password",
));
```

Está pronto!

Nota: O transporte `gmail` é simplesmente um atalho que usa o transporte `smtp` e seta as definições `encryption`, `auth_mode` e `host` para funcionar com o Gmail.

Como Trabalhar com E-mails Durante o Desenvolvimento

Quando você estiver criando uma aplicação que envia e-mails, muitas vezes não vai desejar enviar os e-mails ao destinatário especificado, durante o desenvolvimento. Se você estiver usando o `SwiftmailerBundle` com o Symfony2, poderá facilmente conseguir isso através de definições de configuração sem ter que fazer quaisquer alterações no código da sua aplicação. Existem duas opções principais quando se trata de manipulação de e-mails durante o desenvolvimento: (a) desativação do envio de e-mails totalmente ou (b) o envio de todos os e-mails para um endereço especificado.

Desativando o Envio

Você pode desativar o envio de e-mails, definindo a opção `disable_delivery` para `true`. Este é o padrão para o ambiente `test` na distribuição Standard. Se você fizer isso especificamente na configuração `test`, então os emails não será enviados quando você executar testes, mas continuarão a ser enviados nos ambientes `prod` e `dev`:

- *YAML*

```yaml
# app/config/config_test.yml
swiftmailer:
    disable_delivery:  true
```

- *XML*

```xml
<!-- app/config/config_test.xml -->

<!--
xmlns:swiftmailer="http://symfony.com/schema/dic/swiftmailer"
http://symfony.com/schema/dic/swiftmailer http://symfony.com/schema/dic/swiftmailer/swiftmailer-
-->

<swiftmailer:config
    disable-delivery="true" />
```

- *PHP*

```php
// app/config/config_test.php
$container->loadFromExtension('swiftmailer', array(
    'disable_delivery'  => "true",
));
```

Se você também gostaria de desativar a entrega no ambiente `dev`, simplesmente adicione esta configuração ao arquivo `config_dev.yml`.

Enviando para um Endereço Especificado

Você também pode optar por enviar todos os emails para um endereço específico, em vez do endereço atualmente especificado, ao enviar a mensagem. Isto pode ser feito através da opção `delivery_address`:

- *YAML*

```yaml
# app/config/config_dev.yml
swiftmailer:
    delivery_address:  dev@example.com
```

- *XML*

```xml
<!-- app/config/config_dev.xml -->

<!--
xmlns:swiftmailer="http://symfony.com/schema/dic/swiftmailer"
http://symfony.com/schema/dic/swiftmailer http://symfony.com/schema/dic/swiftmailer/swiftmailer-
-->

<swiftmailer:config
    delivery-address="dev@example.com" />
```

- *PHP*

```
    // app/config/config_dev.php
    $container->loadFromExtension('swiftmailer', array(
        'delivery_address'  => "dev@example.com",
    ));
```

Agora, suponha que você está enviando um email para `recipient@example.com`.

```
public function indexAction($name)
{
    $message = \Swift_Message::newInstance()
        ->setSubject('Hello Email')
        ->setFrom('send@example.com')
        ->setTo('recipient@example.com')
        ->setBody($this->renderView('HelloBundle:Hello:email.txt.twig', array('name' => $name)))
    ;
    $this->get('mailer')->send($message);

    return $this->render(...);
}
```

No ambiente `dev`, o e-mail será enviado para `dev@example.com`. O Swiftmailer irá adicionar um cabeçalho extra para o e-mail, `X-Swift-To` contendo o endereço substituído, assim você ainda poderá visualizar para quem ele teria sido enviado.

Nota: Além do endereço `to`, ele também irá parar os e-mails sendo enviados para quaisquer endereços `CC` e `BCC` definidos. O SwiftMailer irá adicionar cabeçalhos adicionais para o e-mail com os endereços substituídos neles. Eles são `X-Swift-Cc` e `X-Swift-Bcc` para os endereços `CC` e `BCC`, respectivamente.

Visualização na Barra de Ferramentas de Debug Web

Você pode visualizar quaisquer e-mails enviados por uma página quando estiver no ambiente `dev` usando a Barra de Ferramentas para Debug Web. O ícone de e-mail na barra de ferramentas irá mostrar quantos e-mails foram enviados. Se você clicar nele, um relatório mostrando os detalhes dos e-mails será aberto.

Se você estiver enviando um e-mail e imediatamente executar um redirecionamento, você precisará definir a opção `intercept_redirects` para `true` no arquivo `config_dev.yml` para que possa ver o e-mail na barra de ferramentas de debug web antes de ser redirecionado.

Como fazer Spool de E-mail

Quando você estiver usando o `SwiftmailerBundle` para enviar um email de uma aplicação Symfony2, ele irá, por padrão, enviar o e-mail imediatamente. Você pode, entretanto, desejar evitar um impacto no desempenho da comunicação entre o `Swiftmailer` e o transporte do e-mail, o que poderia fazer com que o usuário tenha que aguardar a próxima página carregar, enquanto está enviando o e-mail. Isto pode ser evitado escolhendo pelo "spool" dos e-mails em vez de enviá-los diretamente. Isto significa que o `Swiftmailer` não tentará enviar o email, mas, ao invés, salvará a mensagem em algum lugar, como um arquivo. Outro processo poderá então ler a partir do spool e cuidar de enviar os e-mails no spool. Atualmente, apenas o spool para arquivo ou memória são suportados pelo `Swiftmailer`.

Spool usando memória

Quando você usa o spool para armazenar os e-mails em memória, eles são enviados mesmo antes do kernel terminar. Isto significa que o e-mail só é enviado se o pedido foi todo executado sem qualquer exceção não tratada ou quaisquer erros. Para configurar o SwiftMailer com a opção de memória, utilize a seguinte configuração:

- *YAML*

```yaml
# app/config/config.yml
swiftmailer:
    # ...
    spool: { type: memory }
```

- *XML*

```xml
<!-- app/config/config.xml -->

<!--
    xmlns:swiftmailer="http://symfony.com/schema/dic/swiftmailer"
    http://symfony.com/schema/dic/swiftmailer http://symfony.com/schema/dic/swiftmailer/swiftmai
-->

<swiftmailer:config>
     <swiftmailer:spool type="memory" />
</swiftmailer:config>
```

- *PHP*

```php
// app/config/config.php
$container->loadFromExtension('swiftmailer', array(
     ...,
    'spool' => array('type' => 'memory')
));
```

Spool usando um arquivo

Para utilizar o spool com um arquivo, use a seguinte configuração:

- *YAML*

```yaml
# app/config/config.yml
swiftmailer:
    # ...
    spool:
        type: file
        path: /path/to/spool
```

- *XML*

```xml
<!-- app/config/config.xml -->

<!--
xmlns:swiftmailer="http://symfony.com/schema/dic/swiftmailer"
http://symfony.com/schema/dic/swiftmailer http://symfony.com/schema/dic/swiftmailer/swiftmailer-
-->

<swiftmailer:config>
     <swiftmailer:spool
```

```
            type="file"
            path="/path/to/spool" />
    </swiftmailer:config>
```

- *PHP*

```
    // app/config/config.php
    $container->loadFromExtension('swiftmailer', array(
        // ...
        'spool' => array(
            'type' => 'file',
            'path' => '/path/to/spool',
        )
    ));
```

Dica: Se você deseja armazenar o spool em algum lugar no diretório do seu projeto, lembre-se que você pode usar o parâmetro *%kernel.root_dir%* para referenciar o raiz do seu projeto:

```
path: %kernel.root_dir%/spool
```

Agora, quando a sua aplicação enviar um e-mail, ele não será realmente enviado, ao invés, será adicionado ao spool. O envio de mensagens do spool é feito separadamente. Existe um comando do console para enviar as mensagens que encontram-se no spool:

```
php app/console swiftmailer:spool:send
```

Ele tem uma opção para limitar o número de mensagens a serem enviadas:

```
php app/console swiftmailer:spool:send --message-limit=10
```

Você também pode definir o limite de tempo em segundos:

```
php app/console swiftmailer:spool:send --time-limit=10
```

Claro que, na realidade, você não vai querer executar ele manualmente. Em vez disso, o comando do console deve ser disparado por um cron job ou tarefa agendada e executar em um intervalo regular.

3.1.11 Segurança

Listas de controle de acesso (ACLs)

Em aplicativos complexos, comumente existem o problema que as decisões de permitir ou negar acesso não podem ser tomadas somente baseada no usuário (`Token`) solicitando acesso, mas também deve levar em consideração o objeto de domínio que está tendo o acesso solicitado. É aí que o sistema ACL entra em ação.

Imagine que está projetando um sistema de blog onde seus usuário podem comentar os textos (posts) publicados. Agora, você deseja que um usuário possa editar seus próprios comentários, mas não os comentários dos outros usuários. Além disso, você como administrador deseja pode editar todos os comentários. Neste cenário, `Comment` seria seu objeto de domínio ao qual você quer restringir acesso. Você poderia usar várias abordagens para conseguir o mesmo resultado. Duas dessas seriam:

- *Impor segurança em seus métodos*: Basicamente, isso significa que deverá manter referências em cada `Comment` de todos os usuários que têm acesso e depois comparar com o usuário `Token` solicitando acesso.
- *Impor segurança com perfis*: Nesta abordagem, você adicionaria um perfil para cada objeto `Comment`, isto é, `ROLE_COMMENT_1`, `ROLE_COMMENT_2`, etc.

Ambas abordagens são perfeitamnete válidas. Elas, porém, amarram sua lógica de autorização de acesso com seu código, deixando-o mais difícil de reusar em outros contextos. Também aumenta a dificuldade de criar testes unitários. Além disso, pode-se ter problemas de performance caso muitos usuários tenham acesso a um único objeto de domínio.

Felizmente, há uma maneira melhor que veremos a seguir.

Configuração

Agora, antes de realmente começarmos, precisamos fazer algumas configurações. Primeiramente, precisamos configurar a conexão de banco de dados queo sistema ACL utilizará.

- *YAML*

```yaml
# app/config/security.yml
security:
    acl:
        connection: default
```

- *XML*

```xml
<!-- app/config/security.xml -->
<acl>
    <connection>default</connection>
</acl>
```

- *PHP*

```php
// app/config/security.php
$container->loadFromExtension('security', 'acl', array(
    'connection' => 'default',
));
```

Nota: O sistema ACL requer que ao menos uma conexão Doctrine DBAL esteja configurada. Isto, porém, não significa que você tem que utilizar o Doctrine para mapear seus objetos de domínio. Você pode utilizar qualquer mapeamento que quiser para seus objetos, seja ele Doctrine ORM, Mongo ODM, Propel, ou SQL puro. A escolha é sua.

Depois de configurar a conexão, temos que importar a estrutura do banco de dados. Felizmente, temos um comando para isto. Rode o seguinte comando.

```
php app/console init:acl
```

Começando

Voltando ao nosso pequeno exemplo do início, vamos implementar o sistema ACL dele.

Criando uma ACL, e adicionando uma entrada (ACE)

```php
use Symfony\Component\Security\Core\Exception\AccessDeniedException;
use Symfony\Component\Security\Acl\Domain\ObjectIdentity;
use Symfony\Component\Security\Acl\Domain\UserSecurityIdentity;
use Symfony\Component\Security\Acl\Permission\MaskBuilder;
// ...

// BlogController.php
```

```
public function addCommentAction(Post $post)
{
    $comment = new Comment();

    // setup $form, and bind data
    // ...

    if ($form->isValid()) {
        $entityManager = $this->get('doctrine.orm.default_entity_manager');
        $entityManager->persist($comment);
        $entityManager->flush();

        // creating the ACL
        $aclProvider = $this->get('security.acl.provider');
        $objectIdentity = ObjectIdentity::fromDomainObject($comment);
        $acl = $aclProvider->createAcl($objectIdentity);

        // retrieving the security identity of the currently logged-in user
        $securityContext = $this->get('security.context');
        $user = $securityContext->getToken()->getUser();
        $securityIdentity = UserSecurityIdentity::fromAccount($user);

        // grant owner access
        $acl->insertObjectAce($securityIdentity, MaskBuilder::MASK_OWNER);
        $aclProvider->updateAcl($acl);
    }
}
```

Há algumas importantes decisões de implementação neste trecho de código. Por enquanto, gostaria de destacar duas.

Primeiro, note que o método `->createAcl()` não aceita objetos de domínio diretamente, mas somente implementações de `ObjectIdentityInterface`. Este passo adicional permite que trabalhe com ACLs mesmo quando não tiver uma instância do objeto de domínio disponível. Isto será extremamente útil se você quiser verificar permissões para um grande número de objetos sem realmente criar os objetos.

Outra parte interessante é a chamada `->insertObjectAce()`. Em nosso exemplo, estamos concedendo ao usuário que está autenticado permissão de proprietário do objeto Comment. `MaskBuilder::MASK_OWNER` é uma máscara (integer bitmask) pré-definida. Não se preocupe que MaskBuilder abstrai a maior parte dos detalhes técnicos, mas saiba que utilizando esta técnica é possível armazenar muitas permissões diferentes em apenas uma linha do banco de dados, o que significa uma considerável melhora na performance.

Dica: A ordem em que as entradas de controle (ACE) são checadas é importante. Como regra geral, você deve colocar as entradas mais específicas no início.

Verificando o acesso

```
// BlogController.php
public function editCommentAction(Comment $comment)
{
    $securityContext = $this->get('security.context');

    // check for edit access
    if (false === $securityContext->isGranted('EDIT', $comment))
    {
        throw new AccessDeniedException();
    }
```

```
    // retrieve actual comment object, and do your editing here
    // ...
}
```

Neste exemplo, verificamos se o usuário tem permissão de edição (`EDIT`). Internamente, Symfony2 mapea a permissão para várias máscaras (integer bitmasks) e verifica se o usuário tem alguma delas.

Nota: Você pode definir até 32 permissões base (dependendo do seu SO, pode variar entre 30 e 32). Você ainda pode definir permisões cumulativas.

Permissões Cumulativas

No nosso primeiro exemplo acima, nós concedemos somente a permissão base `OWNER`. Apesar disso significar que o usuário pode executar qualquer operação no objeto de domínio tais como exibir, editar, etc, em alguns casos você pode querer conceder essas permissões explicitamente.

O `MaskBuilder` pode ser usado para criar máscaras (bit masks) facilmente através da combinação de várias permissões base.

```
$builder = new MaskBuilder();
$builder
    ->add('view')
    ->add('edit')
    ->add('delete')
    ->add('undelete')
;
$mask = $builder->get(); // int(15)
```

Este inteiro (integer bitmask) pode então ser usado para conceder a um usuário todas as permissões base que você adicionou acima.

```
$acl->insertObjectAce(new UserSecurityIdentity('johannes'), $mask);
```

O usuário agora poderá exibir, editar, deletar e desfazer a deleção dos objetos.

Como usar Conceitos Avançados de ACL

O objetivo deste capítulo é fornecer uma visão mais aprofundada do sistema de ACL, e também explicar algumas das decisões de projeto por trás dele.

Conceitos de Projeto

Os recursos de uma instância do objeto de segurança do Symfony2 tem base no conceito de uma Lista de Controle de Acesso (ACL). Cada **instância** do objeto de domínio tem a sua própria ACL. A instância ACL contém uma lista detalhada de Entradas de Controle de Acesso (ACEs), que são usadas para tomar decisões de acesso. O sistema ACL do Symfony2 concentra-se em dois objetivos principais:

- fornecer uma maneira de recuperar uma grande quantidade de ACLs/ACEs de forma eficiente para os seus objetos de domínio e para modificá-los;
- fornecer uma maneira de tomar decisões facilmente para saber se uma pessoa está autorizada a executar uma ação em um objeto de domínio ou não.

Conforme indicado no primeiro ponto, uma das principais capacidades do sistema ACL do Symfony2 é uma forma de recuperar ACLs/ACEs com alto desempenho. Isto é extremamente importante, pois cada ACL pode ter várias ACEs, e herdam de outra ACL em forma de árvore. Portanto, nenhum ORM é utilizado, em vez disso, a implementação padrão interage com a sua conexão diretamente usando o DBAL do Doctrine.

Identidades de Objeto O sistema de ACL é completamente desacoplado de seus objetos de domínio. Eles não tem que ser armazenados na mesma base de dados, ou no mesmo servidor. De forma a atingir esse desacoplamento, os seus objetos, no sistema ACL, são representados através de objetos de identidade de objeto. Toda vez que você deseja recuperar a ACL para um objeto de domínio, o sistema de ACL vai primeiro criar uma identidade de objeto do seu objeto de domínio e, em seguida, passar essa identidade de objeto ao provedor ACL para processamento adicional.

Identidades de Segurança Isto é análogo à identidade de objeto, mas representa um usuário ou um papel em sua aplicação. Cada papel ou usuário tem a sua própria identidade de segurança.

Estrutura da Tabela de Banco de Dados

A implementação padrão usa cinco tabelas de banco de dados, conforme listado abaixo. Em uma aplicação típica, as tabelas são ordenadas da com menos linhas para a com mais linhas:

- *acl_security_identities*: Esta tabela registra todas as identidades de segurança (SID) que detêm ACEs. A implementação padrão vem com duas identidades de segurança: `RoleSecurityIdentity` e `UserSecurityIdentity`
- *acl_classes*: Esta classe mapeia nomes de classes para um ID único, que pode ser referenciado a partir de outras tabelas.
- *acl_object_identities*: Cada linha nessa tabela representa um único domínio de instância de objeto.
- *acl_object_identity_ancestors*: Esta tabela permite que todos os ancestrais de uma ACL, possam ser determinados de uma maneira muito eficiente.
- *acl_entries*: Esta tabela contém todas as ACEs. Essa é tipicamente a tabela com o maior número de linhas. Ela pode conter dezenas de milhões sem afetar significativamente o desempenho.

Escopo das Entradas de Controle de Acesso

Entradas de controle de acesso podem ter escopos diferentes nos quais se aplicam. No Symfony2, existem basicamente dois escopos diferentes:

- Class-Scope: Essas entradas se aplicam a todos os objetos com a mesma classe.
- Object-Scope: Este foi o escopo utilizado unicamente no capítulo anterior, e ele só se aplica a um objeto específico.

Às vezes, você vai encontrar a necessidade de aplicar uma ACE apenas a um campo específico do objeto. Vamos dizer que você quer que o ID somente seja visualizado por um administrador, mas não por seu serviço do cliente. Para resolver este problema comum, mais dois sub-escopos foram adicionados:

- Classe-Field-Scope: Essas entradas se aplicam a todos os objetos com a mesma classe, mas apenas a um campo específico dos objetos.
- Object-Field-Scope: Essas entradas se aplicam a um objeto específico, e apenas a um campo específico do objeto.

Decisões de Pré-Autorização

Para as decisões de pré-autorização, que são as decisões tomadas antes de qualquer método seguro (ou ação segura) ser invocado, o serviço AccessDecisionManager é usado. O AccessDecisionManager também é usado para tomar decisões de autorização com base em papéis. Assim como os papéis, o sistema de ACL adiciona vários atributos novos que podem ser utilizados para verificar se há permissões diferentes.

Mapa de Permissão Integrado

Atributo	Significado Pretendido	Bitmasks I
VIEW	Se alguém tem permissão para ver o objeto de domínio.	VIEW, EDI OPERATOI ou OWNER
EDIT	Se alguém tem permissão para fazer alterações no objeto de domínio.	EDIT, OPE MASTER o
CREATE	Se alguém tem permissão para criar o objeto de domínio.	CREATE, C MASTER o
DELETE	Se alguém tem permissão excluir o objeto de domínio.	DELETE, C MASTER o
UNDELETE	Se alguém tem permissão para restaurar o objeto de domínio anteriormente excluído.	UNDELETI OPERATOI ou OWNER
OPERATOR	Se alguém tem permissão para executar todas as ações acima.	OPERATOI ou OWNER
MASTER	Se alguém tem permissão para executar todas as ações acima, e além disso é autorizada a conceder qualuqer uma das permissões acima para outros.	MASTER o
OWNER	Se alguém é dono do objeto de domínio. Um dono pode executar qualquer uma das ações acima *e* conceder permissões master e owner	OWNER

Atributos de Permissão vs Bitmasks de Permissão Os atributos são usados pelo AccessDecisionManager, assim como papéis. Muitas vezes, estes atributos representam, de fato, um agregado de bitmasks inteiros. Bitmasks inteiros por outro lado, são utilizados internamente pelo sistema de ACL para armazenar de forma eficiente suas permissões de usuário no banco de dados e executar verificações de acesso usando operações bitmask extremamente rápidas.

Extensibilidade O mapa de permissão acima não é de forma alguma estático, e teoricamente poderia ser completamente substituído. No entanto, ele deve cobrir a maioria dos problemas que você encontra e para a interoperabilidade com outros bundles, você é incentivado a manter o significado previsto por eles.

Decisões Pós-Autorização

Decisões pós-autorização são feitas depois de um método seguro ter sido invocado, e normalmente envolvem o objeto de domínio que é retornado por esse método. Após a invocação de providers, também é permitido modificar ou filtrar o objeto de domínio antes de ser devolvido.

Devido às limitações atuais da linguagem PHP, não existem recursos de pós-autorização integrados no componente de segurança. No entanto, existe um JMSSecurityExtraBundle experimental que adiciona esses recursos. Consulte a documentação para obter mais informações sobre a forma como isso é feito.

Processo para Tomar Decisões de Autorização

A classe ACL fornece dois métodos para determinar se uma identidade de segurança tem as bitmasks necessárias, `isGranted` e `isFieldGranted`. Quando a ACL recebe um pedido de autorização através de um desses métodos, ela delega esse pedido para uma implementação de PermissionGrantingStrategy. Isso permite a substituição da forma como as decisões de acesso são tomadas sem modificar a própria classe ACL.

O PermissionGrantingStrategy primeiro verifica todos os seus ACEs object-scope, se nenhum for aplicável, as ACEs class-scope serão verificadas, se nenhuma for aplicável, em seguida, o processo vai ser repetido com as ACEs da ACL pai. Se não existe nenhuma ACL pai, uma exceção será lançada.

Como forçar HTTPS ou HTTP para URLs Diferentes

Através da configuração de segurança, você pode forçar que áreas do seu site usem o protocolo `HTTPS`. Isso é feito através das regras `access_control` usando a opção `requires_channel`. Por exemplo, se você quiser forçar que todas as URLs que começam com `/secure` utilizem `HTTPS` então você poderia usar a seguinte configuração:

- *YAML*

```
access_control:
    - path: ^/secure
      roles: ROLE_ADMIN
      requires_channel: https
```

- *XML*

```
<access-control>
    <rule path="^/secure" role="ROLE_ADMIN" requires_channel="https" />
</access-control>
```

- *PHP*

```
'access_control' => array(
    array(
        'path'             => '^/secure',
        'role'             => 'ROLE_ADMIN',
        'requires_channel' => 'https',
    ),
),
```

O próprio formulário de login precisa permitir acesso anônimo, caso contrário, os usuários não seriam capazes de se autenticar. Para forçá-lo a usar `HTTPS` você pode ainda usar regras `access_control` com o papel `IS_AUTHENTICATED_ANONYMOUSLY`:

- *YAML*

```
access_control:
    - path: ^/login
      roles: IS_AUTHENTICATED_ANONYMOUSLY
      requires_channel: https
```

- *XML*

```
<access-control>
    <rule path="^/login"
          role="IS_AUTHENTICATED_ANONYMOUSLY"
          requires_channel="https" />
</access-control>
```

- *PHP*

```
'access_control' => array(
    array(
        'path'             => '^/login',
        'role'             => 'IS_AUTHENTICATED_ANONYMOUSLY',
        'requires_channel' => 'https',
    ),
),
```

Também é possível especificar o uso de `HTTPS` na configuração de roteamento. Veja Como forçar as rotas a usar sempre HTTPS ou HTTP para mais detalhes.

Como personalizar o seu Formulário de Login

Usar um *formulário de login* para autenticação é é um método comum e flexível para lidar com a autenticação no Symfony2. Praticamente todos os aspectos do formulário de login podem ser personalizados. A configuração padrão completa é mostrada na próxima seção.

Configuração de Referência do Formulário de Login

- *YAML*

```
# app/config/security.yml
security:
    firewalls:
        main:
            form_login:
                # the user is redirected here when he/she needs to login
                login_path:                     /login

                # if true, forward the user to the login form instead of redirecting
                use_forward:                    false

                # submit the login form here
                check_path:                     /login_check

                # by default, the login form *must* be a POST, not a GET
                post_only:                      true

                # login success redirecting options (read further below)
                always_use_default_target_path: false
                default_target_path:            /
                target_path_parameter:          _target_path
                use_referer:                    false

                # login failure redirecting options (read further below)
                failure_path:                   null
                failure_forward:                false

                # field names for the username and password fields
                username_parameter:             _username
                password_parameter:             _password

                # csrf token options
                csrf_parameter:                 _csrf_token
                intention:                      authenticate
```

- *XML*

```xml
<!-- app/config/security.xml -->
<config>
    <firewall>
        <form-login
            check_path="/login_check"
            login_path="/login"
            use_forward="false"
            always_use_default_target_path="false"
            default_target_path="/"
            target_path_parameter="_target_path"
            use_referer="false"
            failure_path="null"
            failure_forward="false"
            username_parameter="_username"
            password_parameter="_password"
            csrf_parameter="_csrf_token"
            intention="authenticate"
            post_only="true"
        />
    </firewall>
</config>
```

- *PHP*

```php
// app/config/security.php
$container->loadFromExtension('security', array(
    'firewalls' => array(
        'main' => array('form_login' => array(
            'check_path'                     => '/login_check',
            'login_path'                     => '/login',
            'user_forward'                   => false,
            'always_use_default_target_path' => false,
            'default_target_path'            => '/',
            'target_path_parameter'          => _target_path,
            'use_referer'                    => false,
            'failure_path'                   => null,
            'failure_forward'                => false,
            'username_parameter'             => '_username',
            'password_parameter'             => '_password',
            'csrf_parameter'                 => '_csrf_token',
            'intention'                      => 'authenticate',
            'post_only'                      => true,
        )),
    ),
));
```

Redirecionando após Sucesso

Você pode alterar o local para onde o formulário de login redireciona após um login bem-sucedido usando opções de configuração diferentes. Por padrão, o formulário irá redirecionar para a URL solicitada pelo usuário (ou seja, a URL que acionou o formulário de login que está sendo exibido). Por exemplo, se o usuário solicitou `http://www.example.com/admin/post/18/edit`, então, após o login bem-sucedido, ele será enviado de volta para `http://www.example.com/admin/post/18/edit` . Isto é feito através do armazenamento da URL solicitada em sessão. Se nenhuma URL estiver presente na sessão (talvez o usuário acessou diretamente a página de login), então, o usuário é redirecionado para a página padrão, que é `/` (ou seja, a homepage). Você pode alterar esse

comportamento de várias formas.

Nota: Como mencionado, por padrão, o usuário é redirecionado para a página que ele solicitou originalmente. Às vezes, isso pode causar problemas, como, por exemplo, uma requisição AJAX em background "aparece" sendo a última URL visitada, fazendo com que o usuário seja redirecionado para lá. Para informações sobre como controlar esse comportamento, consulte `/cookbook/security/target_path`.

Alterando a Página Padrão Primeiro, a página padrão pode ser definida (ou seja, a página para a qual o usuário será redirecionado se nenhuma página anterior foi armazenada em sessão). Para configurá-la para `/admin` use a seguinte configuração:

- *YAML*

```yaml
# app/config/security.yml
security:
    firewalls:
        main:
            form_login:
                # ...
                default_target_path: /admin
```

- *XML*

```xml
<!-- app/config/security.xml -->
<config>
    <firewall>
        <form-login
            default_target_path="/admin"
        />
    </firewall>
</config>
```

- *PHP*

```php
// app/config/security.php
$container->loadFromExtension('security', array(
    'firewalls' => array(
        'main' => array('form_login' => array(
            ...,
            'default_target_path' => '/admin',
        )),
    ),
));
```

Agora, quando não houver uma URL definida na sessão, os usuários serão enviados para `/admin`.

Sempre Redirecionar para a Página Padrão Você pode fazer com que os usuários sejam sempre redirecionados para a página padrão, independentemente da URL que eles tenham solicitado anteriormente, definindo a a opção `always_use_default_target_path` para true:

- *YAML*

```yaml
# app/config/security.yml
security:
    firewalls:
        main:
```

```
            form_login:
                # ...
                always_use_default_target_path: true
```

- *XML*

```
<!-- app/config/security.xml -->
<config>
    <firewall>
        <form-login
            always_use_default_target_path="true"
        />
    </firewall>
</config>
```

- *PHP*

```
// app/config/security.php
$container->loadFromExtension('security', array(
    'firewalls' => array(
        'main' => array('form_login' => array(
            ...,
            'always_use_default_target_path' => true,
        )),
    ),
));
```

Usando a URL de referência (Referring URL) No caso de nenhuma URL anterior estar armazenada em sessão, você pode desejar usar o `HTTP_REFERER` no lugar, pois este, muitas vezes, é o mesmo. Você pode fazer isso configurando o `use_referer` para true (o padrão é false):

- *YAML*

```
# app/config/security.yml
security:
    firewalls:
        main:
            form_login:
                # ...
                use_referer:        true
```

- *XML*

```
<!-- app/config/security.xml -->
<config>
    <firewall>
        <form-login
            use_referer="true"
        />
    </firewall>
</config>
```

- *PHP*

```
// app/config/security.php
$container->loadFromExtension('security', array(
    'firewalls' => array(
        'main' => array('form_login' => array(
            ...,
```

```
                'use_referer' => true,
            )),
        ),
    ));
```

Novo na versão 2.1: A partir da versão 2.1, se o referer for igual à opção `login_path`, o usuário será redirecionado para `default_target_path`.

Controlando a URL de redirecionamento dentro do Formulário Você também pode sobrescrever para onde o usuário será redirecionado, através do próprio formulário incluindo um campo oculto com o nome `_target_path`. Por exemplo, para redirecionar para a URL definida por rota `account`, use o seguinte:

- *Twig*

```
{# src/Acme/SecurityBundle/Resources/views/Security/login.html.twig #}
{% if error %}
    <div>{{ error.message }}</div>
{% endif %}

<form action="{{ path('login_check') }}" method="post">
    <label for="username">Username:</label>
    <input type="text" id="username" name="_username" value="{{ last_username }}" />

    <label for="password">Password:</label>
    <input type="password" id="password" name="_password" />

    <input type="hidden" name="_target_path" value="account" />

    <input type="submit" name="login" />
</form>
```

- *PHP*

```
<!-- src/Acme/SecurityBundle/Resources/views/Security/login.html.php -->
<?php if ($error): ?>
    <div><?php echo $error->getMessage() ?></div>
<?php endif; ?>

<form action="<?php echo $view['router']->generate('login_check') ?>" method="post">
    <label for="username">Username:</label>
    <input type="text" id="username" name="_username" value="<?php echo $last_username ?>" />

    <label for="password">Password:</label>
    <input type="password" id="password" name="_password" />

    <input type="hidden" name="_target_path" value="account" />

    <input type="submit" name="login" />
</form>
```

Agora, o usuário será redirecionado para o valor do campo oculto do formulário. O valor do atributo pode ser um caminho relativo, uma URL absoluta, ou um nome de rota. Você pode até mesmo alterar o nome do campo oculto do formulário, alterando a opção `target_path_parameter` para um outro valor.

- *YAML*

```
# app/config/security.yml
security:
    firewalls:
```

```
            main:
                form_login:
                    target_path_parameter: redirect_url
```

- *XML*

```
<!-- app/config/security.xml -->
<config>
    <firewall>
        <form-login
            target_path_parameter="redirect_url"
        />
    </firewall>
</config>
```

- *PHP*

```
// app/config/security.php
$container->loadFromExtension('security', array(
    'firewalls' => array(
        'main' => array('form_login' => array(
            'target_path_parameter' => redirect_url,
        )),
    ),
));
```

Redirecionando quando o Login Falhar Além de redirecionar o usuário após um login bem-sucedido, você também pode definir a URL que o usuário deve ser redirecionado após uma falha de login (por exemplo, quando um nome de usuário ou senha inválida foi submetida). Por padrão, o usuário é redirecionado de volta ao formulário de login. Você pode definir isso para uma URL diferente com a seguinte configuração:

- *YAML*

```
# app/config/security.yml
security:
    firewalls:
        main:
            form_login:
                # ...
                failure_path: /login_failure
```

- *XML*

```
<!-- app/config/security.xml -->
<config>
    <firewall>
        <form-login
            failure_path="login_failure"
        />
    </firewall>
</config>
```

- *PHP*

```
// app/config/security.php
$container->loadFromExtension('security', array(
    'firewalls' => array(
        'main' => array('form_login' => array(
            ...,
```

```
                'failure_path' => login_failure,
            )),
        ),
    ));
```

Como criar um Provider de Usuário Personalizado

Parte do processo de autenticação padrão do Symfony depende de "providers de usuário". Quando um usuário submete um nome de usuário e senha, a camada de autenticação solicita ao provider de usuário configurado para retornar um objeto de usuário para um determinado nome de usuário. Em seguida, o Symfony verifica se a senha deste usuário está correta e gera um token de segurança para que o usuário permaneça autenticado durante a sessão atual. O Symfony vem com os providers de usuário "in_memory" e "entity" prontos para uso. Neste artigo, você vai aprender como criar o seu próprio provider de usuário, que pode ser útil se os usuários são acessados através de um banco de dados personalizado, um arquivo ou - como mostrado neste exemplo - um serviço web.

Crie uma Classe de Usuário

Em primeiro lugar, independentemente de *onde* os seus dados de usuário estão vindo, você vai precisar criar uma classe `User` que representa esses dados. No entando, a classe `User` pode parecer com o que você quiser e conter quaisquer dados. O único requisito é que a classe implemente `UserInterface`. Os métodos dessa interface devem ser definidos na classe de usuário personalizada: `getRoles()`, `getPassword()`, `getSalt()`, `getUsername()`, `eraseCredentials()`. Também pode ser útil implementar a interface `EquatableInterface`, a qual define um método para verificar se o usuário é igual ao usuário atual. Essa interface requer um método `isEqualTo()` .

Vamos ver isso na prática:

```
// src/Acme/WebserviceUserBundle/Security/User/WebserviceUser.php
namespace Acme\WebserviceUserBundle\Security\User;

use Symfony\Component\Security\Core\User\UserInterface;
use Symfony\Component\Security\Core\User\EquatableInterface;

class WebserviceUser implements UserInterface, EquatableInterface
{
    private $username;
    private $password;
    private $salt;
    private $roles;

    public function __construct($username, $password, $salt, array $roles)
    {
        $this->username = $username;
        $this->password = $password;
        $this->salt = $salt;
        $this->roles = $roles;
    }

    public function getRoles()
    {
        return $this->roles;
    }

    public function getPassword()
    {
        return $this->password;
```

```
    }

    public function getSalt()
    {
        return $this->salt;
    }

    public function getUsername()
    {
        return $this->username;
    }

    public function eraseCredentials()
    {
    }

    public function isEqualTo(UserInterface $user)
    {
        if (!$user instanceof WebserviceUser) {
            return false;
        }

        if ($this->password !== $user->getPassword()) {
            return false;
        }

        if ($this->getSalt() !== $user->getSalt()) {
            return false;
        }

        if ($this->username !== $user->getUsername()) {
            return false;
        }

        return true;
    }
}
```

Se você tiver mais informações sobre seus usuários - como um "primeiro nome" - então você pode adicionar um campo `firstName` para guardar esse dado.

Criar um Provider de Usuário

Agora que você tem uma classe `User`, você vai criar um provider de usuário, que irá pegar informações de usuário de algum serviço web, criar um objeto `WebserviceUser` e popular ele com os dados.

O provider de usuário é apenas uma classe PHP que deve implementar a `UserProviderInterface`, que requer a definição de três métodos: `loadUserByUsername($username)`, `refreshUser(UserInterface $user)` e `supportsClass($class)`. Para mais detalhes, consulte `UserProviderInterface`.

Aqui está um exemplo de como isso pode parecer:

```
// src/Acme/WebserviceUserBundle/Security/User/WebserviceUserProvider.php
namespace Acme\WebserviceUserBundle\Security\User;

use Symfony\Component\Security\Core\User\UserProviderInterface;
use Symfony\Component\Security\Core\User\UserInterface;
use Symfony\Component\Security\Core\Exception\UsernameNotFoundException;
```

```
use Symfony\Component\Security\Core\Exception\UnsupportedUserException;

class WebserviceUserProvider implements UserProviderInterface
{
    public function loadUserByUsername($username)
    {
        // make a call to your webservice here
        $userData = ...
        // pretend it returns an array on success, false if there is no user

        if ($userData) {
            $password = '...';

            // ...

            return new WebserviceUser($username, $password, $salt, $roles);
        }

        throw new UsernameNotFoundException(sprintf('Username "%s" does not exist.', $username));
    }

    public function refreshUser(UserInterface $user)
    {
        if (!$user instanceof WebserviceUser) {
            throw new UnsupportedUserException(sprintf('Instances of "%s" are not supported.', get_cl
        }

        return $this->loadUserByUsername($user->getUsername());
    }

    public function supportsClass($class)
    {
        return $class === 'Acme\WebserviceUserBundle\Security\User\WebserviceUser';
    }
}
```

Crie um Serviço para o Provider de Usuário

Agora você tornará o provider de usuário disponível como um serviço:

- *YAML*

```
# src/Acme/WebserviceUserBundle/Resources/config/services.yml
parameters:
    webservice_user_provider.class: Acme\WebserviceUserBundle\Security\User\WebserviceUserProvid

services:
    webservice_user_provider:
        class: "%webservice_user_provider.class%"
```

- *XML*

```
<!-- src/Acme/WebserviceUserBundle/Resources/config/services.xml -->
<parameters>
    <parameter key="webservice_user_provider.class">Acme\WebserviceUserBundle\Security\User\Webs
</parameters>
```

```xml
<services>
    <service id="webservice_user_provider" class="%webservice_user_provider.class%"></service>
</services>
```

- *PHP*

```php
// src/Acme/WebserviceUserBundle/Resources/config/services.php
use Symfony\Component\DependencyInjection\Definition;

$container->setParameter('webservice_user_provider.class', 'Acme\WebserviceUserBundle\Security\U

$container->setDefinition('webservice_user_provider', new Definition('%webservice_user_provider.
```

Dica: A verdadeira implementação do provider de usuário provavelmente terá algumas dependências, opções de configuração ou outros serviços. Adicione eles como argumentos na definição de serviço.

Nota: Certifique-se que o arquivo de serviços está sendo importado. Veja *Importando configuração com imports* para mais detalhes.

Modifique o `security.yml`

Tudo será combinado em sua configuração de segurança. Adicione o provider de usuário na lista de providers na seção "security". Escolha um nome para o provider de usuário (por exemplo, "webservice") e mencione o id do serviço que você acabou de definir.

- *YAML*

```yaml
// app/config/security.yml
security:
    providers:
        webservice:
            id: webservice_user_provider
```

- *XML*

```xml
<!-- app/config/security.xml -->
<config>
    <provider name="webservice" id="webservice_user_provider" />
</config>
```

- *PHP*

```php
// app/config/security.php
$container->loadFromExtension('security', array(
    'providers' => array(
        'webservice' => array(
            'id' => 'webservice_user_provider',
        ),
    ),
));
```

O Symfony também precisa saber como codificar as senhas que são fornecidas no site pelos usuários, por exemplo, através do preenchimento de um formulário de login. Você pode fazer isso adicionando uma linha na seção "encoders" da sua configuração de segurança:

- *YAML*

```yaml
# app/config/security.yml
security:
    encoders:
        Acme\WebserviceUserBundle\Security\User\WebserviceUser: sha512
```

- *XML*

```xml
<!-- app/config/security.xml -->
<config>
    <encoder class="Acme\WebserviceUserBundle\Security\User\WebserviceUser">sha512</encoder>
</config>
```

- *PHP*

```php
// app/config/security.php
$container->loadFromExtension('security', array(
    'encoders' => array(
        'Acme\WebserviceUserBundle\Security\User\WebserviceUser' => 'sha512',
    ),
));
```

O valor aqui deve corresponder porém com as senhas que foram originalmente codificadas ao criar os seus usuários (no entanto os usuários foram criados). Quando um usuário submete a sua senha, o salt é acrescentado ao valor da senha e então são codificados usando este algoritmo antes de ser comparada com o hash da senha retornado pelo seu método `getPassword()`. Além disso, dependendo das suas opções, a senha pode ser codificada várias vezes e codificada para base64.

Detalhes sobre como as senhas são codificadas

O Symfony utiliza um método específico para combinar o salt e codificar a senha antes de compará-la com a senha codificada. Se o `getSalt()` não retornar nada, então a senha submetida é simplesmente codificada utilizando o algoritmo que você especificou no `security.yml`. Se um salt *é* especificado, então o valor seguinte é criado e *então* feito o hash através do algoritmo:

```
$password.'{'.$salt.'}';``
```

Caso os usuários externos tenham nas suas senhas um salt através de um método diferente, então você terá um pouco mais de trabalho para que o Symfony codifique corretamente a senha. Isso está além do escopo deste artigo, mas incluiria estender a classe `MessageDigestPasswordEncoder` e sobrescrever o método `mergePasswordAndSalt`.

Além disso, o hash, por padrão, é codificado várias vezes e codificado para base64. Para obter detalhes específicos, consulte MessageDigestPasswordEncoder. Para evitar isso, configure ele no seu arquivo de configuração:

- *YAML*

```yaml
# app/config/security.yml
security:
    encoders:
        Acme\WebserviceUserBundle\Security\User\WebserviceUser:
            algorithm: sha512
            encode_as_base64: false
            iterations: 1
```

- *XML*

```xml
<!-- app/config/security.xml -->
<config>
    <encoder class="Acme\WebserviceUserBundle\Security\User\WebserviceUser"
        algorithm="sha512"
        encode-as-base64="false"
        iterations="1"
    />
</config>
```

- *PHP*

```php
// app/config/security.php
$container->loadFromExtension('security', array(
    'encoders' => array(
        'Acme\WebserviceUserBundle\Security\User\WebserviceUser' => array(
            'algorithm'         => 'sha512',
            'encode_as_base64'  => false,
            'iterations'        => 1,
        ),
    ),
));
```

3.1.12 Cache

Como usar Varnish para aumentar a velocidade do meu Website

Pelo cache do Symfony2 usar os cache headers padrões do HTTP, o *Proxy Reverso do Symfony2* pode facilmente se substituido por qualquer por qualquer outro proxy reverso. Varnish é um poderoso, open-source, HTTP accelerator capaz de servir conteúdo em cache de forma rápida em incluindo suporte à *Edge Side Includes* (ESI).

Configuração

Como visto anteriormente, Symfony2 é esperto o bastente para detectar se fala com um proxy reverso que compreende ESI ou não. Ele trabalha fora da caixa quando você usa o proxy reverso do Symfony2, mas você precisa de um configuração especial para poder trabalhar com Varnish. Felizmente, Symfony2 depende ainda de outro padrão escrito por Akamaï (Edge Architecture), então as dicas de configuração desde capítulo podem ser úteis mesmo se você não usar Symfony2.

Nota: Varnish suporta apenas o atributo `src` para tags ESI (atributos `onerror` e `alt` são ignorados).

Primeiro, configure o Varnish para que ele informe o suporte à ESI adicionando um `Surrogate-Capability` ao cabeçalho nas requisições enviadas ao servidor de aplicação:

```
sub vcl_recv {
    set req.http.Surrogate-Capability = "abc=ESI/1.0";
}
```

Após isso, otimize o Varnish para que ele apenas analise o conteúdo da resposta quando existir ao menos uma tag ESI, verificando o cabeçalho `Surrogate-Control` que é adicionado automaticamente pelo Symfony2.

```
sub vcl_fetch {
    if (beresp.http.Surrogate-Control ~ "ESI/1.0") {
        unset beresp.http.Surrogate-Control;

        // para Varnish >= 3.0
        set beresp.do_esi = true;
        // para Varnish < 3.0
        // esi;
    }
}
```

Cuidado: Compressão com ESI não é suportada pelo Varnish até a versão 3.0 (leia GZIP and Varnish). Se você não está utilizando Varnish 3.0, coloque um servidor web a frente do Varnish para executar a compressão.

Invalidação do Cache

Você nunca deverá precisar invalidar os dados em cache porque a invalidação é colocada nativamente nos modelos de cache HTTP (veja *Invalidação do Cache*).

Ainda assim, o Varnish pode ser configurado para aceitar um método HTTP `PURGE` especial que invalidará o cache para derterminado recurso:

```
sub vcl_hit {
    if (req.request == "PURGE") {
        set obj.ttl = 0s;
        error 200 "Purgado";
    }
}

sub vcl_miss {
    if (req.request == "PURGE") {
        error 404 "Não Purgado";
    }
}
```

Cuidado: Você deve proteger o método HTTP‘‘PURGE‘‘ para evitar que qualquer pessoa possa purgar os dados em cache.

3.1.13 Templating

Utilizando variáveis em todas templates (Variáveis globais)

Algumas vezes você quer que uma variável esteja disponível em todas as templates que utiliza. Isto é possível configurando o twig dentro do arquivo `app/config/config.yml` :

```
# app/config/config.yml
twig:
    # ...
    globals:
        ga_tracking: UA-xxxxx-x
```

Agora a variável `ga_tracking` está disponível em todas templates Twig e pode ser acessada da seguinte forma.

```
<p>Our google tracking code is: {{ ga_tracking }} </p>
```

É fácil! Você também pode utilizar do sistema de parâmetros (*Parâmetros do Serviço*), que permite você isolar e reutilizar o valor como a seguir.

```
; app/config/parameters.yml
[parameters]
    ga_tracking: UA-xxxxx-x
```

```
# app/config/config.yml
twig:
    globals:
        ga_tracking: %ga_tracking%
```

A mesma variável está disponível exatamente como antes.

Variáveis globais mais complexas

Se a variável global que deseja definir é mais complexa, como um objeto por exemplo, então você não poderá utilizar o método acima. Ao invés disso, precisa criar uma extensão Twig (Twig Extension) e retornar a variável global como uma das entradas no método `getGlobals`.

3.1.14 Log

Como usar o Monolog para escrever Logs

O Monolog é uma biblioteca de log para o PHP 5.3 usada pelo Symfony2. É inspirado pela biblioteca LogBook do Python.

Utilização

Para fazer o log de uma mensagem, basta obter o serviço logger a partir do container em seu controlador:

```
public function indexAction()
{
    $logger = $this->get('logger');
    $logger->info('I just got the logger');
    $logger->err('An error occurred');

    // ...
}
```

O serviço `logger` possui métodos diferentes para os níveis de log. Para mais detalhes sobre os métodos que estão disponíveis, veja `LoggerInterface`.

Manipuladores (handlers) e Canais (channels): Escrevendo logs em locais diferentes

No Monolog, cada logger define um canal de log, que organiza as suas mensagens de log em diferentes "categorias". Então, cada canal tem uma pilha de manipuladores para escrever os logs (os manipuladores podem ser compartilhados).

Dica: Ao injetar o logger em um serviço você pode usar um canal personalizado para controlar que "canal" o logger irá realizar o log.

O manipulador básico é o `StreamHandler`, que grava logs em um stream (por padrão em `app/logs/prod.log` no ambiente prod e `app/logs/dev.log` no ambiente dev).

O Monolog também vem com um poderoso manipulador integrado para realizar log no ambiente prod: `FingersCrossedHandler`. Ele permite que você armazene as mensagens em um buffer e registre o log somente se a mensagem alcança o nível de ação (ERROR na configuração fornecida na edição standard), enviando as mensagens para outro manipulador.

Usando vários manipuladores O logger usa uma pilha de manipuladores que são chamados sucessivamente. Isto permite registrar facilmente as mensagens de várias maneiras.

- *YAML*

```
# app/config/config.yml
monolog:
    handlers:
        applog:
            type: stream
            path: /var/log/symfony.log
            level: error
        main:
            type: fingers_crossed
            action_level: warning
            handler: file
        file:
            type: stream
            level: debug
        syslog:
            type: syslog
            level: error
```

- *XML*

```
<!-- app/config/config.xml -->
<container xmlns="http://symfony.com/schema/dic/services"
```

```
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:monolog="http://symfony.com/schema/dic/monolog"
        xsi:schemaLocation="http://symfony.com/schema/dic/services http://symfony.com/schema/dic/ser
                            http://symfony.com/schema/dic/monolog http://symfony.com/schema/dic/monc

        <monolog:config>
            <monolog:handler
                name="applog"
                type="stream"
                path="/var/log/symfony.log"
                level="error"
            />
            <monolog:handler
                name="main"
                type="fingers_crossed"
                action-level="warning"
                handler="file"
            />
            <monolog:handler
                name="file"
                type="stream"
                level="debug"
            />
            <monolog:handler
                name="syslog"
                type="syslog"
                level="error"
            />
        </monolog:config>
    </container>
```

- *PHP*

```
    // app/config/config.php
    $container->loadFromExtension('monolog', array(
        'handlers' => array(
            'applog' => array(
                'type'  => 'stream',
                'path'  => '/var/log/symfony.log',
                'level' => 'error',
            ),
            'main' => array(
                'type'         => 'fingers_crossed',
                'action_level' => 'warning',
                'handler'      => 'file',
            ),
            'file' => array(
                'type'  => 'stream',
                'level' => 'debug',
            ),
            'syslog' => array(
                'type'  => 'syslog',
                'level' => 'error',
            ),
        ),
    ));
```

A configuração acima define uma pilha de manipuladores que será chamada na ordem em que eles são definidos.

Dica: O manipulador chamado "file" não será incluído na própria pilha, pois, ele é usado como um manipulador aninhado do manipulador `fingers_crossed`.

Nota: Se você quiser alterar a configuração do MonologBundle em outro arquivo de configuração, você precisa redefinir toda a pilha. Não pode ser feito o merge porque a ordem é importante e um merge não permite controlar a ordem.

Alterando o formatador O manipulador usa um `Formatter` para formatar o registro antes de efetuar o log. Todos os manipuladores do Monolog usam uma instância do `Monolog\Formatter\LineFormatter` por padrão, mas você pode substituí-lo facilmente. O formatador deve implementar `Monolog\Formatter\FormatterInterface`.

- *YAML*

```
# app/config/config.yml
services:
    my_formatter:
        class: Monolog\Formatter\JsonFormatter
monolog:
    handlers:
        file:
            type: stream
            level: debug
            formatter: my_formatter
```

- *XML*

```
<!-- app/config/config.xml -->
<container xmlns="http://symfony.com/schema/dic/services"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:monolog="http://symfony.com/schema/dic/monolog"
    xsi:schemaLocation="http://symfony.com/schema/dic/services http://symfony.com/schema/dic/ser
                        http://symfony.com/schema/dic/monolog http://symfony.com/schema/dic/monc

    <services>
        <service id="my_formatter" class="Monolog\Formatter\JsonFormatter" />
    </services>

    <monolog:config>
        <monolog:handler
            name="file"
            type="stream"
            level="debug"
            formatter="my_formatter"
        />
    </monolog:config>
</container>
```

- *PHP*

```
// app/config/config.php
$container
    ->register('my_formatter', 'Monolog\Formatter\JsonFormatter');
```

```
    $container->loadFromExtension('monolog', array(
        'handlers' => array(
            'file' => array(
                'type'      => 'stream',
                'level'     => 'debug',
                'formatter' => 'my_formatter',
            ),
        ),
    ));
```

Adicionando alguns dados extras nas mensagens de log

O Monolog permite processar o registro antes de fazer o log para adicionar alguns dados extras. Um processador pode ser aplicado para toda a pilha do manipulador ou somente para um manipulador específico.

Um processador é simplesmente um callable recebendo o registro como seu primeiro argumento.

Os processadores são configurados usando a tag DIC `monolog.processor`. Veja a referencia sobre ela.

Adicionando uma Sessão/Token de Pedido Às vezes, é difícil dizer que entradas no log pertencem a qual sessão e/ou pedido. O exemplo a seguir irá adicionar um token exclusivo para cada pedido utilizando um processador.

```
namespace Acme\MyBundle;

use Symfony\Component\HttpFoundation\Session\Session;

class SessionRequestProcessor
{
    private $session;
    private $token;

    public function __construct(Session $session)
    {
        $this->session = $session;
    }

    public function processRecord(array $record)
    {
        if (null === $this->token) {
            try {
                $this->token = substr($this->session->getId(), 0, 8);
            } catch (\RuntimeException $e) {
                $this->token = '????????';
            }
            $this->token .= '-' . substr(uniqid(), -8);
        }
        $record['extra']['token'] = $this->token;

        return $record;
    }
}
```

- *YAML*

```
    # app/config/config.yml
    services:
        monolog.formatter.session_request:
```

```
            class: Monolog\Formatter\LineFormatter
            arguments:
                - "[%%datetime%%] [%%extra.token%%] %%channel%%.%%level_name%%: %%message%%\n"

        monolog.processor.session_request:
            class: Acme\MyBundle\SessionRequestProcessor
            arguments:  ["@session"]
            tags:
                - { name: monolog.processor, method: processRecord }

    monolog:
        handlers:
            main:
                type: stream
                path: "%kernel.logs_dir%/%kernel.environment%.log"
                level: debug
                formatter: monolog.formatter.session_request
```

- *XML*

```
    <container xmlns="http://symfony.com/schema/dic/services"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:monolog="http://symfony.com/schema/dic/monolog"
        xsi:schemaLocation="http://symfony.com/schema/dic/services http://symfony.com/schema/dic/ser
                            http://symfony.com/schema/dic/monolog http://symfony.com/schema/dic/monc

        <services>
            <service id="monolog.formatter.session_request" class="Monolog\Formatter\LineFormatter">
                <argument>[%%datetime%%] [%%extra.token%%] %%channel%%.%%level_name%%: %%message%%&#
            </service>

            <service id="monolog.processor.session_request" class="Acme\MyBundle\SessionRequestProce
                <argument type="service" id="session" />
                <tag name="monolog.processor" method="processRecord" />
            </service>
        </services>

        <monolog:config>
            <monolog:handler
                name="main"
                type="stream"
                path="%kernel.logs_dir%/%kernel.environment%.log"
                level="debug"
                formatter="monolog.formatter.session_request"
            />
        </monolog:config>
    </container>
```

- *PHP*

```
    // app/config/config.php
    $container
        ->register('monolog.formatter.session_request', 'Monolog\Formatter\LineFormatter')
        ->addArgument('[%%datetime%%] [%%extra.token%%] %%channel%%.%%level_name%%: %%message%%\n');

    $container
        ->register('monolog.processor.session_request', 'Acme\MyBundle\SessionRequestProcessor')
        ->addArgument(new Reference('session'))
        ->addTag('monolog.processor', array('method' => 'processRecord'));
```

```
    $container->loadFromExtension('monolog', array(
        'handlers' => array(
            'main' => array(
                'type'      => 'stream',
                'path'      => '%kernel.logs_dir%/%kernel.environment%.log',
                'level'     => 'debug',
                'formatter' => 'monolog.formatter.session_request',
            ),
        ),
    ));
```

Nota: Se você usa vários manipuladores, você também pode registrar o processador no nível do manipulador, em vez de globalmente.

Como configurar o Monolog para enviar erros por e-mail

O Monolog pode ser configurado para enviar um e-mail quando ocorrer um erro na aplicação. A configuração para isto requer alguns manipuladores aninhados a fim de evitar o recebimento de muitos e-mails. Esta configuração parece complicada no início, mas cada manipulador é bastante simples quando é separado.

- *YAML*

```
    # app/config/config_prod.yml
    monolog:
        handlers:
            mail:
                type:         fingers_crossed
                action_level: critical
                handler:      buffered
            buffered:
                type:    buffer
                handler: swift
            swift:
                type:       swift_mailer
                from_email: error@example.com
                to_email:   error@example.com
                subject:    An Error Occurred!
                level:      debug
```

- *XML*

```
    <!-- app/config/config_prod.xml -->
    <container xmlns="http://symfony.com/schema/dic/services"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:monolog="http://symfony.com/schema/dic/monolog"
        xsi:schemaLocation="http://symfony.com/schema/dic/services http://symfony.com/schema/dic/ser
                            http://symfony.com/schema/dic/monolog http://symfony.com/schema/dic/monc

        <monolog:config>
            <monolog:handler
                name="mail"
                type="fingers_crossed"
                action-level="critical"
                handler="buffered"
            />
```

```
            <monolog:handler
                name="buffered"
                type="buffer"
                handler="swift"
            />
            <monolog:handler
                name="swift"
                from-email="error@example.com"
                to-email="error@example.com"
                subject="An Error Occurred!"
                level="debug"
            />
        </monolog:config>
    </container>
```

- *PHP*

```
    // app/config/config_prod.php
    $container->loadFromExtension('monolog', array(
        'handlers' => array(
            'mail' => array(
                'type'         => 'fingers_crossed',
                'action_level' => 'critical',
                'handler'      => 'buffered',
            ),
            'buffered' => array(
                'type'    => 'buffer',
                'handler' => 'swift',
            ),
            'swift' => array(
                'type'       => 'swift_mailer',
                'from_email' => 'error@example.com',
                'to_email'   => 'error@example.com',
                'subject'    => 'An Error Occurred!',
                'level'      => 'debug',
            ),
        ),
    ));
```

O manipulador `mail` é um manipulador `fingers_crossed`, significando que ele é acionado apenas quando o nível de ação, neste caso, `critical` é alcançado. Em seguida, ele faz log de tudo, incluindo mensagens abaixo do nível de ação. O nível `critical` só é acionado para erros HTTP de código 5xx. A definição `handler` significa que a saída é, então, transferida para o manipulador `buffered`.

Dica: Se você deseja que tanto erros de nível 400 quanto de nível 500 acionem um e-mail, defina o `action_level` para `error` ao invés de `critical`.

O manipulador `buffered` simplesmente mantém todas as mensagens para um pedido e em seguida, passa elas para o manipulador aninhado de uma só vez. Se você não usar este manipulador, então cada mensagem será enviada separadamente. Em seguida, é passado para o manipulador `swift`. Este é o manipulador que efetivamente trata de enviar o erro para o e-mail. As configurações para isso são simples, os endereços de (to), para (from) e o assunto (subject).

Você pode combinar esses manipuladores com outros manipuladores de modo que os erros ainda serão registrados no servidor, bem como enviados os e-mails:

- *YAML*

```yaml
# app/config/config_prod.yml
monolog:
    handlers:
        main:
            type:         fingers_crossed
            action_level: critical
            handler:      grouped
        grouped:
            type:    group
            members: [streamed, buffered]
        streamed:
            type:  stream
            path:  "%kernel.logs_dir%/%kernel.environment%.log"
            level: debug
        buffered:
            type:    buffer
            handler: swift
        swift:
            type:       swift_mailer
            from_email: error@example.com
            to_email:   error@example.com
            subject:    An Error Occurred!
            level:      debug
```

- *XML*

```xml
<!-- app/config/config_prod.xml -->
<container xmlns="http://symfony.com/schema/dic/services"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:monolog="http://symfony.com/schema/dic/monolog"
    xsi:schemaLocation="http://symfony.com/schema/dic/services http://symfony.com/schema/dic/ser
                        http://symfony.com/schema/dic/monolog http://symfony.com/schema/dic/monc

    <monolog:config>
        <monolog:handler
            name="main"
            type="fingers_crossed"
            action_level="critical"
            handler="grouped"
        />
        <monolog:handler
            name="grouped"
            type="group"
        >
            <member type="stream"/>
            <member type="buffered"/>
        </monolog:handler>
        <monolog:handler
            name="stream"
            path="%kernel.logs_dir%/%kernel.environment%.log"
            level="debug"
        />
        <monolog:handler
            name="buffered"
            type="buffer"
            handler="swift"
        />
        <monolog:handler
```

```
                name="swift"
                from-email="error@example.com"
                to-email="error@example.com"
                subject="An Error Occurred!"
                level="debug"
            />
        </monolog:config>
    </container>
```

- *PHP*

```
    // app/config/config_prod.php
    $container->loadFromExtension('monolog', array(
        'handlers' => array(
            'main' => array(
                'type'         => 'fingers_crossed',
                'action_level' => 'critical',
                'handler'      => 'grouped',
            ),
            'grouped' => array(
                'type'    => 'group',
                'members' => array('streamed', 'buffered'),
            ),
            'streamed'  => array(
                'type'  => 'stream',
                'path'  => '%kernel.logs_dir%/%kernel.environment%.log',
                'level' => 'debug',
            ),
            'buffered'    => array(
                'type'    => 'buffer',
                'handler' => 'swift',
            ),
            'swift' => array(
                'type'       => 'swift_mailer',
                'from_email' => 'error@example.com',
                'to_email'   => 'error@example.com',
                'subject'    => 'An Error Occurred!',
                'level'      => 'debug',
            ),
        ),
    ));
```

Isto usa o manipulador `group` para enviar as mensagens para os dois membros do grupo, os manipuladores `buffered` e `stream`. As mensagens serão agora gravadas no arquivo de log e enviadas por e-mail.

Como fazer log de Mensagens em Arquivos Diferentes

Novo na versão 2.1: A capacidade de especificar os canais para um manipulador específico foi adicionada ao MonologBundle para o Symfony 2.1.

A Edição Standard do Symfony contém vários canais para log: `doctrine`, `event`, `security` e `request`. Cada canal corresponde a um serviço logger (`monolog.logger.XXX`) no container e, é injetado no serviço em questão. A finalidade dos canais é de serem capazes de organizar diferentes tipos de mensagens de log.

Por padrão, o Symfony2 realiza o log de todas as mensagens em um único arquivo (independentemente do canal).

Mudando um canal para um Handler diferente

Agora, suponha que você queira registrar o canal `doctrine` para um arquivo diferente.

Para isso, basta criar um novo manipulador (handler) e configurá-lo como segue:

- *YAML*

```
monolog:
    handlers:
        main:
            type: stream
            path: /var/log/symfony.log
            channels: !doctrine
        doctrine:
            type: stream
            path: /var/log/doctrine.log
            channels: doctrine
```

- *XML*

```
<monolog:config>
    <monolog:handlers>
        <monolog:handler name="main" type="stream" path="/var/log/symfony.log">
            <monolog:channels>
                <type>exclusive</type>
                <channel>doctrine</channel>
            </monolog:channels>
        </monolog:handler>

        <monolog:handler name="doctrine" type="stream" path="/var/log/doctrine.log" />
            <monolog:channels>
                <type>inclusive</type>
                <channel>doctrine</channel>
            </monolog:channels>
        </monolog:handler>
    </monolog:handlers>
</monolog:config>
```

Especificação Yaml

Você pode especificar a configuração de várias formas:

```
channels: ~    # Include all the channels

channels: foo  # Include only channel "foo"
channels: !foo # Include all channels, except "foo"

channels: [foo, bar]   # Include only channels "foo" and "bar"
channels: [!foo, !bar] # Include all channels, except "foo" and "bar"

channels:
    type:     inclusive # Include only those listed below
    elements: [ foo, bar ]
channels:
    type:     exclusive # Include all, except those listed below
    elements: [ foo, bar ]
```

Criando o seu próprio Canal

Você pode mudar o canal que monolog faz o log para um serviço de cada vez. Isto é feito adicionando a tag `monolog.logger` ao seu serviço e especificando qual canal o serviço deve fazer o log. Ao fazer isso, o logger que é injetado naquele serviço é pré-configurado para usar o canal que você especificou.

Para mais informações - incluindo um exemplo completo - leia "dic_tags-monolog" na seção de referência das Tags de Injeção de Dependência.

Aprenda mais no Cookbook

- Como usar o Monolog para escrever Logs

3.1.15 Console

Como criar um Comando de Console

A página Console na seção Componentes (O Componente Console) aborda como criar um comando de Console. Este artigo do cookbook abrange as diferenças ao criar comandos do Console com o framework Symfony2.

Registrando os Comandos Automaticamente

Para disponibilizar os comandos de console automaticamente com o Symfony2, adicione um diretório `Command` dentro de seu bundle e crie um arquivo php com o sufixo `Command.php` para cada comando que você deseja fornecer. Por exemplo, se você deseja estender o `AcmeDemoBundle` (disponível na Edição Standard do Symfony) para cumprimentar-nos a partir da linha de comando, crie o `GreetCommand.php` e adicione o seguinte código:

```
// src/Acme/DemoBundle/Command/GreetCommand.php
namespace Acme\DemoBundle\Command;

use Symfony\Bundle\FrameworkBundle\Command\ContainerAwareCommand;
use Symfony\Component\Console\Input\InputArgument;
use Symfony\Component\Console\Input\InputInterface;
use Symfony\Component\Console\Input\InputOption;
use Symfony\Component\Console\Output\OutputInterface;

class GreetCommand extends ContainerAwareCommand
{
    protected function configure()
    {
        $this
            ->setName('demo:greet')
            ->setDescription('Greet someone')
            ->addArgument('name', InputArgument::OPTIONAL, 'Who do you want to greet?')
            ->addOption('yell', null, InputOption::VALUE_NONE, 'If set, the task will yell in upperca
        ;
    }

    protected function execute(InputInterface $input, OutputInterface $output)
    {
        $name = $input->getArgument('name');
        if ($name) {
            $text = 'Hello '.$name;
        } else {
```

```
            $text = 'Hello';
        }

        if ($input->getOption('yell')) {
            $text = strtoupper($text);
        }

        $output->writeln($text);
    }
}
```

Este comando estará disponível automaticamente para executar:

```
$ app/console demo:greet Fabien
```

Testando Comandos

Ao testar os comandos utilizados como parte do framework completo `Application` deve ser usado em vez de `Application`:

```
use Symfony\Component\Console\Tester\CommandTester;
use Symfony\Bundle\FrameworkBundle\Console\Application;
use Acme\DemoBundle\Command\GreetCommand;

class ListCommandTest extends \PHPUnit_Framework_TestCase
{
    public function testExecute()
    {
        // mock the Kernel or create one depending on your needs
        $application = new Application($kernel);
        $application->add(new GreetCommand());

        $command = $application->find('demo:greet');
        $commandTester = new CommandTester($command);
        $commandTester->execute(array('command' => $command->getName()));

        $this->assertRegExp('/.../', $commandTester->getDisplay());

        // ...
    }
}
```

Obtendo Serviços do Container de Serviços

Usando `ContainerAwareCommand` como classe base para o comando (em vez do mais básico `Command`), você tem acesso ao container de serviço. Em outras palavras, você tem acesso a qualquer serviço configurado. Por exemplo, você pode facilmente estender a tarefa para a mesma ser traduzível:

```
protected function execute(InputInterface $input, OutputInterface $output)
{
    $name = $input->getArgument('name');
    $translator = $this->getContainer()->get('translator');
    if ($name) {
        $output->writeln($translator->trans('Hello %name%!', array('%name%' => $name)));
    } else {
```

```
        $output->writeln($translator->trans('Hello!'));
    }
}
```

Como usar o Console

A página `/components/console/usage` da documentação dos componentes aborda sobre as opções globais do console. Quando você usar o console como parte do framework full stack, algumas opções globais adicionais também estão disponíveis.

Por padrão, os comandos de console executam no ambiente `dev` e você pode desejar alterar isso para alguns comandos. Por exemplo, se você deseja executar alguns comandos no ambiente `prod` por motivos de desempenho. Além disso, o resultado de alguns comandos será diferente, dependendo do ambiente. Por exemplo, o comando `cache:clear` irá limpar e fazer o warm do cache apenas para o ambiente especificado. Para limpar e realizar o warm do cache de `prod` você precisa executar:

```
$ php app/console cache:clear --env=prod
```

ou o equivalente:

```
$ php app/console cache:clear -e=prod
```

Além de alterar o ambiente, você pode optar também por desativar o modo de depuração. Isto pode ser útil quando deseja-se executar comandos no ambiente `dev` mas evitar o impacto no desempenho devido a coleta de dados de depuração:

```
$ php app/console list --no-debug
```

Há um shell interativo que permite à você digitar os comandos sem ter que especificar `php app/console` toda vez, o que é útil se você precisa executar vários comandos. Para entrar no shell execute:

```
$ php app/console --shell
$ php app/console -s
```

Agora você pode simplesmente executar comandos com o nome do comando:

```
Symfony > list
```

Ao usar o shell você pode escolher em executar cada comando em um processo separado:

```
$ php app/console --shell --process-isolation
$ php app/console -s --process-isolation
```

Quando fizer isso, a saída não será colorida e a interatividade não é suportada, então, você vai precisar passar todos os parâmetros de comando explicitamente.

Nota: A menos que você estiver usando processos isolados, limpar o cache no shell não terá efeito sobre os comandos subsequentes que você executar. Isto é porque os arquivos originais em cache ainda estão sendo usados.

Como gerar URLs com um Host personalizado em Comandos de Console

Infelizmente, o contexto de linha de comando não sabe sobre o seu VirtualHost ou nome de domínio. Isto significa que se você gerar URLs absolutas em um comando de Console, você provavelmente vai acabar com algo como `http://localhost/foo/bar` o que não é muito útil.

Para corrigir isso, você precisa configurar o "contexto do pedido", que é uma maneira elegante de dizer que você precisa configurar o seu ambiente para que ele saiba qual URL ele deve usar ao gerar as URLs.

Há duas maneiras de configurar o contexto do pedido: a nível da aplicação e por Comando.

Configurando o Contexto do Pedido globalmente

Para configurar o Contexto do Pedido - que é usado pelo Gerador de URL - você pode redefinir os parâmetros que ele usa como valores padrão para alterar o host padrão (localhost) e o esquema (http). Note que isso não impacta nas URLs geradas através de solicitações normais da web, uma vez que substituirá os padrões.

- *YAML*

```
# app/config/parameters.yml
parameters:
    router.request_context.host: example.org
    router.request_context.scheme: https
```

- *XML*

```
<!-- app/config/parameters.xml -->
<?xml version="1.0" encoding="UTF-8"?>

<container xmlns="http://symfony.com/schema/dic/services"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

    <parameters>
        <parameter key="router.request_context.host">example.org</parameter>
        <parameter key="router.request_context.scheme">https</parameter>
    </parameters>
</container>
```

- *PHP*

```
// app/config/config_test.php
$container->setParameter('router.request_context.host', 'example.org');
$container->setParameter('router.request_context.scheme', 'https');
```

Configurando o Contexto do Pedido por Comando

Para alterá-lo em apenas um comando, você pode simplesmente chamar o serviço do Contexto de Pedido e sobrescrever as suas configurações:

```
// src/Acme/DemoBundle/Command/DemoCommand.php

// ...
class DemoCommand extends ContainerAwareCommand
{
    protected function execute(InputInterface $input, OutputInterface $output)
    {
        $context = $this->getContainer()->get('router')->getContext();
        $context->setHost('example.com');
        $context->setScheme('https');

        // ... your code here
    }
}
```

3.1.16 Como otimizar seu ambiente de desenvolvimento para a depuraração

Ao trabalhar em um projeto Symfony na sua máquina local, você deve usar o ambiente `dev` (front controller `app_dev.php`). Esta configuração de ambiente é otimizada para dois principais propósitos:

- Fornecer ao desenvolvedor um feedback preciso sempre que algo der errado (barra de ferramentas de depuração web, páginas de exceção agradáveis, profiler, ...);
- Ser o mais semelhante possível do ambiente de produção para evitar problemas ao implantar o projeto.

Desabilitando o Arquivo de Bootstrap e o Cache de Classe

E para tornar o ambiente de produção o mais rápido possível, o Symfony cria grandes arquivos PHP em seu cache que contém a agregação das classes PHP que o projeto precisa para cada solicitação. No entanto, este comportamento pode confundir a sua IDE ou seu depurador. Esta receita mostra como você pode ajustar este mecanismo de cache para torná-lo mais amigável quando você precisar depurar código que envolve classes do Symfony.

O front controller `app_dev.php`, por padrão, é o seguinte:

```
// ...

require_once __DIR__.'/../app/bootstrap.php.cache';
require_once __DIR__.'/../app/AppKernel.php';

use Symfony\Component\HttpFoundation\Request;

$kernel = new AppKernel('dev', true);
$kernel->loadClassCache();
$kernel->handle(Request::createFromGlobals())->send();
```

Para deixar o seu depurador ainda mais feliz, desabilite todos os caches de classe PHP, removendo a chamada para `loadClassCache()` e substituindo as declarações require como abaixo:

```
// ...

// require_once __DIR__.'/../app/bootstrap.php.cache';
require_once __DIR__.'/../app/autoload.php';
require_once __DIR__.'/../app/AppKernel.php';

use Symfony\Component\HttpFoundation\Request;

$kernel = new AppKernel('dev', true);
// $kernel->loadClassCache();
$kernel->handle(Request::createFromGlobals())->send();
```

Dica: Se você desativar os caches PHP, não se esqueça de voltar depois da sua sessão de depuração.

Algumas IDEs não gostam do fato de que algumas classes são armazenadas em locais diferentes. Para evitar problemas, você pode dizer a sua IDE para ignorar os arquivos de cache PHP , ou você pode mudar a extensão usada pelo Symfony para esses arquivos:

```
$kernel->loadClassCache('classes', '.php.cache');
```

3.1.17 Dispatcher de Eventos

Como configurar Filtros aplicados antes e após

É bastante comum, no desenvolvimento de aplicações web, precisar que alguma lógica seja executada antes ou após as ações de seu controlador, atuando como filtros ou hooks.

No symfony1, isto era feito através dos métodos PreExecute e postExecute. A maioria dos principais frameworks possuem métodos semelhantes, mas isso não existe no Symfony2. A boa nova é que há uma forma muito melhor para interferir no processo Pedido -> Resposta usando o `componente EventDispatcher`.

Exemplo de validação de token

Imagine que você precisa desenvolver uma API onde alguns controladores são públicos mas outros são restritos a um ou alguns clientes. Para estas funcionalidades privadas, você pode fornecer um token para os clientes identificarem-se.

Então, antes de executar a ação do controlador, você precisa verificar se a ação é restrita ou não. Se for restrita, você precisa validar o token informado.

Nota: Por favor, note que, por simplicidade, nesta receita os tokens serão definidos na configuração e não será usada configuração de banco de dados nem autenticação através do componente de Segurança.

Filtros aplicados Antes com o evento `kernel.controller`

Primeiro, armazene algumas configurações básicas do token usando o `config.yml` e a chave `parameters`:

- *YAML*

```
# app/config/config.yml
parameters:
    tokens:
        client1: pass1
        client2: pass2
```

- *XML*

```
<!-- app/config/config.xml -->
<parameters>
    <parameter key="tokens" type="collection">
        <parameter key="client1">pass1</parameter>
        <parameter key="client2">pass2</parameter>
    </parameter>
</parameters>
```

- *PHP*

```
// app/config/config.php
$container->setParameter('tokens', array(
    'client1' => 'pass1',
    'client2' => 'pass2',
));
```

Tags de Controladores a serem verificadas Um ouvinte `kernel.controller` é notificado em *todos* os pedidos, mesmo antes do controlador ser executado. Então, primeiro, você precisa de alguma forma para identificar se o controlador, que coincide com o pedido, precisa de validação do token.

Uma maneira fácil e limpa é criar uma interface vazia e fazer os controladores implementá-la:

```
namespace Acme\DemoBundle\Controller;

interface TokenAuthenticatedController
{
    // ...
}
```

Um controlador que implementa essa interface ficaria assim:

```
namespace Acme\DemoBundle\Controller;

use Acme\DemoBundle\Controller\TokenAuthenticatedController;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class FooController extends Controller implements TokenAuthenticatedController
{
    // An action that needs authentication
    public function barAction()
    {
        // ...
    }
}
```

Criando um Ouvinte de Evento Em seguida, você precisa criar um ouvinte de evento, que irá conter a lógica que você deseja executar antes de seus controladores. Se você não está familiarizado com ouvintes de eventos, você pode aprender mais sobre eles em `/cookbook/service_container/event_listener`:

```
// src/Acme/DemoBundle/EventListener/TokenListener.php
namespace Acme\DemoBundle\EventListener;

use Acme\DemoBundle\Controller\TokenAuthenticatedController;
use Symfony\Component\HttpKernel\Exception\AccessDeniedHttpException;
use Symfony\Component\HttpKernel\Event\FilterControllerEvent;

class TokenListener
{
    private $tokens;

    public function __construct($tokens)
    {
        $this->tokens = $tokens;
    }

    public function onKernelController(FilterControllerEvent $event)
    {
        $controller = $event->getController();

        /*
         * $controller passed can be either a class or a Closure. This is not usual in Symfony2 but i
         * If it is a class, it comes in array format
         */
        if (!is_array($controller)) {
```

```
            return;
        }

        if ($controller[0] instanceof TokenAuthenticatedController) {
            $token = $event->getRequest()->query->get('token');
            if (!in_array($token, $this->tokens)) {
                throw new AccessDeniedHttpException('This action needs a valid token!');
            }
        }
    }
}
```

Registrando o Ouvinte Finalmente, registre seu ouvinte como um serviço e adicione a ele uma tag de ouvinte de evento. Ao ouvir o `kernel.controller`, você está dizendo ao Symfony que deseja que seu ouvinte seja chamado antes que qualquer controlador seja executado.

- *YAML*

```
# app/config/config.yml (or inside your services.yml)
services:
    demo.tokens.action_listener:
        class: Acme\DemoBundle\EventListener\TokenListener
        arguments: [ %tokens% ]
        tags:
            - { name: kernel.event_listener, event: kernel.controller, method: onKernelControlle
```

- *XML*

```
<!-- app/config/config.xml (or inside your services.xml) -->
<service id="demo.tokens.action_listener" class="Acme\DemoBundle\EventListener\TokenListener">
    <argument>%tokens%</argument>
    <tag name="kernel.event_listener" event="kernel.controller" method="onKernelController" />
</service>
```

- *PHP*

```
// app/config/config.php (or inside your services.php)
use Symfony\Component\DependencyInjection\Definition;

$listener = new Definition('Acme\DemoBundle\EventListener\TokenListener', array('%tokens%'));
$listener->addTag('kernel.event_listener', array('event' => 'kernel.controller', 'method' => 'on
$container->setDefinition('demo.tokens.action_listener', $listener);
```

Com esta configuração, seu método `TokenListener` `onKernelController` será executado em cada pedido. Se o controlador que está prestes a ser executado implementa `TokenAuthenticatedController`, o token de autenticação é aplicado. Isso permite que você tenha um filtro "antes" em qualquer controlador que desejar.

Filtros aplicados "Após" com o evento `kernel.response`

Além de ter um "hook" que é executado antes de seu controlador, você também pode adicionar um hook que será executado *após* seu controlador. Para este exemplo, imagine que você deseja adicionar um hash sha1 (com um salt usando aquele token) para todas as respostas que passaram este token de autenticação.

Outro evento do núcleo do Symfony - chamado `kernel.response` - é notificado em cada pedido, mas depois que o controlador retorna um objeto de Resposta. Criar um ouvinte "após" é tão fácil quanto criar uma classe ouvinte e registrá-la como um serviço neste evento.

Por exemplo, considere o `TokenListener` do exemplo anterior e primeiro grave o token de autenticação dentro dos atributos do pedido. Isto servirá como uma flag básica de que este pedido foi submetido à autenticação por token:

```
public function onKernelController(FilterControllerEvent $event)
{
    // ...

    if ($controller[0] instanceof TokenAuthenticatedController) {
        $token = $event->getRequest()->query->get('token');
        if (!in_array($token, $this->tokens)) {
            throw new AccessDeniedHttpException('This action needs a valid token!');
        }

        // mark the request as having passed token authentication
        $event->getRequest()->attributes->set('auth_token', $token);
    }
}
```

Agora, adicione um outro método nesta classe - `onKernelResponse` - que procura por esta flag no objeto do pedido e define um cabeçalho personalizado na resposta se ele for encontrado:

```
// add the new use statement at the top of your file
use Symfony\Component\HttpKernel\Event\FilterResponseEvent;

public function onKernelResponse(FilterResponseEvent $event)
{
    // check to see if onKernelController marked this as a token "auth'ed" request
    if (!$token = $event->getRequest()->attributes->get('auth_token')) {
        return;
    }

    $response = $event->getResponse();

    // create a hash and set it as a response header
    $hash = sha1($response->getContent().$token);
    $response->headers->set('X-CONTENT-HASH', $hash);
}
```

Finalmente, uma segunda "tag" é necessária na definição do serviço para notificar o Symfony que o evento `onKernelResponse` deve ser notificado para o evento `kernel.response`:

- *YAML*

```
# app/config/config.yml (or inside your services.yml)
services:
    demo.tokens.action_listener:
        class: Acme\DemoBundle\EventListener\TokenListener
        arguments: [ %tokens% ]
        tags:
            - { name: kernel.event_listener, event: kernel.controller, method: onKernelControlle
            - { name: kernel.event_listener, event: kernel.response, method: onKernelResponse }
```

- *XML*

```
<!-- app/config/config.xml (or inside your services.xml) -->
<service id="demo.tokens.action_listener" class="Acme\DemoBundle\EventListener\TokenListener">
    <argument>%tokens%</argument>
    <tag name="kernel.event_listener" event="kernel.controller" method="onKernelController" />
    <tag name="kernel.event_listener" event="kernel.response" method="onKernelResponse" />
</service>
```

- *PHP*

```
// app/config/config.php (or inside your services.php)
use Symfony\Component\DependencyInjection\Definition;

$listener = new Definition('Acme\DemoBundle\EventListener\TokenListener', array('%tokens%'));
$listener->addTag('kernel.event_listener', array('event' => 'kernel.controller', 'method' => 'on
$listener->addTag('kernel.event_listener', array('event' => 'kernel.response', 'method' => 'onKe
$container->setDefinition('demo.tokens.action_listener', $listener);
```

É isso! O `TokenListener` agora é notificado antes de cada controlador ser executado (`onKernelController`) e depois de cada controlador retornar uma resposta (`onKernelResponse`). Fazendo com que controladores específicos implementem a interface `TokenAuthenticatedController` , o ouvinte saberá em quais controladores ele deve agir. E armazenando um valor nos "atributos" do pedido, o método `onKernelResponse` sabe adicionar o cabeçalho extra. Divirta-se!

Como estender uma Classe sem usar Herança

Para permitir que várias classes adicionem métodos para uma outra, você pode definir o método mágico `__call()` na classe que você deseja que seja estendida da seguinte forma:

```
class Foo
{
    // ...

    public function __call($method, $arguments)
    {
        // cria um evento chamado 'foo.method_is_not_found'
        $event = new HandleUndefinedMethodEvent($this, $method, $arguments);
        $this->dispatcher->dispatch($this, 'foo.method_is_not_found', $event);

        // nenhum listener foi capaz de processar o evento? O método não existe
        if (!$event->isProcessed()) {
            throw new \Exception(sprintf('Call to undefined method %s::%s.', get_class($this), $metho
        }

        // retorna o valor retornado pelo listener
        return $event->getReturnValue();
    }
}
```

Ela utiliza um `HandleUndefinedMethodEvent` especial que também deve ser criado. Esta é uma classe genérica que poderia ser reutilizada cada vez que você precisa utilizar esse padrão de extensão de classe:

```
use Symfony\Component\EventDispatcher\Event;

class HandleUndefinedMethodEvent extends Event
{
    protected $subject;
    protected $method;
    protected $arguments;
    protected $returnValue;
    protected $isProcessed = false;

    public function __construct($subject, $method, $arguments)
    {
        $this->subject = $subject;
        $this->method = $method;
```

```
        $this->arguments = $arguments;
    }

    public function getSubject()
    {
        return $this->subject;
    }

    public function getMethod()
    {
        return $this->method;
    }

    public function getArguments()
    {
        return $this->arguments;
    }

    /**
     * Define o valor de retorno e pára a notificação para outros listeners
     */
    public function setReturnValue($val)
    {
        $this->returnValue = $val;
        $this->isProcessed = true;
        $this->stopPropagation();
    }

    public function getReturnValue($val)
    {
        return $this->returnValue;
    }

    public function isProcessed()
    {
        return $this->isProcessed;
    }
}
```

Em seguida, crie uma classe que vai ouvir o evento `foo.method_is_not_found` e *adicionar* o método `bar()`:

```
class Bar
{
    public function onFooMethodIsNotFound(HandleUndefinedMethodEvent $event)
    {
        // queremos somente responder as chamadas do método 'bar'
        if ('bar' != $event->getMethod()) {
            // permite que outro listener cuide deste método desconhecido
            return;
        }

        // o objeto (a instância foo)
        $foo = $event->getSubject();

        // os argumentos do método bar
        $arguments = $event->getArguments();

        // faz algo
```

```
        // ...

        // define o valor de retorno
        $event->setReturnValue($someValue);
    }
}
```

Por fim, adicione o novo método `bar` na classe `Foo` para registrar uma instância de `Bar` com o evento `foo.method_is_not_found`:

```
$bar = new Bar();
$dispatcher->addListener('foo.method_is_not_found', $bar);
```

Como personalizar o Comportamento do Método sem o uso de Herança

Realizar algo antes ou após a Chamada de um Método

Se você deseja realizar algo logo antes ou após um método ser chamado, você pode despachar um evento, respectivamente, no início ou no fim do método:

```
class Foo
{
    // ...

    public function send($foo, $bar)
    {
        // do something before the method
        $event = new FilterBeforeSendEvent($foo, $bar);
        $this->dispatcher->dispatch('foo.pre_send', $event);

        // get $foo and $bar from the event, they may have been modified
        $foo = $event->getFoo();
        $bar = $event->getBar();

        // the real method implementation is here
        $ret = ...;

        // do something after the method
        $event = new FilterSendReturnValue($ret);
        $this->dispatcher->dispatch('foo.post_send', $event);

        return $event->getReturnValue();
    }
}
```

Neste exemplo, dois eventos são lançados: `foo.pre_send`, antes do método ser executado, e `foo.post_send` após o método ser executado. Cada um usa uma classe `Event` personalizada para comunicar informações para os ouvintes dos dois eventos. Essas classes de evento teriam que ser criadas por você e, devem permitir, neste exemplo, que as variáveis `$foo`, `$bar` e `$ret` sejam recuperadas e definidas pelos ouvintes.

Por exemplo, supondo que o `FilterSendReturnValue` tem um método `setReturnValue`, um ouvinte pode ter o seguinte aspecto:

```
public function onFooPostSend(FilterSendReturnValue $event)
{
    $ret = $event->getReturnValue();
    // modify the original ``$ret`` value
```

```
    $event->setReturnValue($ret);
}
```

3.1.18 Request

Como registrar um novo Formato de Requisição e de Mime Type

Todo `Request` possui um "formato" (e.g. `html`, `json`), que é usado para determinar o tipo de conteúdo a ser retornado pelo `Response`. Na verdade, o formato de requisição, acessível pelo método `getRequestFormat()`, é usado para definir o MIME type do cabeçalho `Content-Type` no objeto `Response`. Internamente, Symfony contém um mapa dos formatos mais comuns (e.g. `html`, `json`) e seus MIME types associados (e.g. `text/html`, `application/json`). Naturalmente, formatos adicionais de MIME type de entrada podem ser facilmente adicionados. Este documento irá mostrar como você pode adicionar o formato `jsonp` e seu MIME type correspondente.

Criando um `kernel.request` Listener

A chave para definir um novo MIME type é criar uma classe que irá "ouvir" o evento `kernel.request` enviado pelo kernel do Symfony. O evento `kernel.request` é enviado no início no processo de manipulação da requisição Symfony e permite que você modifique o objeto da requisição.

Crie a seguinte classe, substituindo o caminho com um caminho para um pacote em seu projeto:

```
// src/Acme/DemoBundle/RequestListener.php
namespace Acme\DemoBundle;

use Symfony\Component\HttpKernel\HttpKernelInterface;
use Symfony\Component\HttpKernel\Event\GetResponseEvent;

class RequestListener
{
    public function onKernelRequest(GetResponseEvent $event)
    {
        $event->getRequest()->setFormat('jsonp', 'application/javascript');
    }
}
```

Registrando seu Listener

Como para qualquer outro listener, você precisa adicioná-lo em um arquivo de configuração e registrá-lo como um listerner adicionando a tag `kernel.event_listener`:

- *XML*

```
<!-- app/config/config.xml -->
<?xml version="1.0" ?>

<container xmlns="http://symfony.com/schema/dic/services"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/dic/services http://symfony.com/schema/dic/ser

    <service id="acme.demobundle.listener.request" class="Acme\DemoBundle\RequestListener">
        <tag name="kernel.event_listener" event="kernel.request" method="onKernelRequest" />
    </service>
```

```
    </container>
```

- *YAML*

```
    # app/config/config.yml
    services:
        acme.demobundle.listener.request:
            class: Acme\DemoBundle\RequestListener
            tags:
                - { name: kernel.event_listener, event: kernel.request, method: onKernelRequest }
```

- *PHP*

```
    # app/config/config.php
    $definition = new Definition('Acme\DemoBundle\RequestListener');
    $definition->addTag('kernel.event_listener', array('event' => 'kernel.request', 'method' => 'onK
    $container->setDefinition('acme.demobundle.listener.request', $definition);
```

Neste ponto, o serviço `acme.demobundle.listener.request` foi configurado e será notificado quando o Symfony kernel enviar um evento `kernel.request`.

Dica: Você também pode registrar o ouvinte em uma classe de extensão de configuração (see *Importando Configuração através de Extensões do Container* para mais informações).

3.1.19 Profiler

Como criar um Coletor de Dados personalizado

O Profiler do Symfony2 delega a coleta de dados para os coletores de dados. O Symfony2 vem com alguns deles, mas você pode criar o seu próprio facilmente.

Criando um Coletor de Dados Personalizado

Criar um coletor de dados personalizado é tão simples quanto implementar o `DataCollectorInterface`:

```
interface DataCollectorInterface
{
    /**
     * Collects data for the given Request and Response.
     *
     * @param Request    $request   A Request instance
     * @param Response   $response  A Response instance
     * @param \Exception $exception An Exception instance
     */
    function collect(Request $request, Response $response, \Exception $exception = null);

    /**
     * Returns the name of the collector.
     *
     * @return string The collector name
     */
    function getName();
}
```

O método `getName()` deve retornar um nome único. Ele é usado para acessar a informação mais tarde (veja o `/cookbook/testing/profiling` por exemplo).

O método `collect()` é responsável por armazenar os dados que ele quer fornecer acesso nas propriedades locais.

Cuidado: Como o profiler serializa instâncias do coletor de dados, você não deve armazenar objetos que não podem ser serializados (como objetos PDO), ou você precisa fornecer o seu próprio método `serialize()`.

Na maioria das vezes, é conveniente estender o `DataCollector` e popular a propriedade `$this->data` (que cuida da serialização da propriedade `$this->data`):

```
class MemoryDataCollector extends DataCollector
{
    public function collect(Request $request, Response $response, \Exception $exception = null)
    {
        $this->data = array(
            'memory' => memory_get_peak_usage(true),
        );
    }

    public function getMemory()
    {
        return $this->data['memory'];
    }

    public function getName()
    {
        return 'memory';
    }
}
```

Ativando Coletores de Dados Personalizados

Para ativar um coletor de dados, adicione-o como um serviço regular, em uma de suas configurações, e adicione a tag `data_collector` a ele:

- *YAML*

```
services:
    data_collector.your_collector_name:
        class: Fully\Qualified\Collector\Class\Name
        tags:
            - { name: data_collector }
```

- *XML*

```
<service id="data_collector.your_collector_name" class="Fully\Qualified\Collector\Class\Name">
    <tag name="data_collector" />
</service>
```

- *PHP*

```
$container
    ->register('data_collector.your_collector_name', 'Fully\Qualified\Collector\Class\Name')
    ->addTag('data_collector')
;
```

Adição de Templates no Profiler Web

Quando você quiser exibir os dados coletados pelo seu Coletor de Dados na barra de ferramentas para debug web ou no profiler web, adicione um template Twig seguindo este esqueleto:

```
{% extends 'WebProfilerBundle:Profiler:layout.html.twig' %}

{% block toolbar %}
    {# the web debug toolbar content #}
{% endblock %}

{% block head %}
    {# if the web profiler panel needs some specific JS or CSS files #}
{% endblock %}

{% block menu %}
    {# the menu content #}
{% endblock %}

{% block panel %}
    {# the panel content #}
{% endblock %}
```

Cada bloco é opcional. O bloco `toolbar` é usado para a barra de ferramentas para debug web e o `menu` e `panel` são usados para adicionar um painel no profiler web.

Todos os blocos têm acesso ao objeto `collector`.

Dica: Templates integrados usam uma imagem codificada em base64 para a barra de ferramentas (`<img src="..."`). Você pode facilmente calcular o valor base64 para uma imagem com este pequeno script: `echo base64_encode(file_get_contents($_SERVER['argv'][1]));`.

Para ativar o template, adicione um atributo `template` para a tag `data_collector` em sua configuração. Por exemplo, assumindo que o seu template está em algum `AcmeDebugBundle`:

- *YAML*

```
services:
    data_collector.your_collector_name:
        class: Acme\DebugBundle\Collector\Class\Name
        tags:
            - { name: data_collector, template: "AcmeDebugBundle:Collector:templatename", id: "y
```

- *XML*

```
<service id="data_collector.your_collector_name" class="Acme\DebugBundle\Collector\Class\Name">
    <tag name="data_collector" template="AcmeDebugBundle:Collector:templatename" id="your_collec
</service>
```

- *PHP*

```
$container
    ->register('data_collector.your_collector_name', 'Acme\DebugBundle\Collector\Class\Name')
    ->addTag('data_collector', array(
        'template' => 'AcmeDebugBundle:Collector:templatename',
        'id'       => 'your_collector_name',
    ))
;
```

3.1.20 Como o Symfony2 difere do symfony1

O framework Symfony2 incorpora uma evolução significativa quando comparado com a primeira versão do framework. Felizmente, com a arquitetura MVC em sua essência, as habilidades usadas para dominar um projeto symfony1 continuam a ser muito relevantes ao desenvolver com o Symfony2. Claro, o `app.yml` não existe mais, mas quanto ao roteamento, controladores e templates, todos permanecem.

Neste capítulo, vamos percorrer as diferenças entre o symfony1 e Symfony2. Como você verá, muitas tarefas são abordadas de uma forma ligeiramente diferente. Você vai apreciar estas pequenas diferenças pois elas promovem um código estável, previsível, testável e desacoplado em suas aplicações Symfony2.

Então, sente e relaxe, pois vamos guiá-lo a partir de agora.

Estrutura de Diretórios

Ao olhar um projeto Symfony2 - por exemplo, o Symfony2 Standard - você verá uma estrutura de diretórios muito diferente da encontrada no symfony1. As diferenças, no entanto, são um tanto superficiais.

O Diretório `app/`

No symfony1, o seu projeto tem uma ou mais aplicações, e cada uma fica localizada dentro do diretório `apps/` (ex. `apps/frontend`). Por padrão, no Symfony2, você tem apenas uma aplicação representada pelo diretório `app/`. Como no symfony1, o diretório `app/` contém a configuração específica para determinada aplicação. Ele também contém os diretórios cache, log e template específicos para a aplicação, bem como, uma classe `Kernel` (`AppKernel`), que é o objeto base que representa a aplicação.

Ao contrário do symfony1, quase nenhum código PHP reside no diretório `app/`. Este diretório não pretende armazenar os módulos ou arquivos de biblioteca como acontece no symfony1. Em vez disso, ele é simplesmente o lar da configuração e outros recursos (templates, arquivos de tradução).

O Diretório `src/`

Simplificando, o seu código fica armazendo aqui. No Symfony2, todo o código real da aplicação reside dentro de um bundle (equivalente a um plugin no symfony1) e, por padrão, cada bundle reside dentro do diretório `src`. Dessa forma, o diretório `src` é um pouco parecido com o diretório `plugins` do symfony1, mas muito mais flexível. Além disso, enquanto os *seus* bundles vão residir no diretório `src/`, os bundles de terceiros irão residir am algum lugar no diretório `vendor/`.

Para obter uma imagem melhor do diretório `src/`, vamos primeiro pensar em uma aplicação symfony1. Primeiro, parte do seu código provavelmente reside dentro de uma ou mais aplicações. Mais geralmente estas incluem módulos, mas também podem incluir quaisquer outras classes PHP que você adicionar na sua aplicação. Você também pode ter criado um arquivo `schema.yml` no diretório `config` do seu projeto e construiu vários arquivos de modelo. Finalmente, para ajudar com alguma funcionalidade comum, você está usando vários plugins de terceiros que residem no diretório `plugins/`. Em outras palavras, o código que compõem a sua aplicação reside em vários locais diferentes.

No Symfony2, a vida é muito mais simples porque *todo* o código Symfony2 deve residir em um bundle. Em nosso projeto symfony1, todo o código *pode* ser movido em um ou mais plugins (o que, na verdade, é uma prática muito boa). Assumindo que todos os módulos, classes PHP, esquema, configuração de roteamento, etc, foram movidos para um plugin, o diretório `plugins/` do symfony1 seria muito semelhante ao diretório `src/` do Symfony2.

Simplificando novamente, o diretório `src/` é onde o seu código, assets, templates e mais qualquer outra coisa específica ao seu projeto, vai residir.

O Diretório `vendor/`

O diretório `vendor/` é basicamente equivalente ao diretório `lib/vendor/` do symfony1, que foi o diretório convencional para todas as bibliotecas vendor e bundles. Por padrão, você vai encontrar os arquivos de biblioteca do Symfony2 nesse diretório, juntamente com várias outras bibliotecas dependentes, como Doctrine2, Twig e SwiftMailer. Bundles Symfony2 de terceiros residem em algum local no diretório `vendor/`.

O Diretório `web/`

Não mudou muita coisa no diretório `web/`. A diferença mais notável é a ausência dos diretórios `css/`, `js/` e `images/`. Isto é intencional. Tal como o seu código PHP, todos os assets também devem residir dentro de um bundle. Com a ajuda de um comando do console, o diretório `Resources/public/` de cada pacote é copiado ou ligado simbolicamente ao diretório `web/bundles/`. Isto permite-lhe manter os assets organizados dentro do seu bundle, mas, ainda torná-los disponíveis ao público. Para certificar-se de que todos os bundles estão disponíveis, execute o seguinte comando:

```
php app/console assets:install web
```

Nota: Este comando é equivalente no Symfony2 ao comando `plugin:publish-assets` do symfony1.

O Autoloading

Uma das vantagens dos frameworks modernos é nunca ter que preocupar-se em incluir arquivos. Ao fazer uso de um autoloader, você pode fazer referência à qualquer classe em seu projeto e confiar que ela estará disponível. O autoloading mudou no Symfony2 para ser mais universal, mais rápido e independente da necessidade de limpar o seu cache.

No symfony1, o autoloading é realizado pesquisando em todo o projeto pela presença de arquivos de classe PHP e realizando o cache desta informação em um array gigante. Este array diz ao symfony1 exatamente qual arquivo continha cada classe. No ambiente de produção, isto faz com que você precise limpar o cache quando as classes forem adicionadas ou movidas.

No Symfony2, uma nova classe - `UniversalClassLoader` - lida com esse processo. A idéia por trás do autoloader é simples: o nome da sua classe (incluindo o namespace) deve coincidir com o caminho para o arquivo que contém essa classe. Considere, por exemplo, o `FrameworkExtraBundle` da Edição Standard do Symfony2:

```
namespace Sensio\Bundle\FrameworkExtraBundle;

use Symfony\Component\HttpKernel\Bundle\Bundle;
// ...

class SensioFrameworkExtraBundle extends Bundle
{
    // ...
```

O arquivo em si reside em `vendor/sensio/framework-extra-bundle/Sensio/Bundle/FrameworkExtraBundle/S`
Como você pode ver, a localização do arquivo segue o namespace da classe. Especificamente, o namespace, `Sensio\Bundle\FrameworkExtraBundle`, indica o diretório que o arquivo deve residir em (`vendor/sensio/framework-extra-bundle/Sensio/Bundle/FrameworkExtraBundle/`).
Isto é porque, no arquivo `app/autoload.php`, você vai configurar o Symfony para procurar pelo namespace `Sensio` no diretório `vendor/sensio`:

```
// app/autoload.php

// ...
$loader->registerNamespaces(array(
    // ...
    'Sensio'           => __DIR__.'/../vendor/sensio/framework-extra-bundle',
));
```

Se o arquivo *não* residir nesta localização exata, você receberá o seguinte erro: `Class "Sensio\Bundle\FrameworkExtraBundle\SensioFrameworkExtraBundle"does not exist.`. No Symfony2, a mensagem "classe não existe" significa que o namespace suspeito da classe e a sua localização física não correspondem. Basicamente, o Symfony2 estará procurando em um local exato por esta classe, mas esse local não existe (ou contém uma classe diferente). A fim de que uma classe seja carregada automaticamente, você **nunca precisará limpar o cache** no Symfony2.

Como mencionado anteriormente, para o autoloader funcionar, ele precisa saber que o namespace `Sensio` reside no diretório `vendor/bundles` e que, por exemplo, o namespace `Doctrine` reside no diretório `vendor/doctrine/lib/` . Este mapeamento é controlado inteiramente por você através do arquivo `app/autoload.php`.

Se você olhar o `HelloController` da Edição Standard do Symfony2 poderá ver que ele reside no namespace `Acme\DemoBundle\Controller`. Sim, o namespace `Acme` não é definido no `app/autoload.php`. Por padrão, você não precisa configurar explicitamente o local dos bundles que residem no diretório `src/`. O `UniversalClassLoader` está configurado para usar como alternativa o diretório `src/` usando o seu método `registerNamespaceFallbacks`:

```
// app/autoload.php

// ...
$loader->registerNamespaceFallbacks(array(
    __DIR__.'/../src',
));
```

Usando o Console

No symfony1, o console está no diretório raiz do seu projeto e é chamado `symfony`:

```
php symfony
```

No Symfony2, o console está agora no sub-diretório app e é chamado `console`:

```
php app/console
```

Aplicações

Em um projeto symfony1, é comum ter várias aplicações: uma para o frontend e uma para o backend, por exemplo.

Em um projeto Symfony2, você só precisa criar uma única aplicação (um aplicação blog, uma aplicação intranet, ...). Na maioria das vezes, se você desejar criar uma segunda aplicação, você pode, em vez, criar outro projeto e compartilhar alguns bundles entre elas.

E, se você precisa separar as funcionalidades do frontend e backend de alguns bundles, você pode criar sub-namespaces para os controladores, sub-diretórios para templates, diferentes configurações semânticas, configurações separadas de roteamento, e assim por diante.

Claro, não há nada de errado em ter várias aplicações em seu projeto, isto depende inteiramente de você. Uma segunda aplicação significaria um novo diretório, por exemplo: `my_app/`, com a mesma configuração básica do diretório `app/`.

Dica: Leia a definição de Projeto, Aplicação e Bundle no glossário.

Bundles e Plugins

Em um projeto symfony1, um plugin pode conter configuração, módulos, bibliotecas PHP, assets e qualquer outra coisa relacionada ao seu projeto. No Symfony2, a idéia de um plugin é substituída pelo "bundle". Um bundle é ainda mais poderoso do que um plugin porque o núcleo do framework Symfony2 é fornecido através de uma série de bundles. No Symfony2, os bundles são cidadãos de primeira classe que são tão flexíveis que mesmo o código do núcleo em si é um bundle.

No symfony1, um plugin deve ser ativado dentro da classe `ProjectConfiguration`:

```
// config/ProjectConfiguration.class.php
public function setup()
{
    $this->enableAllPluginsExcept(array(/* some plugins here */));
}
```

No Symfony2, os bundles são ativados dentro do kernel da aplicação:

```
// app/AppKernel.php
public function registerBundles()
{
    $bundles = array(
        new Symfony\Bundle\FrameworkBundle\FrameworkBundle(),
        new Symfony\Bundle\TwigBundle\TwigBundle(),
        // ...
        new Acme\DemoBundle\AcmeDemoBundle(),
    );

    return $bundles;
}
```

Roteamento (`routing.yml`) e configuração (`config.yml`)

No symfony1, os arquivos de configuração `routing.yml` e `app.yml` são automaticamente carregados dentro de qualquer plugin. No Symfony2, o roteamento e a configuração da aplicação dentro de um bundle devem ser incluídos manualmente. Por exemplo, para incluir um recurso de roteamento de um bundle chamado `AcmeDemoBundle`, você pode fazer o seguinte:

```
# app/config/routing.yml
_hello:
    resource: "@AcmeDemoBundle/Resources/config/routing.yml"
```

Isto irá carregar as rotas encontradas no arquivo `Resources/config/routing.yml` do `AcmeDemoBundle`. O especial `@AcmeDemoBundle` é uma sintaxe de atalho que, internamente, resolve o caminho completo para esse bundle.

Você pode usar essa mesma estratégia para trazer a configuração de um bundle:

```
# app/config/config.yml
imports:
    - { resource: "@AcmeDemoBundle/Resources/config/config.yml" }
```

No Symfony2, a configuração é um pouco semelhante ao `app.yml` do symfony1, exceto que é muito mais sistemática. Com o `app.yml`, você poderia simplesmente criar as chaves que desejava. Por padrão, as entradas eram sem significado e dependia inteiramente de como você utilizava em sua aplicação:

```
# some app.yml file from symfony1
all:
  email:
    from_address:  foo.bar@example.com
```

No Symfony2, você também pode criar entradas arbitrárias sob a chave `parameters` de sua configuração:

```
parameters:
    email.from_address: foo.bar@example.com
```

Você pode agora acessar ele a partir de um controlador, por exemplo:

```
public function helloAction($name)
{
    $fromAddress = $this->container->getParameter('email.from_address');
}
```

Na realidade, a configuração no Symfony2 é muito mais potente e é usada principalmente para configurar objetos que você pode usar. Para maiores informações, visite o capítulo intitulado "Container de Serviço".

3.1.21 Como implantar uma aplicação Symfony2

Nota: A implantação pode ser uma tarefa complexa e variada, dependendo das configurações e necessidades. Este artigo não tenta explicar tudo, mas oferece os requisitos e idéias mais comuns para a implantação.

Noções básicas sobre implantação no Symfony2

As etapas comuns realizadas ao implantar uma aplicação Symfony2 incluem:

1. Fazer o upload do seu código modificado para o servidor;
2. Atualizar as suas dependências vendor (normalmente feita com o Composer, e pode ser feita antes do upload);
3. Executar migrações de banco de dados ou tarefas similares para atualizar quaisquer estruturas de dados modificadas;
4. Limpeza (e talvez mais importante, warming up) do cache.

A implantação também pode incluir outras coisas, tais como:

- Marcação (Tag) de uma versão específica do seu código como um release em seu repositório de controle de código fonte;
- Criação de uma "staging area" temporária para construir a sua configuração atualizada "offline";
- Execução de testes disponíveis para garantir a estabilidade do código e/ou do servidor;
- Remoção de todos os arquivos desnecessários do `web` para manter o seu ambiente de produção limpo;
- Limpeza de sistemas de cache externos (como Memcached ou Redis).

Como implantar uma aplicação Symfony2

Existem várias maneiras de implantar uma aplicação Symfony2.

Vamos começar com algumas estratégias de implantação básicas e construir a partir daí.

Transferência de Arquivo Básica

A forma mais básica de implantar uma aplicação é copiando os arquivos manualmente via FTP/SCP (ou método similar). Isto tem as suas desvantagens, pois você não têm controle sobre o sistema com o progresso de atualização. Este método também requer que você realize algumas etapas manuais após transferir os arquivos (consulte *Tarefas Comuns Pós-Implantação*)

Usando Controle de Código Fonte

Se você estiver usando controle de código fonte (por exemplo, git ou svn), você pode simplificar com a sua instalação ao vivo também sendo uma cópia de seu repositório. Quando estiver pronto para atualizá-lo é tão simples quanto buscar as últimas atualizações de seu sistema de controle de código fonte.

Isso torna a atualização de seus arquivos *fácil*, mas você ainda precisa se preocupar com a realização manual de outros passos (consulte *Tarefas Comuns Pós-Implantação*).

Usando Build scripts e outras ferramentas

Há também ferramentas de alta qualidade para ajudar a aliviar o sofrimento da implantação. Existem até mesmo algumas ferramentas que foram especificamente adaptadas às exigências do Symfony2, e que tem um cuidado especial para garantir que tudo, antes, durante e após uma implantação ocorreu corretamente.

Veja *As ferramentas* para uma lista de ferramentas que podem ajudar com a implantação.

Tarefas Comuns Pós-Implantação

Depois de implantar o seu código fonte, há uma série de tarefas comuns que precisam ser feitas:

A) Configurar o seu arquivo `app/config/parameters.ini`

Este arquivo deve ser personalizado em cada sistema. O método utilizado para implantar o seu código fonte *não* deve implantar esse arquivo. Em vez disso, você deve configurá-lo manualmente (ou através de algum processo de construção) em seu(s) servidor(es).

B) Atualizar os seus vendors

Seus vendors podem ser atualizados antes de transferir o seu código fonte (ou seja, atualizar o diretório `vendor/`, e, em seguida, transferir com seu código fonte) ou depois no servidor. De qualquer forma, apenas atualize os seus vendors como faria normalmente:

```
$ php composer.phar install --optimize-autoloader
```

Dica: A flag `--optimize-autoloader` deixa o autoloader do Composer com mais performance através da construção de um "mapa de classe".

C) Limpar o cache do Symfony

Certifique-se de limpar (e o warm-up) o cache do Symfony:

```
$ php app/console cache:clear --env=prod --no-debug
```

D) Dump dos assets do Assetic

Se você estiver usando o Assetic, você também vai desejar fazer o dump de seus assets:

```
$ php app/console assetic:dump --env=prod --no-debug
```

E) Outras coisas!

Podem haver muitas outras coisas que você precisa fazer, dependendo de sua configuração:

- A execução de quaisquer migrações de banco de dados
- Limpar o cache do APC
- Executar `assets:install` (já considerado em `composer.phar install`)
- Adicionar/editar CRON jobs
- Mover os assets para um CDN
- ...

Ciclo de Vida da Aplicação: Integração Contínua, QA, etc

Embora este artigo abrange os detalhes técnicos da implantação, o ciclo de vida completo de buscar o código do desenvolvimento até a produção pode ter muito mais passos (pense na implantação para staging, QA, execução de testes, etc.)

O uso de staging, testes, QA, integração contínua, as migrações de banco de dados e a capacidade de reverter em caso de falha são todos fortemente aconselhados. Existem ferramentas simples e outras mais complexas, que podem fazer a implantação tão fácil (ou sofisticada) quanto o seu ambiente requer.

Não se esqueça que a implantação de sua aplicação também envolve a atualização de qualquer dependência (normalmente através do Composer), a migração do seu banco de dados, limpar o cache e outras coisas potenciais como mover os assets para um CDN (consulte *Tarefas Comuns Pós-Implantação*).

As Ferramentas

Capifony:

> Esta ferramenta fornece um conjunto especializado de ferramentas no topo do Capistrano, especificamente sob medida para projetos symfony e Symfony2.

sf2debpkg:

Esta ferramenta ajuda a criar um pacote Debian nativo para o seu projeto Symfony2.

Magallanes:

Esta ferramenta de implantação, semelhante ao Capistrano, é construída em PHP, e pode ser mais fácil, para os desenvolvedores PHP, estendê-la para as suas necessidades.

Bundles:

Há muitos bundles que adicionam recursos de implantação diretamente em seu console do Symfony2.

Script básico:

Você pode, com certeza, usar o shell, Ant, ou qualquer outra ferramenta de construção para criar o script de implantação de seu projeto.

Plataforma como Prestadora de Serviços:

PaaS é uma forma relativamente nova para implantar sua aplicação. Tipicamente uma PaaS vai usar um único arquivo de configuração no diretório raiz do seu projeto para determinar como construir um ambiente em tempo real que suporta o seu software. Um provedor com suporte confirmado para o Symfony2 é o PagodaBox.

Dica: Procurando mais? Fale com a comunidade no Canal IRC do Symfony #symfony (no freenode) para obter mais informações.

- Assetic
  - Como usar o Assetic para o Gerenciamento de Assets
  - Como Minificar JavaScripts e Folhas de Estilo com o YUI Compressor
  - Como usar o Assetic para otimização de imagem com funções do Twig
  - Como Aplicar um filtro Assetic a uma extensão de arquivo específica
- Bundles
  - Como usar Melhores Práticas para a Estruturação dos Bundles
  - Como usar herança para substituir partes de um Bundle
  - Como Sobrescrever qualquer parte de um Bundle
  - Como expor uma Configuração Semântica para um Bundle
- Cache
  - Como usar Varnish para aumentar a velocidade do meu Website
- Configuração
  - Como Dominar e Criar novos Ambientes
  - Como Substituir a Estrutura de Diretório Padrão do Symfony
  - Como definir Parâmetros Externos no Container de Serviços
  - Como usar o PdoSessionStorage para armazenar as Sessões no Banco de Dados
  - Como usar o Apache Router
- Console
  - Como criar um Comando de Console
  - Como usar o Console

Leia o Cookbook.

CAPÍTULO 4

Componentes

4.1 Os Componentes

4.1.1 O Componente ClassLoader

O Componente ClassLoader carrega as classes do seu projeto automaticamente se elas seguirem algumas convenções PHP padrão.

Sempre que você usar uma classe indefinida, o PHP utiliza o mecanismo de autoload automático para delegar o carregamento de um arquivo definindo a classe. O Symfony2 fornece um autoloader "universal", que é capaz de carregar classes de arquivos que implementam uma das seguintes convenções:

- Os padrões técnicos de interoperabilidade para namespaces do PHP 5.3 e para nomes de classes;
- A convenção de nomenclatura de classes do PEAR.

Se as suas classes e as bibliotecas de terceiros que você usa no seu projeto seguem estes padrões, o autoloader do Symfony2 é o único autoloader que você vai precisar.

Instalação

Você pode instalar o componente de várias formas diferentes:

- Usando o repositório Git oficial (https://github.com/symfony/ClassLoader);
- Instalando via PEAR (*pear.symfony.com/ClassLoader*);
- Instalando via Composer (*symfony/class-loader* no Packagist).

Uso

Novo na versão 2.1: O método `useIncludePath` foi adicionado no Symfony 2.1.

Registrar o autoloader `UniversalClassLoader` é simples:

```
require_once '/path/to/src/Symfony/Component/ClassLoader/UniversalClassLoader.php';

use Symfony\Component\ClassLoader\UniversalClassLoader;

$loader = new UniversalClassLoader();

// You can search the include_path as a last resort.
```

```
$loader->useIncludePath(true);

// ... register namespaces and prefixes here - see below

$loader->register();
```

Para ganhos de desempenho menores, os caminhos das classes podem ser armazenados em memória usando com o APC registrando o `ApcUniversalClassLoader`:

```
require_once '/path/to/src/Symfony/Component/ClassLoader/UniversalClassLoader.php';
require_once '/path/to/src/Symfony/Component/ClassLoader/ApcUniversalClassLoader.php';

use Symfony\Component\ClassLoader\ApcUniversalClassLoader;

$loader = new ApcUniversalClassLoader('apc.prefix.');
$loader->register();
```

O autoloader somente é útil se você adicionar algumas bibliotecas para autoload.

Nota: O autoloader é automaticamente registrado em uma aplicação Symfony2 (veja `app/autoload.php`).

Se as classes para o autoload usam namespaces, utilize os métodos `registerNamespace()` ou `registerNamespaces()`

```
$loader->registerNamespace('Symfony', __DIR__.'/vendor/symfony/symfony/src');

$loader->registerNamespaces(array(
    'Symfony' => __DIR__.'/../vendor/symfony/symfony/src',
    'Monolog' => __DIR__.'/../vendor/monolog/monolog/src',
));

$loader->register();
```

Para as classes que seguem a convenção de nomenclatura do PEAR, utilize os métodos `registerPrefix()` ou `registerPrefixes()`

```
$loader->registerPrefix('Twig_', __DIR__.'/vendor/twig/twig/lib');

$loader->registerPrefixes(array(
    'Swift_' => __DIR__.'/vendor/swiftmailer/swiftmailer/lib/classes',
    'Twig_'  => __DIR__.'/vendor/twig/twig/lib',
));

$loader->register();
```

Nota: Algumas bibliotecas também exigem que seu caminho raiz seja registrado no include path do PHP (`set_include_path()`).

Classes de um sub-namespace ou uma sub-hierarquia das classes PEAR podem ser buscadas em uma lista de localização para facilitar o vendoring de um sub-conjunto de classes para projetos grandes:

```
$loader->registerNamespaces(array(
    'Doctrine\\Common'           => __DIR__.'/vendor/doctrine/common/lib',
    'Doctrine\\DBAL\\Migrations' => __DIR__.'/vendor/doctrine/migrations/lib',
    'Doctrine\\DBAL'             => __DIR__.'/vendor/doctrine/dbal/lib',
    'Doctrine'                   => __DIR__.'/vendor/doctrine/orm/lib',
```

```
));

$loader->register();
```

Neste exemplo, se você tentar usar uma classe no namespace `Doctrine\Common` ou em um de seus filhos, o autoloader vai procurar primeiro pela classe sob o diretório `doctrine-common`, e vai, em seguida, retornar para o diretório padrão `Doctrine` (o último configurado) se não for encontrada, antes de desistir. A ordem das inscrições é significativa neste caso.

4.1.2 Console

O Componente Console

O componente Console facilita a criação de interfaces de linha de comando bonitas e testáveis.

O componente Console permite a criação dos seus próprios comandos de linha de comando. Seus comandos do console podem ser usados para qualquer tarefa recorrente, como cronjobs, importações ou outros trabalhos realizados em lote.

Instalação

Você pode instalar o componente de várias formas diferentes:

- Usando o repositório Git oficial (https://github.com/symfony/Console);
- Instalando via PEAR (*pear.symfony.com/Console*);
- Instalando via Composer (*symfony/console* on Packagist).

Criando um comando básico

Para fazer um comando de console que nos cumprimenta na linha de comando, crie o `GreetCommand.php` e adicione o seguinte código nele:

```
namespace Acme\DemoBundle\Command;

use Symfony\Component\Console\Command\Command;
use Symfony\Component\Console\Input\InputArgument;
use Symfony\Component\Console\Input\InputInterface;
use Symfony\Component\Console\Input\InputOption;
use Symfony\Component\Console\Output\OutputInterface;

class GreetCommand extends Command
{
    protected function configure()
    {
        $this
            ->setName('demo:greet')
            ->setDescription('Greet someone')
            ->addArgument('name', InputArgument::OPTIONAL, 'Who do you want to greet?')
            ->addOption('yell', null, InputOption::VALUE_NONE, 'If set, the task will yell in upperca
        ;
    }

    protected function execute(InputInterface $input, OutputInterface $output)
```

```
    {
        $name = $input->getArgument('name');
        if ($name) {
            $text = 'Hello '.$name;
        } else {
            $text = 'Hello';
        }

        if ($input->getOption('yell')) {
            $text = strtoupper($text);
        }

        $output->writeln($text);
    }
}
```

Você também precisa criar o arquivo para ser executado na linha de comando que cria uma `Application` e adicionar comandos à ele:

```
#!/usr/bin/env php
# app/console
<?php

use Acme\DemoBundle\Command\GreetCommand;
use Symfony\Component\Console\Application;

$application = new Application();
$application->add(new GreetCommand);
$application->run();
```

Teste o novo comando de console executando o seguinte

```
$ app/console demo:greet Fabien
```

Isto irá imprimir o seguinte na linha de comando:

```
Hello Fabien
```

Você também pode usar a opção `--yell` para converter tudo em letras maiúsculas:

```
$ app/console demo:greet Fabien --yell
```

Irá mostrar:

```
HELLO FABIEN
```

Colorindo a saída Sempre que a saída for um texto, você pode envolvê-lo com tags para colorir a sua saída. Por exemplo:

```
// green text
$output->writeln('<info>foo</info>');

// yellow text
$output->writeln('<comment>foo</comment>');

// black text on a cyan background
$output->writeln('<question>foo</question>');
```

```
// white text on a red background
$output->writeln('<error>foo</error>');
```

É possível definir os seus próprios estilos usando a classe `OutputFormatterStyle`:

```
$style = new OutputFormatterStyle('red', 'yellow', array('bold', 'blink'));
$output->getFormatter()->setStyle('fire', $style);
$output->writeln('<fire>foo</fire>');
```

As cores de primeiro plano e de fundo disponíveis são: `black`, `red`, `green`, `yellow`, `blue`, `magenta`, `cyan` e `white`.

E as opções disponíveis são: `bold`, `underscore`, `blink`, `reverse` e `conceal`.

Usando argumentos de comando

A parte mais interessante dos comandos são os argumentos e opções que você pode disponibilizar. Argumentos são as strings - separados por espaços - que vem após o nome do comando. Eles são ordenados, e podem ser opcionais ou obrigatórios. Por exemplo, adicione um argumento opcional `last_name` ao comando e torne o argumento `name` obrigatório:

```
$this
    // ...
    ->addArgument('name', InputArgument::REQUIRED, 'Who do you want to greet?')
    ->addArgument('last_name', InputArgument::OPTIONAL, 'Your last name?');
```

Agora, você tem acesso a um argumento `last_name` no seu comando:

```
if ($lastName = $input->getArgument('last_name')) {
    $text .= ' '.$lastName;
}
```

O comando pode agora ser usado em qualquer uma das seguintes formas:

```
$ app/console demo:greet Fabien
$ app/console demo:greet Fabien Potencier
```

Utilizando opções de comando

Ao contrário dos argumentos, as opções não são ordenadas (ou seja, você pode especificá-las em qualquer ordem) e são especificadas com dois traços (ex., `--yell` - você também pode declarar um atalho de uma letra que pode chamar com um traço único, como `-y`). As opções são *sempre* opcionais, e podem ser configuradas para aceitar um valor (ex. `dir=src`) ou simplesmente como um sinalizador booleano sem um valor (ex. `yell`).

Dica: Também é possível fazer uma opção *opcionalmente* aceitar um valor (de modo que `--yell` ou `yell=loud` funcionem). Opções também podem ser configuradas para aceitar um array de valores.

Por exemplo, adicione uma nova opção ao comando que pode ser usada para especificar quantas vezes a mensagem deve ser impressa:

```
$this
    // ...
    ->addOption('iterations', null, InputOption::VALUE_REQUIRED, 'How many times should the message b
```

Em seguida, use o código abaixo no comando para imprimir a mensagem várias vezes:

```
for ($i = 0; $i < $input->getOption('iterations'); $i++) {
    $output->writeln($text);
}
```

Agora, quando executar a tarefa, você pode, opcionalmente, especificar uma flag `--iterations`:

```
$ app/console demo:greet Fabien
$ app/console demo:greet Fabien --iterations=5
```

O primeiro exemplo irá imprimir apenas uma vez, pois `iterations` está vazio e o padrão é `1` (o último argumento de `addOption`). O segundo exemplo irá imprimir cinco vezes.

Lembre-se de que as opções não se importam com a ordem. Então, qualquer um dos seguintes comandos vai funcionar:

```
$ app/console demo:greet Fabien --iterations=5 --yell
$ app/console demo:greet Fabien --yell --iterations=5
```

Existem quatro variantes de opções que você pode usar:

Opção	Valor
InputOption::VALUE_IS_ARRAY	Esta opção aceita vários valores (ex. `--dir=/foo --dir=/bar`)
InputOption::VALUE_NONE	Não aceitar a entrada para esta opção (ex. `--yell`)
InputOption::VALUE_REQUIRED	Este valor é obrigatório (ex. `--iterations=5`), a opção em si ainda é opcional
InputOption::VALUE_OPTIONAL	Esta opção pode ou não ter um valor (ex. `yell` ou `yell=loud`)

Você pode combinar VALUE_IS_ARRAY com VALUE_REQUIRED ou VALUE_OPTIONAL como abaixo:

```
$this
    // ...
    ->addOption('iterations', null, InputOption::VALUE_REQUIRED | InputOption::VALUE_IS_ARRAY, 'How r
```

Perguntando ao usuário informações

Ao criar comandos, você tem a possibilidade de coletar mais informações do usuário, fazendo-lhe perguntas. Por exemplo, suponha que você queira confirmar uma ação antes de executá-la. Adicione o seguinte ao seu comando:

```
$dialog = $this->getHelperSet()->get('dialog');
if (!$dialog->askConfirmation($output, '<question>Continue with this action?</question>', false)) {
    return;
}
```

Neste caso, o usuário será perguntado "Continue with this action", e, a menos que ele responda com `y`, a tarefa não irá executar. O terceiro argumento para `askConfirmation` é o valor padrão para retornar se o usuário não informar nenhuma entrada.

Você também pode fazer perguntas com mais do que uma simples resposta sim/não. Por exemplo, se você precisa saber o nome de alguma coisa, você pode fazer o seguinte:

```
$dialog = $this->getHelperSet()->get('dialog');
$name = $dialog->ask($output, 'Please enter the name of the widget', 'foo');
```

Testando Comandos

O Symfony2 fornece várias ferramentas para ajudar a testar os seus comandos. A mais útil é a classe `CommandTester`. Ela usa classes de entrada e saída especiais para facilitar o teste sem um console real:

```
use Symfony\Component\Console\Application;
use Symfony\Component\Console\Tester\CommandTester;
use Acme\DemoBundle\Command\GreetCommand;

class ListCommandTest extends \PHPUnit_Framework_TestCase
{
    public function testExecute()
    {
        $application = new Application();
        $application->add(new GreetCommand());

        $command = $application->find('demo:greet');
        $commandTester = new CommandTester($command);
        $commandTester->execute(array('command' => $command->getName()));

        $this->assertRegExp('/.../', $commandTester->getDisplay());

        // ...
    }
}
```

O método `getDisplay()` retorna o que teria sido exibido durante uma chamada normal ao console.

Você pode testar o envio de argumentos e opções para o comando, passando-os como um array para o método `getDisplay()`

```
use Symfony\Component\Console\Application;
use Symfony\Component\Console\Tester\CommandTester;
use Acme\DemoBundle\Command\GreetCommand;

class ListCommandTest extends \PHPUnit_Framework_TestCase
{
    // ...

    public function testNameIsOutput()
    {
        $application = new Application();
        $application->add(new GreetCommand());

        $command = $application->find('demo:greet');
        $commandTester = new CommandTester($command);
        $commandTester->execute(
            array('command' => $command->getName(), 'name' => 'Fabien')
        );

        $this->assertRegExp('/Fabien/', $commandTester->getDisplay());
    }
}
```

Dica: Você também pode testar uma aplicação de console inteira usando `ApplicationTester`.

Chamando um comando existente

Se um comando depende que outro seja executado antes dele, em vez de pedir ao usuário para lembrar a ordem de execução, você mesmo pode chamá-lo diretamente. Isso também é útil se você quiser criar um comando "meta" que apenas executa vários outros comandos (por exemplo, todos os comandos que precisam ser executados quando código do projeto mudou nos servidores de produção: limpar o cache, gerar proxies do Doctrine2, realizar o dump dos assets do Assetic, ...).

Chamar um comando a partir de outro é simples:

```
protected function execute(InputInterface $input, OutputInterface $output)
{
    $command = $this->getApplication()->find('demo:greet');

    $arguments = array(
        'command' => 'demo:greet',
        'name'    => 'Fabien',
        '--yell'  => true,
    );

    $input = new ArrayInput($arguments);
    $returnCode = $command->run($input, $output);

    // ...
}
```

Primeiro, você `find()` o comando que você deseja executar, passando o nome de comando.

Em seguida, você precisa cria um novo `ArrayInput` com os argumentos e opções que você deseja passar para o comando.

Eventualmente, chamando o método `run()` efetivamente executa o comando e retorna o código retornado pelo comando (valor de retorno do método `execute()` do comando).

Nota: Na maioria das vezes, chamando um comando a partir de código que não é executado na linha de comando não é uma boa idéia por várias razões. Primeiro, a saída do comando é otimizada para o console. Mas, mais importante, você pode pensar em um comando como sendo um controlador, que deve usar o modelo para fazer algo e exibir informações para o usuário. Assim, em vez de chamar um comando pela web, refatore o seu código e mova a lógica para um nova classe apropriada.

- **Class Loader**
  - O Componente ClassLoader
- Console
  - O Componente Console

Leia a documentação sobre os Componentes.

Documentos de Referência

Obtenha respostas rapidamente com os documentos de referência:

5.1 Documentos de Referência

5.1.1 Referência de Configuração

- *YAML*

```
monolog:
    handlers:

        # Examples:
        syslog:
            type:               stream
            path:               /var/log/symfony.log
            level:              ERROR
            bubble:             false
            formatter:          my_formatter
            processors:
                - some_callable
        main:
            type:               fingerscrossed
            action_level:       WARNING
            buffer_size:        30
            handler:            custom
        custom:
            type:               service
            id:                 my_handler

        # Default options and values for some "my_custom_handler"
        my_custom_handler:
            type:                ~ # Required
            id:                  ~
            priority:            0
            level:               DEBUG
            bubble:              true
            path:                %kernel.logs_dir%/%kernel.environment%.log
            ident:               false
            facility:            user
            max_files:           0
```

```
                action_level:         WARNING
                activation_strategy:  ~
                stop_buffering:       true
                buffer_size:          0
                handler:              ~
                members:              []
                channels:
                    type:     ~
                    elements: ~
                from_email:           ~
                to_email:             ~
                subject:              ~
                email_prototype:
                    id:                   ~ # Required (when the email_prototype is used)
                    factory-method:       ~
                channels:
                    type:                 ~
                    elements:             []
                formatter:            ~
```

- *XML*

```
    <container xmlns="http://symfony.com/schema/dic/services"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:monolog="http://symfony.com/schema/dic/monolog"
        xsi:schemaLocation="http://symfony.com/schema/dic/services http://symfony.com/schema/dic/ser
                            http://symfony.com/schema/dic/monolog http://symfony.com/schema/dic/monc

        <monolog:config>
            <monolog:handler
                name="syslog"
                type="stream"
                path="/var/log/symfony.log"
                level="error"
                bubble="false"
                formatter="my_formatter"
            />
            <monolog:handler
                name="main"
                type="fingerscrossed"
                action-level="warning"
                handler="custom"
            />
            <monolog:handler
                name="custom"
                type="service"
                id="my_handler"
            />
        </monolog:config>
    </container>
```

Nota: Quando o profiler é ativado, um handler é adicionado para armazenar as mensagens dos logs' no profiler. O Profiler usa o nome "debug", então, ele é reservado e não pode ser utilizado na configuração.

5.1.2 Referência para Tipos de Form

Um form é composto por *campos*, cada um deles é construido com a ajuda de um campo do *tipo* (ex: um tipo `text`, tipo `choice`, etc). Symfony2 tem por padrão uma vasta lista de tipos de campos que podem ser usados em sua aplicação.

Tipos de campos suportados

Os tipos de campo seguintes estão disponíveis nativamente no Symfony2:

5.1.3 Referência das Constraints de Validação

O Validator é projetado para validar objetos comparando com *constraints*. Na vida real, uma constraint poderia ser: "O bolo não deve ser queimado". No Symfony2, as constraints são semelhantes: Elas são afirmações de que uma condição é verdadeira.

Constraints Suportadas

As seguintes constraints estão disponíveis nativamente no Symfony2:

5.1.4 Requisitos para o funcionamento do Symfony2

Para funcionar o Symfony2, seu sistema precisa seguir uma lista de requisitos. Você pode facilmente verificar se o sistema passa por todos os requisitos executando o `web/config.php` em sua distribuição Symfony. Uma vez que a CLI geralmente utiliza um arquivo de configuração `php.ini` diferente, também é uma boa idéia verificar as necessidades a partir da linha de comando via:

```
php app/check.php
```

Abaixo está a lista de requisitos obrigatórios e opcionais.

Obrigatório

- Você precisa ter pelo menos a versão 5.3.3 do PHP
- JSON precisa estar habilitado
- ctype precisa estar habilitado
- Seu PHP.ini precisa ter o date.timezone definido

Opcional

- Você precisa ter o módulo PHP-XML instalado
- Você precisa ter pelo menos a versão 2.6.21 da libxml
- PHP tokenizer precisa estar habilitado
- As funções da mbstring precisam estar habilitadas
- iconv precisa estar habilitado
- POSIX precisa estar habilitado (apenas no *nix)

- Intl precisa estar instalado com ICU 4+
- APC 3.0.17+ (ou outro apcode cache precisa estar instalado)
- Configurações recomendadas no PHP.ini
  - `short_open_tag = Off`
  - `magic_quotes_gpc = Off`
  - `register_globals = Off`
  - `session.autostart = Off`

Doctrine

Se você quer usar o Doctrine, você precisará ter a PDO instalada. Além disso, você precisa ter o driver PDO para o servidor de banco de dados que você deseja usar.

- **Opções de Configuração**:

  Já imaginou quais opções de configuração estão disponíveis para você em arquivos como o `app/config/config.yml`? Nesta seção, todas as configurações disponíveis serão separadas por chaves (ex.: `framework`) que definem cada seção possível de sua configuração do Symfony2.

  - monolog
- *Outras Áreas**
  - Requisitos para o funcionamento do Symfony2
- **Opções de Configuração**:

  Já imaginou quais opções de configuração estão disponíveis para você em arquivos como o `app/config/config.yml`? Nesta seção, todas as configurações disponíveis serão separadas por chaves (ex.: `framework`) que definem cada seção possível de sua configuração do Symfony2.

  - monolog
- *Outras Áreas**
  - Requisitos para o funcionamento do Symfony2

CAPÍTULO 6

Bundles

A Edição Standard do Symfony vem com alguns bundles. Saiba mais sobre eles:

6.1 Bundles da Edição Standard do Symfony

- `SensioFrameworkExtraBundle`
- `SensioGeneratorBundle`
- JMSSecurityExtraBundle
- JMSDiExtraBundle
- `DoctrineFixturesBundle`
- `DoctrineMigrationsBundle`
- `DoctrineMongoDBBundle`
- `SensioFrameworkExtraBundle`
- `SensioGeneratorBundle`
- JMSSecurityExtraBundle
- JMSDiExtraBundle
- `DoctrineFixturesBundle`
- `DoctrineMigrationsBundle`
- `DoctrineMongoDBBundle`

CAPÍTULO 7

Contribuindo

Contribua com o Symfony2:

7.1 Contribuindo

7.1.1 Contribuindo com o Código

Reportando um Bug

Sempre que você encontrar um bug no Symfony2, pedimos gentilmente para reportá-lo. Isso ajuda-nos a tornar o Symfony2 melhor.

> **Cuidado:** Se você acredita ter encontrado um problema de segurança, por favor, use o `procedimento` especial em seu lugar.

Antes de submeter um bug:

- Verifique a documentação oficial para certificar-se de que não está fazendo mau uso do framework;
- Peça ajuda na **'lista de discussão dos usuários'_**, no **'fórum'_**, ou no canal IRC do #symfony se você não tiver certeza se o problema é realmente um bug.

Se o seu problema definitivamente parece um bug, relate-o usando o tracker oficial de bug e siga algumas regras básicas:

- Use o campo de título para descrever claramente o problema;
- Descreva os passos necessários para reproduzir o bug com exemplos curtos de código (fornecer um teste unitário que ilustra o bug é melhor);
- Forneça o máximo de detalhes possível sobre o seu ambiente (SO, versão do PHP, versão do Symfony, extensões habilitadas, ...);
- *(opcional)* Anexe um `patch`.

Padrões de Codificação

Para contribuir com código para o Symfony2, você deve seguir seus padrões de codificação. Para encurtar a história, esta é a regra de ouro: **Imite o código existente no Symfony2**. Muitos Bundles e bilbiotecas usados pelo Symfony2 também seguem as mesmas regras, e você também deveria.

Lembre-se que a principal vantagem de padrões é que cada pedaço de código parece familiar, não é sobre isso ou aquilo ser mais legível.

Como uma imagem — ou código — diz mais que mil palavras, aqui está um exemplo curto, contendo a maior parte dos aspectos descrito abaixo.

```
<?php

/*
 * This file is part of the Symfony package.
 *
 * (c) Fabien Potencier <fabien@symfony.com>
 *
 * For the full copyright and license information, please view the LICENSE
 * file that was distributed with this source code.
 */

namespace Acme;

class Foo
{
    const SOME_CONST = 42;

    private $foo;

    /**
     * @param string $dummy Some argument description
     */
    public function __construct($dummy)
    {
        $this->foo = $this->transform($dummy);
    }

    /**
     * @param string $dummy Some argument description
     * @return string|null Transformed input
     */
    private function transform($dummy)
    {
        if (true === $dummy) {
            return;
        }
        if ('string' === $dummy) {
            $dummy = substr($dummy, 0, 5);
        }

        return $dummy;
    }
}
```

Estrutura

- Nunca use *short tags* (<?);
- Não termine arquivos de classe com o a tag ?> de costume;
- Indentação é feita em passos de 4 espaços (tabulações nunca são permitidas);
- Use o caractere de nova linha (LF ou *0x0A*) para encerrar linhas;

- Coloque um único espaço depois de cada vírgula;
- Não coloque espaços depois de abrir parênteses ou antes de fechá-los;
- Coloque um único espaço em volta de operadores (==, *&&*, ...);
- Coloque um único espaço antes de abrir parênteses de uma palavra de controle (*if*, *else*, *for*, *while*, ...);
- Coloque uma linha em braco antes de uma declaração *return*, a não ser que a declaração esteja sozinha dentro de um bloco (como dentro de um *if*);
- Não acrescente espaços no final das linhas;
- Use chaves para indicar o corpo de estruturas de controle, não importando o númedo de declarações que ele contém;
- Coloque chaves nas suas prórpias linhas na declaração de classes, métodos e funções;
- Separe as declarações de estruturas de controle (*if*, *else*, ...) e suas chaves de abertura com um único espaço e nenhuma linha em branco;
- Declare explicitamente a visibilidade de classes, métodos e propriedades. O uso de *var* é proibido;
- Use as constantes nativas do PHP em caixa baixa: *false*, *true* e *null*. O mesmo vale para *array()*;
- Use caixa alta para constantes, com as palavras separadas por _;
- Defina uma classe por arquivo;
- Declare as propriedades da classe antes dos métodos;
- Declare métodos públicos primeiro, depois os protegidos e, finalmente, os privados.

Padrões de nomeação

- Use camelCase, não underscores (_), para variáveis, funções e métodos;
- Use underscores para nomes de opções, argumentos e parâmetros;
- Use namespaces em todas as classes;
- Sufixe nomes de interface com *Interface*;
- Use caracteres alfanuméricos e underscores para nomes de arquivos;
- Não se esqueça de ver no documento mais explicativo `conventions` para considerações de nomeação mais subjetivas.

Documentação

- Insira blocks PHPDoc para todas as classes, métodos e funções;
- Omita a tag *@return* se o método não retorna nada;
- As anotações *@package* e *@subpackage* não são usadas.

Licença

- Symfony é distribuido sob a licença MIT, e o bloco de licença deve estar

presente no topo de todo arquivo PHP, antes do namespace.

7.1.2 Contribuindo com a Comunidade

Outros Recursos

Para acompanhar o que está acontecendo na comunidade você pode achar úteis estes recursos adicionais:

- Lista dos pull requests abertos
- Lista dos commits recentes
- Lista das melhorias e bugs abertos
- Lista dos bundles open source
- **Código**:
  - Bugs
  - Padrões de Codificação
- **Comunidade**:
  - Outros Recursos
- **Código**:
  - Bugs
  - Padrões de Codificação
- **Comunidade**:
  - Outros Recursos

A

B

C

P

R

S

T

V