symfony-docs-pt-BR Documentation

Versão stable

30/12/2015

Sumário

CAPÍTULO 1

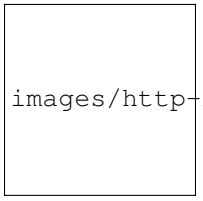
Fundamentos de Symfony e HTTP

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.

1.1 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:



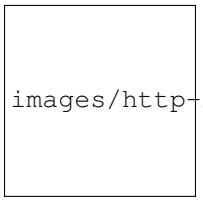
Apesar da linguagem real ser um pouco mais formal, ainda assim ela é bastante simples. HTTP é o termo usado para descrever essa linguagem simples baseada em texto. Não importa como você desenvolva para a web, o objetivo do seu servidor *sempre* será entender simples requisições de texto e enviar simples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

1.1.1 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:

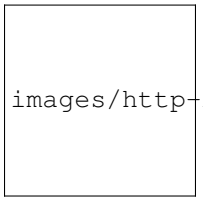
```
DELETE /blog/15 HTTP/1.1
```

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

1.1.2 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.

1.1.3 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.

1.2 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
<?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
```

1.3 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 `Symfony\Component\HttpFoundation\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.

1.4 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.

1.4.1 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.

1.4.2 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.

1.4.3 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. 1.1: 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.

1.4.4 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.

1.5 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.

1.5.1 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.

1.5.2 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.

CAPÍTULO 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.

2.1 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.

2.1.1 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).

2.1.2 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.

2.1.3 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.

2.2 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...

2.3 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.

2.3.1 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
<?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`:

```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?

2.3.2 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 `Symfony\Component\HttpFoundation\Request` e `Symfony\Component\HttpFoundation\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.

2.3.3 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*.

2.3.4 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

2.4 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`.

2.5 Aprenda mais no Cookbook

- `/cookbook/templating/PHP`
- `/cookbook/controller/service`

CAPÍTULO 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*.

3.1 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:

3.1.1 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
            ...
```

3.1.2 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 `/cookbook/configuration/override_dir_structure` para mais informações.

3.1.3 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.

3.1.4 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!

images/quick_tour/welcome.jpg

3.2 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.

3.3 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 /cookbook/workflow/new_project_git.

3.3.1 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.

CAPÍTULO 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 arbritária 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.

4.1 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.

4.2 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`.

4.3 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:

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`.

4.3.1 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:

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.

4.3.2 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); // ...
    }
```

4.4 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 `Symfony\Bundle\FrameworkBundle\Controller\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`.

4.5 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.

4.5.1 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'));
```

4.5.2 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',
));
```

4.5.3 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));
```

4.5.4 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 `/book/service_container`.

4.6 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 "`/cookbook/controller/error_pages`" no cookbook.

4.7 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.

4.7.1 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`:

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.

4.8 O Objeto Response

O único requisito de um controlador é retornar um objeto `Response`. A classe `Symfony\Component\HttpFoundation\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 `Symfony\Component\HttpFoundation\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`.

4.9 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.

4.10 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.

4.11 Saiba mais no Cookbook

- `/cookbook/controller/error_pages`
- `/cookbook/controller/service`

CAPÍTULO 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

5.1 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:

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.

5.2 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. 5.1: A camada de roteamento é uma ferramenta que traduza a URL recebida em um controlador específico para executar.

5.3 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:

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.

5.3.1 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:

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*.

5.3.2 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":

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`.

5.3.3 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:

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:

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`:

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

5.3.4 Adicionando Requisitos

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

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:

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 correponder 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:

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*

5.3.5 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:

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 requisiçõs 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`.

5.3.6 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:

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.

5.3.7 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.

5.4 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 `/cookbook/controller/service`).

5.5 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.

5.6 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:

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:

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

5.6.1 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}`:

A string `/admin` irá agora ser prefixada ao padrão de cada rota carregada do novo recurso de roteamento.

5.7 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
```

5.8 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 **:method:'Symfony\\Component\\Routing\\Router::match'** e

:method:‘Symfony\\Component\\Routing\\Router::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.

5.8.1 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');
```

5.8.2 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
```

5.8.3 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:

URLs absolutas também podem ser geradas.

5.9 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.

5.10 Aprenda mais do Cookbook

- `/cookbook/routing/scheme`

CAPÍTULO 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.

6.1 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>
```

6.1.1 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.

6.2 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:

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:

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 %}
```

6.3 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*.

6.3.1 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.

6.4 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.

6.4.1 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.

Incluir este template de qualquer outro template é fácil:

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}`.

6.4.2 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:

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**):

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.

6.4.3 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`:

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:

6.4.4 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:

Para vincular à página, apenas use a função Twig `path` e refira para a rota:

Como esperado, isso irá gerar a URL `/`. Vamos ver como isso irá funcionar com uma rota mais complicada:

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:

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>
```

6.4.5 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:

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.

6.5 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 `/cookbook/assetic/asset_management`.

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`.

6.6 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:

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).

6.7 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`.

6.7.1 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` .

6.8 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.

6.9 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:

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.

6.9.1 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.

6.9.2 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') ?>';
```

6.10 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:

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`.

6.11 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
```

6.12 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:

6.13 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.

6.14 Aprenda mais do Cookbook

- `/cookbook/templating/PHP`
- `/cookbook/controller/error_pages`

CAPÍTULO 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 "`/cookbook/doctrine/dbal`".

Você também pode persistir dados no MongoDB usando a biblioteca Doctrine ODM. Para mais informações, leia a documentação "`/bundles/DoctrineMongoDBBundle/index`".

7.1 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
```

7.1.1 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 `/cookbook/configuration/external_parameters`.

Agora que o Doctrine sabe sobre seu banco, pode deixar que ele faça a criação dele para você:

```
php app/console doctrine:database:create
```

7.1.2 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(255
```

7.1.3 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.

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

7.1.4 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.

7.1.5 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.

7.1.6 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:

```
1  // src/Acme/StoreBundle/Controller/DefaultController.php
2  use Acme\StoreBundle\Entity\Product;
3  use Symfony\Component\HttpFoundation\Response;
4  // ...
5
6  public function createAction()
7  {
8      $product = new Product();
9      $product->setName('A Foo Bar');
10     $product->setPrice('19.99');
11     $product->setDescription('Lorem ipsum dolor');
12
13     $em = $this->getDoctrine()->getManager();
14     $em->persist($product);
15     $em->flush();
16
17     return new Response('Created product id '.$product->getId());
18 }
```

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`.

7.1.7 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.

7.1.8 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.

7.1.9 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.

7.2 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.

7.2.1 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',
))
```

7.2.2 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.

7.2.3 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.

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()`.

7.3 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.

7.3.1 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`.

7.3.2 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.

7.3.3 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".

7.3.4 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();

    // ...
}
```

7.3.5 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.

7.4 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`.

7.5 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):

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 `/cookbook/doctrine/event_listeners_subscribers`.

7.6 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`.

7.7 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.

7.7.1 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.

7.8 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 `/cookbook/doctrine/reverse_engineering`.
- `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.

7.9 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`
- `/cookbook/doctrine/common_extensions`

CAPÍTULO 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.

8.1 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`.

8.2 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/
```

8.3 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.

8.3.1 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 `Symfony\Component\DomCrawler\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',
    )
);
```

8.4 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();
```

8.4.1 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();
```

8.4.2 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();
```

8.4.3 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

8.4.4 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.

8.4.5 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();
```

8.5 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.

8.5.1 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)`

8.5.2 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');
});
```

8.5.3 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);
```

8.5.4 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).

8.6 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`:

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.

8.6.1 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>
```

8.7 Aprenda mais no Cookbook

- `/cookbook/testing/http_authentication`
- `/cookbook/testing/insulating_clients`
- `/cookbook/testing/profiling`

CAPÍTULO 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.

9.1 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:

Dica:

Propriedades protected e private podem também ser validadas, bem como os métodos

"getter" (veja *validator-constraint-targets*

9.1.1 Usando o serviço `validator`

Próximo passo, para realmente validar um objeto''Author'', use o método `validate` no serviço `validator` (classe `Symfony\Component\Validator\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:

Nota: Cada erro de validação (chamado de "constraint violation"), é representado por um objeto `Symfony\Component\Validator\ConstraintViolation`.

9.1.2 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.

9.2 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:

9.3 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.

9.3.1 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 "`/cookbook/validation/custom_constraint`".

9.3.2 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”: 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.

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).

9.4 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.

9.4.1 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.

9.4.2 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’‘:

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.

9.4.3 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.

9.5 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:

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*.

9.6 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 `Symfony\Component\Validator\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.

9.7 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.

9.8 Aprenda mais do Cookbook

- `/cookbook/validation/custom_constraint`

CAPÍTULO 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.

10.1 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:

```
// 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.

10.1.1 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)*).

10.1.2 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:

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()`).

10.1.3 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.

10.2 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.

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`.

10.2.1 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.

10.2.2 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');
            }
        },
    ));
}
```

10.3 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 "`/cookbook/form/create_custom_field_type`" do cookbook.

10.3.1 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.

10.4 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).

10.4.1 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))
```

10.5 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:

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`:

10.5.1 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`:

Se a label auto-gerada para um campo não estiver correta, você pode especificá-la explicitamente:

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:

10.5.2 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.

10.6 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();
```

10.7 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).

10.8 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.

10.8.1 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`:

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.

10.8.2 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 "`/cookbook/form/form_collections`" e na collection referência dos tipos de campo.

10.9 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:

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:

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 `/cookbook/form/form_customization`.

10.9.1 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`).

10.9.2 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.

10.9.3 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:

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:

Qualquer fragmento dentro do diretório `Acme/TaskBundle/Resources/views/Form` agora será usado globalmente para definir a saída do formulário.

10.10 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.

10.11 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.

10.11.1 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.

10.12 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.

10.13 Aprenda mais no Cookbook

- `/cookbook/doctrine/file_uploads`

- File Field Reference
- Creating Custom Field Types
- /cookbook/form/form_customization
- /cookbook/form/dynamic_form_generation
- /cookbook/form/data_transformers

CAPÍTULO 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.

11.1 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).

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.

11.2 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.

11.2.1 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.

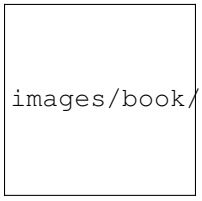
Um firewall será ativado quando a URL requisitada corresponda ao `padrão de caracteres` da expressão regular configurada na configuração de segurança. Neste exemplo, o `padrão de caracteres` (`^/`) corresponde a qualquer solicitação. O fato do firewall ser ativado *não* significa, porém, que a janela 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.

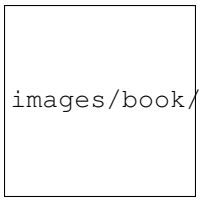
11.2.2 Controles de acesso (Autorização)

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.

11.3 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:

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:

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`):

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:

Dica: A variável `error` passada para a template é uma instância de `Symfony\Component\Security\Core\Exception\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 `/cookbook/security/form_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:
Removendo o controle de acesso para a URL `/login` resolve o problema:
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:
3. Tenha certeza que ‘‘/login_check‘‘ está protegida por um firewall
Certifique-se que a URL indicada em `check_path` (no caso, `/login_check`) esteja protegida por um firewall que está utilizando seu formulário de login (neste exemplo, um único firewall filtra *todas* as URLs, incluindo `/login_check`). Se `/login_check` não estiver atrás de nenhum firewall, uma exceção será gerada `Unable to find the controller for path "/login_check"`.
4. Múltiplos firewalls não compartilham o mesmo contexto de segurança
Se estiver utilizando múltiplos firewalls e se autenticar em um firewall, você *não* estará autenticado nos outros firewalls automaticamente. Firewalls diferentes funcionam como sistemas de segurança diferente. Isto acontece por que para a maioria dos aplicativos ter somente um firewall é o suficiente.

11.4 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.

11.4.1 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.

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 `Symfony\Component\Security\Core\Exception\AccessDeniedException` , que você pode tratar e transformar em uma apresentável página de "Acesso Negado" para o usuário. Veja `/cookbook/controller/error_pages` 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`.

11.4.2 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:

11.4.3 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:

11.4.4 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.

11.4.5 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`.

11.4.6 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: `/cookbook/security/acl`.

11.5 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.

11.5.1 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.

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 (`Symfony\Component\Security\Core\User\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 `Symfony\Component\Security\Core\User\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 `Symfony\Component\Security\Core\User\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`:

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 `/cookbook/security/custom_provider`.

11.5.2 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:

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:

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:

```
$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);
```

11.5.3 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.

11.5.4 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:

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:

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:

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`.

11.6 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_`.

11.6.1 Hierarquia de Perfis

Ao invés de associar muitos perfis aos usuários, você pode defined regras de herança ao criar uma hierarquia de perfis:

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`).

11.7 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:

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:

Note que você *não* precisará implementar o controller para a URL `/logout` já que o firewall cuida disso. Você *deve*, entretante, 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.

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`.

11.8 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:

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).

11.9 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.

11.10 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:

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`:

11.11 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):

Nota: Se utiliza formulário de login, o Symfony2 criará um cookie mesmo se você definir `stateless` como `true`.

11.12 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.

11.13 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

CAPÍTULO 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.

12.1 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.

12.2 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.

12.2.1 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.

12.2.2 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.

12.3 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*.

12.3.1 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:

```
$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);
```

12.3.2 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.

12.3.3 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.

12.3.4 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).

12.4 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.

12.4.1 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`.

12.4.2 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."

12.4.3 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
```

12.4.4 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`.

12.4.5 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 **:method:'Symfony\\Component\\HttpFoundation\\Response::setETag'**.

12.4.6 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.

12.4.7 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 **:method:'Symfony\\Component\\HttpFoundation\\Response::setNotModified'**).

12.4.8 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.

12.4.9 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.

12.4.10 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,
));
```

12.5 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.

12.5.1 Usando ESI no Symfony2

Primeiro, para usar ESI, tenha certeza de ter feito sua habilitação na configuração da sua aplicação:

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:

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`:

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.

12.6 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.

12.7 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.

12.8 Learn more from the Cookbook

- `/cookbook/cache/varnish`

CAPÍTULO 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.

13.1 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:

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*).

13.2 Tradução básica

Tradução do texto é feita done através do serviço `translator` (`Symfony\Component\Translation\Translator`). Para traduzir um bloco de texto (chamado de *messagem*), use o método **:method:'Symfony\\Component\\Translation\\Translator::trans'**. Suponhamos, por exemplo, que estamos traduzindo uma simples mensagem de dentro do controller:

```
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:

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`.

13.2.1 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).

13.2.2 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:

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.

13.3 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
```

13.3.1 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 `Symfony\Component\Translation\Loader\LoaderInterface`. Veja `Custom Translation Loaders` abaixo para aprender como registrar carregadores personalizados.

13.3.2 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):

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:
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:

13.4 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.

13.5 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.

13.5.1 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:

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.

13.5.2 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`:

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.

13.6 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: **:method:‘Symfony\\Component\\Translation\\Translator::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.

13.6.1 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 `Symfony\Component\Translation\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.

13.7 Traduções em Templates

A maior parte do tempo, traduções ocorrem em templates. Symfony2 providencia suporte nativo para ambos os templates PHP e Twig.

13.7.1 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).

13.7.2 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)
) ?>
```

13.8 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',
);
```

13.9 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.

13.10 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 **:method:‘Symfony\\Component\\Translation\\Translator::trans‘** ou **:method:‘Symfony\\Component\\Translation\\Translator::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.

CAPÍTULO 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.

14.1 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.

14.2 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.

14.3 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:

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.

14.4 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:

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.

14.4.1 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.

14.5 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.

14.5.1 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.

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.

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.

14.5.2 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`:

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 "`/cookbook/bundles/extension`" do cookbook.

14.6 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:

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.

14.6.1 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:

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".

14.7 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:

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)
{
    // ...
}
```

14.8 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:

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`.

14.9 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.

14.9.1 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:

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.

14.9.2 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.

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
```

14.9.3 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`.

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.

14.9.4 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:

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

14.10 Saiba mais

- `/components/dependency_injection/factories`
- `/components/dependency_injection/parentservices`
- `/cookbook/controller/service`

CAPÍTULO 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.

15.1 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.

15.1.1 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.

15.2 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 - `Symfony\Component\ClassLoader\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.

15.3 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`.

15.3.1 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.

CAPÍTULO 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

C

D

E

F

V