Symfony Documentation

Выпуск 2.0

Fabien Potencier

15 January 2012

Оглавление

Часть I

Краткий Обзор

Быстрый старт с Symfony2 *Краткий Обзор*:

Глава 1

Краткий тур

Быстрый старт с кратким туром по Symfony2:

- quick_tour/the_big_picture >
- quick_tour/the_view >
- quick_tour/the_controller >
- quick_tour/the_architecture

Часть II

Руководства

Погрузитесь в мир Symfony2 с помощью актуальных руководств:

Глава 2

Книга

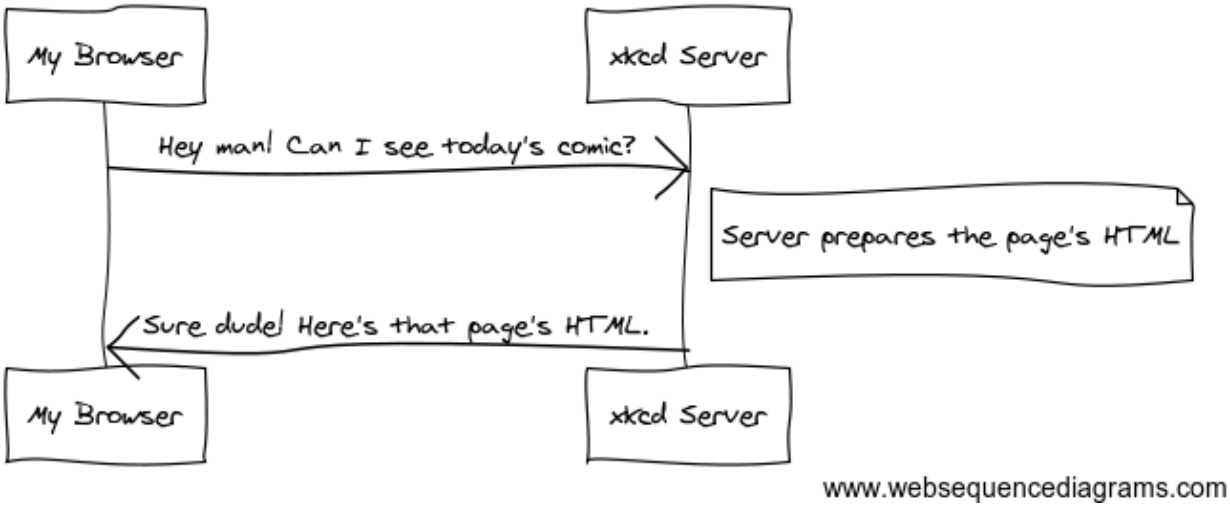
2.1 Symfony2 и основы HTTP

Поздравляем! Начав изучение Symfony2, вы встали на правильный путь, чтобы стать более *продуктивным*, *всесторонне развитым* и *популярным* веб-разработчиком (хотя последнее - на ваше усмотрение). Symfony2 создан, чтобы предоставлять базовые, низкоуровневые инструменты, позволяющие вам разрабатывать быстрее, создавать более надёжные приложения, но при этом быть в строне от вашего собственного пути. Symfony построен на лучших идеях, заимствованных из различных технологий: инструменты и концепции, которые вы готовитесь изучить - представлены усилиями тысяч и тысяч людей на протяжении многих лет. Другими словами, вы не только изучаете “Symfony”, вы изучаете основы web, лучшие практики разработки, а также способы использования многих замечательных PHP-библиотек в составе Symfony2 или не зависимо от него. Итак, приготовьтесь.

Следуя философии Symfony2, эта глава начинается с объяснения основной концепции, типичной для web-разработки: HTTP. Не зависимо от вашего опыта или любимого языка программирования, эта глава **обязательна к прочтению** всем.

2.1.1 HTTP это Просто

HTTP (Hypertext Transfer Protocol или просто Протокол Передачи Гипертекста) - это текстовый язык, позволяющий двум компьютерам обмениваться сообщениями друг с другом. Вот и всё! Например, когда мы хотим посмотреть новенький комикс xkcd, имеет место (примерно) такой диалог:



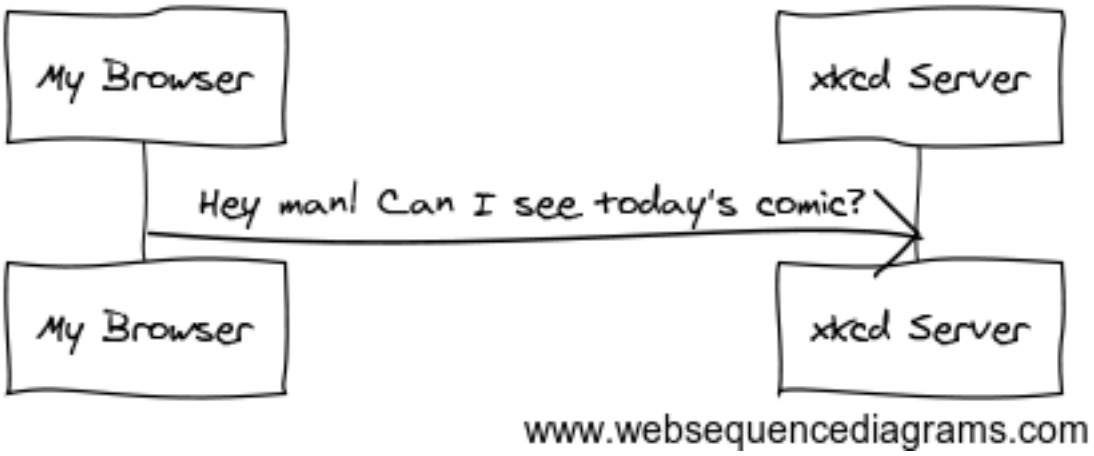
И пока используется реальный язык, хотя он и несколько более формальный, он остаётся предельно простым. HTTP - это термин, используемый для описания этого простого текстового языка. И не важно, как именно вы разрабатываете в web, целью вашего сервера *всегда* является понять простой текстовый запрос и вернуть простой текстовый ответ.

Symfony2 возвышается над этой реальностью. Что бы вы ни делали, HTTP - это то, что вы используете ежедневно. С помощью Symfony2 вы узнаете, как управлять им.

Шаг 1: Клиент отправляет запрос

Любой диалог в сети начинается с *запроса*. Запрос - это текстовое сообщение, создаваемое клиентом (например браузером или iPhone приложением и т.д.) в особом формате, также известном как HTTP. Клиент отправляет этот запрос серверу, и ожидает ответ.

Взгляните на первую часть взаимодействия (запрос) между браузером и веб-сервером xkcd:



На языке HTTP этот запрос будет выглядеть примерно так:

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

Это простое сообщение содержит *всю* необходимую информацию о том, какой именно ресурс запрашивает клиент. Первая строка HTTP запроса наиболее важна - она содержит 2 вещи: запрошенный URI и HTTP-метод.

URI (например `/`, `/contact`, и т.д.) - это уникальный адрес или место, которое определяет запрошенный клиентом ресурс. HTTP-метод (например `GET`) определяет, что именно вы хотите сделать с запрошенным ресурсом. HTTP методы это *глаголы* в запросе и они определяют несколько типичных путей, которыми вы можете взаимодействовать с запрошенным ресурсом:

GET	Получить ресурс с сервера
POST	Создать ресурс на сервере
PUT	Обновить ресурс на сервере
DELETE	Удалить ресурс с сервера

Запомнив эти типы HTTP-методов, вы можете представить себе, как будет выглядеть HTTP-запрос на удаление записи в блоге:

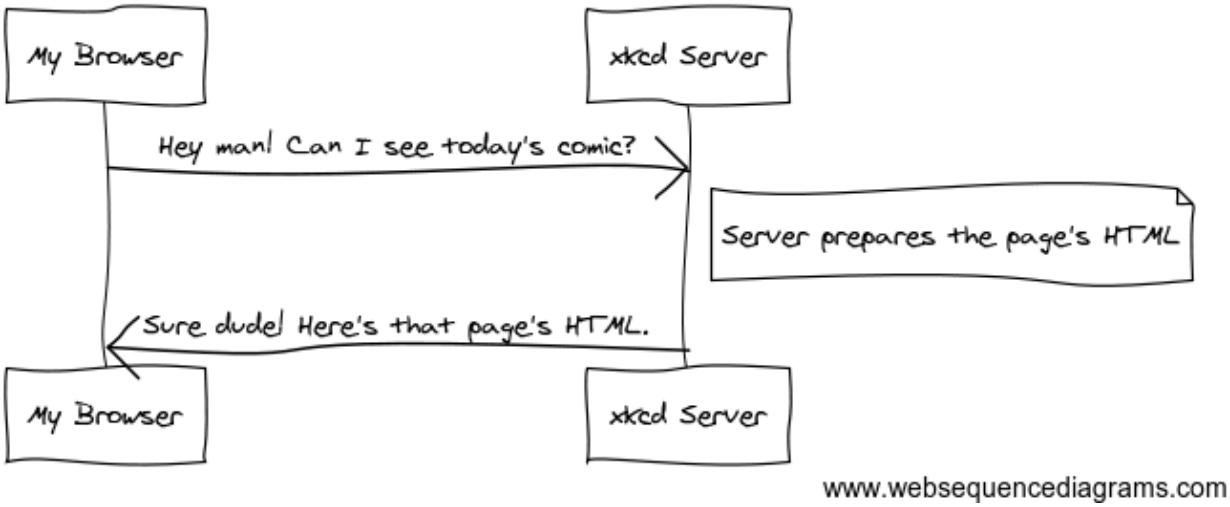
```
DELETE /blog/15 HTTP/1.1
```

Примечание: На самом деле всего существует девять HTTP-методов, определённых в спецификации протокола HTTP, но многие из них очень мало распространены или же ограниченно поддерживаются. К примеру, многие современные браузеры не поддерживают методы `PUT` и `DELETE`.

В дополнение к первой строке, HTTP-запрос всегда содержит несколько информационных строк, именуемых заголовками (headers). Заголовки могут предоставлять различную информацию, такую как запрошенный `Host`, форматы ответа, которые поддерживает клиент (`Accept`) и приложение, используемое клиентом для выполнения запроса (`User-Agent`). Существует также много других заголовков, перечень которых вы можете найти в Википедии на странице List of HTTP header fields.

Шаг 2: Сервер возвращает ответ

С того момента как сервер получил запрос, он точно знает, какой ресурс нужен клиенту (основываясь на URI) и что клиент хочет с этим ресурсом сделать - на основании HTTP-метода. Например, в случае GET-запроса, сервер подготовит запрошенный ресурс и возвратит его в виде HTTP-ответа. Рассмотрим ответ от web сервера xkcd:



Переведённый в формат HTTP, ответ, отправленный обратно в браузер, будет выглядеть примерно так:

```
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>
```

HTTP-ответ содержит запрошенный ресурс (в данном случае это HTML-код страницы), а также дополнительные данные о самом ответе. Первая строка особенно важна - она содержит HTTP статус-код (в данном случае 200). Статус-код сообщает о результате выполнения запроса, направляемом клиенту. Был ли запрос успешен? Была ли в ходе выполнения запроса ошибка? Одни статус-коды обозначают успешные запросы, другие - ошибки, третьи сообщают, что клиент должен выполнить что-либо (например перенаправление на другую страницу). Полный список вы можете найти странице List of HTTP status codes в Википедии.

Подобно запросу, HTTP-ответ содержит дополнительную информацию, называемую HTTP-заголовками. Например, важным заголовком HTTP-ответа является `Content-Type`. Тело одного и того же ресурса может быть возвращено во множестве различных форматов, включая HTML, XML или JSON. Заголовок `Content-Type` сообщает клиенту, какой именно формат используется в данном ответе.

Существует много различных заголовков, некоторые из них предоставляют большие возможности. Например, некоторые заголовки могут быть использованы для создания системы кэширования.

Запросы, Ответы и Web-разработка

Обмен запросами-ответами - это фундаментальный процесс, который движет все коммуникации во всемирной сети. И насколько важен этот процесс, настолько он прост.

Наиболее важным является следующий факт: вне зависимости от того, какой языка программирования вы используете, какое приложение создаёте (web, мобильное, JSON API) и даже какой философии следуете в разработке ПО, конечной целью приложения **всегда** будет приём и разбор запроса и создание соответствующего ответа.

Symfony спроектирован исходя из этих реалий.

Совет: Для того чтобы узнать больше про спецификацию HTTP, прочитайте оригинал HTTP 1.1 RFC или же HTTP Bis, который является инициативой по разъяснению оригинальной спецификации. Замечательный инструмент для проверки заголовков запроса и ответа при сёрфинге - это расширение для Firefox Live HTTP Headers.

2.1.2 Запросы и ответы в PHP

Как же вы обрабатываете “запрос” и создаете “ответ” при использовании PHP? На самом деле PHP немного абстрагирует вас от процесса:

```
<?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;
```

Как бы странно это ни звучало, но это крохотное приложение получает информацию из HTTP-запроса и использует её для создания HTTP-ответа. Вместо того, чтобы парсить необработанный HTTP-запрос, PHP подготавливает суперглобальные переменные, такие как `$_SERVER` и `$_GET`, которые содержат всю информацию о запросе. Аналогично, вместо того, чтобы возвращать текст ответа, форматированный по правилам HTTP, вы можете использовать функции `header()` для создания заголовков ответов и просто вывести на печать основной контент, который станет контентным блоком ответа. В заключении PHP создаст правильный HTTP-ответ и вернет его клиенту:

```
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
```

2.1.3 Запросы и ответы в Symfony

Symfony предоставляет альтернативу прямолинейному подходу из PHP посредством двух классов, которые позволяют взаимодействовать с HTTP-запросом и ответом самым простейшим способом. Класс `Symfony\Component\HttpFoundation\Request` - это простое объектно-ориентированное представление сообщения HTTP-запроса. С его помощью вы имеете все данные из запроса "на кончиках пальцев":

```
<?php

use Symfony\Component\HttpFoundation\Request;

$request = Request::createFromGlobals();

// запрошенный URI (на пример /about) без query parameters
$request->getPathInfo();

// получаем GET и POST переменные соответственно
$request->query->get('foo');
$request->request->get('bar');

// получаем экземпляр UploadedFile определяемый идентификатором foo
$request->files->get('foo');

$request->getMethod();          // GET, POST, PUT, DELETE, HEAD
$request->getLanguages();       // массив языков, принимаемых клиентом
```

В качестве бонуса, класс `Request` выполняет большой объём работы в фоновом режиме, так что вам не придется заботиться о многих вещах. Например, метод `isSecure()` проверяет *три* различных значения в PHP, которые указывают, что пользователь подключается по защищенному протоколу (`https`).

Symfony также предоставляет класс `Response`: простое PHP-представление HTTP-ответа. Это позволяет вашему приложению использовать объектно-ориентированный интерфейс для конструирования ответа, который нужно вернуть клиенту:

```
<?php

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();
```

Если бы Symfony ничего вам не предлагала, вы всегда должны были бы иметь набор инструментов для того чтобы можно было просто и быстро получить доступ к информации из запроса и объектно-ориентированный интерфейс для создания ответа. Даже если вы освоите более мощные возможности в Symfony, всегда держите в голове, что цель вашего приложения всегда заключается в том, чтобы *интерпретировать запрос и создать соответствующий ответ, основываясь на логике вашего приложения*

Совет: Классы `Request` и `Response` являются частью самостоятельного компонента `HttpFoundation`. Этот компонент может быть использован независимо от Symfony и он также предоставляет классы для работы с сессиями и загрузки файлов.

2.1.4 Путешествие от Запроса до Ответа

Как и HTTP-протокол, объекты `Request` и `Response` достаточно просты. Самая сложная часть создания приложения заключается в написании процессов, которые происходят между получением запроса и отправкой ответа. Другими словами, реальная работа заключается в написании кода, который интерпретирует информацию запроса и создает ответ (логика приложения).

Ваше приложение может иметь много функций, например, отправлять email'ы, обрабатывать отправленные формы, сохранять что-то в базу данных, отображать HTML-страницы и защищать контент правилами безопасности. Как управляться со всем этим и чтобы при этом код оставался хорошо организованным и поддерживаемым?

Symfony создана специально для решения этих проблем, значит, вам не придется их решать.

Фронт-контроллер

Традиционно приложения создавались таким образом, чтобы каждая "страница" имела свой собственный файл:

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

При таком подходе имеется целый ряд проблем, включая жёсткие URLы (что если вам потребуется изменить `blog.php` на `news.php` и при этом сохранить все ваши ссылки?),

а также необходимость вручную включать в каждый файл кучу файлов, включающих безопасность, работу с базами данных.

Много более удачным является подход с использованием *front controller*, единственного PHP-файла, который отвечает за каждый запрос к вашему приложению. Например:

`/index.php`	выполняет `index.php`
`/index.php/contact`	выполняет `index.php`
`/index.php/blog`	выполняет `index.php`

Совет: С использованием модуля `mod_rewrite` для Apache (или эквивалента для других web-серверов) URLы легко очистить от упоминания фронт-контроллера, т.е. останется лишь `/`, `/contact` и `/blog`.

Теперь, каждый запрос обрабатывается однообразно. Вместо того чтобы каждый URL соответствовал отдельному PHP-файлу - фронт-контроллер выполняется *всегда* и посредством маршрутизатора вызывает различные части вашего приложения, в зависимости от URL. Это решает многие проблемы, которые порождал традиционный подход. Практически все современные приложения используют этот подход, например WordPress.

Будьте организованы

Итак, мы внутри вашего фронт-контроллера. Но как мы узнаем, какая страница должна быть отображена и как её сформировать? В любом случае вам нужно проверить входящий URI и выполнить какую-то из частей вашего кода, в зависимости от этого значения. Это можно сделать быстро и весьма коряво:

```
<?php
// index.php

$request = Request::createFromGlobals();
$path = $request->getPathInfo(); // запрошенный URL

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();
```

Решить же эту проблему достаточно сложно. К счастью, Symfony создана *именно* для этого.

Как устроено Symfony приложение

Когда вы даёте возможность Symfony обрабатывать запросы, жизнь становится много проще. Symfony следует простому шаблону при обработке каждого запроса:

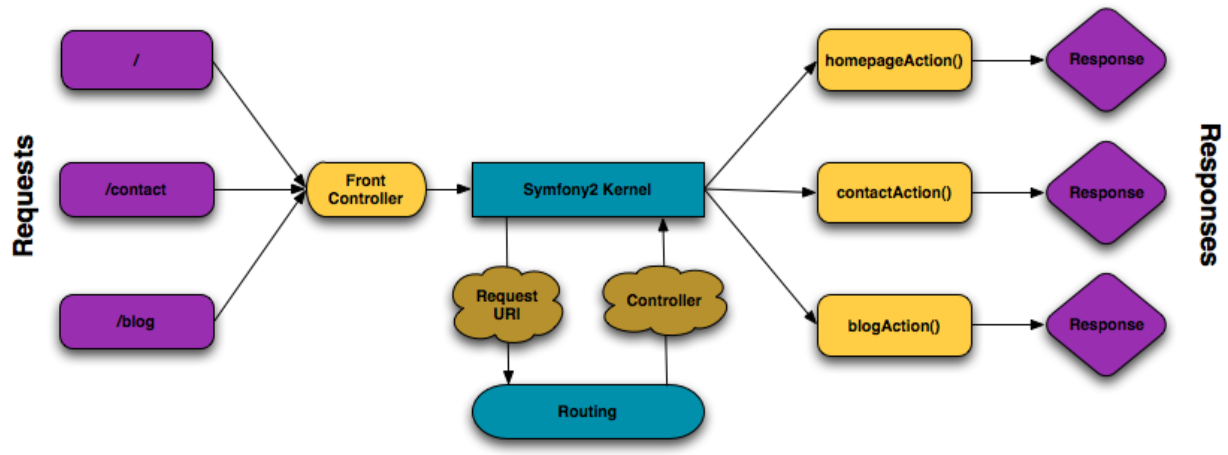


Рис. 2.1: Входящие запросы интерпретируются маршрутизатором и передаются в функцию-контроллер, которая возвращает объект `Response`.

Каждая "страница" вашего сайта должна быть определена в конфигурации маршрутизатора, чтобы распределять различные URL по различным PHP-функциям. Обязанность каждой такой функции, называемой *controller*, используя информацию из запроса - а также используя прочий инструментарий, доступный в Symfony, создать и вернуть объект `Response`. Другими словами, контроллер содержит *ваш* код: именно там вы должны превратить запрос в ответ.

Это не сложно! Давайте-ка взглянем:

- Каждый запрос обрабатывается фронт-контроллером;
- Система маршрутизации определяет, какую именно PHP-функцию необходимо выполнить, основываясь на информации из запроса и конфигурации маршрутизатора, которую вы создали;
- Вызывается необходимая функция, в которой написанный вами код создаёт и возвращает соответствующий логике приложения объект `Response`.

Symfony Request в действии

Не закапываясь глубоко в детали, давайте посмотрим на этот процесс в действии. Предположим, вы хотите добавить страницу `/contact` к вашему Symfony приложению. Во-первых, надо добавить конфигурацию маршрутизатора для `/contact` URI:

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

Примечание: Этот пример использует `YAML` для того чтобы определить конфигурацию маршрутизатора. Конфигурацию можно также задавать и в других форматах - таких как XML или PHP.

Когда кто-либо посещает страницу `/contact`, URI совпадает с маршрутом и указанный нами ранее контроллер выполняется. Как вы узнаете в из главы *Маршрутизация*, строка `AcmeDemoBundle:Main:contact` это короткая форма записи, которая указывает на особый метод `contactAction`, определённый в классе `MainController`:

```
<?php

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

В этом очень простом примере, контроллер создает объект `Response`, содержащий лишь простенький HTML-код "<h1>Contact us!</h1>". В главе *Контроллер*, вы узнаете, как контроллер может отображать шаблоны, позволяя "представлению" существовать раздельно от кода в файлах шаблонов. Это дает возможность сосредоточиться в контроллере на работе с базами данных, обработке отправленных пользователем данных или отправке email сообщений.

2.1.5 Symfony2: Создавайте приложение, а не инструменты.

Теперь вы знаете, что цель вашего приложения заключается в интерпретации входящих запросов и создании адекватного ситуации ответа. По мере роста приложения становится все труднее содержать свой код в порядке. Без сомнений, эта же задача будет повторяться снова и снова: сохранение данных в базу, отображение и повторное использование шаблонов, обработка форм, отправка emails, валидация данных, введённых пользователем и безопасность.

Хорошие новости заключаются в том, что эти проблемы не уникальны. Symfony предоставляет Фреймворк, полный инструментов, которые позволят вам создать ваше собственное приложение, а не ваши инструменты. При помощи Symfony2 вы использовать Фреймворк целиком или же только его часть.

Автономные библиотеки: *Компоненты* Symfony2

Что же собой представляет Symfony2? Прежде всего, Symfony2 - это коллекция более чем 20 независимых библиотек, которые могут быть использованы *в любом* PHP-проекте. Эти библиотеки, называемые *Symfony2 Components*, содержат полезные методы практически на любой случай жизни, не зависимо от того как именно ваш проект разрабатывается. Вот некоторые из них:

- HttpFoundation - Содержит классы `Request` и `Response`, а также классы для работы с сессиями и загрузкой файлов;
- Routing - мощная система маршрутизации, которая позволяет вам ставить в соответствие некоторому URI (например `/contact`) информацию о том, как этот запрос должен быть обработан (например вызвать метод `contactAction()`);
- Form - многофункциональный и гибкий фреймворк для создания форм обработки их сабмита;
- Validator - система, предназначенная для создания правил для данных и последующей валидации - соответствуют ли данные, отправленные пользователями этим правилам;
- ClassLoader - библиотека, позволяющая использовать PHP-классы без использования явного `require` для файлов, включающих требуемые классы.
- Templating - тулкит для рендеринга шаблонов, поддерживает наследование шаблонов (например, декорирование шаблонов при помощи родительского шаблона aka layout), а также прочие типичные для шаблонов операции (escaping, условия, циклы и т.д.);
- Security мощная библиотека для обеспечения всех типов безопасности внутри приложения;
- Translation - Фреймворк для поддержки переводов в вашем приложении.

Каждый из этих компонентов независим и может быть использован в *любом* PHP-проекте, не зависимо от Symfony2.

Комплексное решение: Symfony2 *Framework*

Ну так что же это *такое* - Symfony2 *Framework*? *Symfony2 Framework* это PHP библиотека, которая решает 2 различных задачи:

1. Предоставляет набор отобранных компонент (Symfony2 Components) и сторонних библиотек (например `Swiftmailer` для отправки почты);
2. Предоставляет возможности по конфигурированию всего этого добра и “клей”, который скрепляет все библиотеки в единое целое.

Цель фреймворка - интеграция независимых инструментов и обеспечение их совместной работы. Сам фреймворк представляет собой Symfony Bundle (плагин), который можно конфигурировать или даже заменить.

Symfony2 предоставляет замечательный набор инструментов для быстрой разработки web-приложений, ничего не навязывающий непосредственно вашему приложению. Разработчик может быстро приступить к разработке, используя дистрибутив Symfony2, который предоставляет скелетон с типовыми настройками. А для пытливых умов... у неба нет потолка!)

2.2 Symfony2 против чистого PHP

Почему использовать Symfony2 лучше, чем простой PHP файл, который можно просто открыть и писать код не задумываясь?

Если раньше вы никогда не пользовались PHP-фреймворками, не знакомы с философией Model-View-Controller (здесь и далее MVC) или же удивлены *суматохой* вокруг Symfony2, то эта глава создана специально для вас! Вместо того чтобы *рассказать* вам о том, что Symfony2 позволит разрабатывать быстрее и качественнее, чем при использовании чистого PHP, мы просто покажем вам это.

В этой главе, вы создадите простенькое приложение на чистом PHP и выполните его оптимизацию. Вы совершите своеобразное путешествие свозь время, наблюдая за развитием web-разработки на протяжении последних лет от плоского PHP до сегодняшнего уровня.

В конце концов, вы увидите, как Symfony2 поможет вам избавиться от рутинных задач и вернуть вам контроль над кодом.

2.2.1 Простой блог на чистом PHP

В этой главе вы создадите простое приложение - блог, используя лишь обычный PHP. Чтобы начать, создайте страницу, которая отображает записи в блоге, которые были сохранены в базе данных. Писать на чистом PHP проще простого:

```
<?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);
```

Такой код быстро пишется, также быстро выполняется, и, по мере роста вашего приложения, становится совершенно неподдерживаемым. Таким образом, тут имеется несколько проблем, которые требуется решить:

- **Нет обработчика ошибок**: А что, если подключение к базе данных отвалится?
- **Плохая организация кода**: По мере роста приложения, этот файл будет все больше и больше, в то же время поддерживать его будет всё сложнее и сложнее. Где вы должны будете разместить код, который обрабатывает отправку формы? Как вы будете проверять входные данные? А куда разместить код для отправки email'ов?
- **Сложность (а скорее даже невозможность) повторного использования кода**: Так как весь код располагается в одном файле, нет никакой возможности повторного использования любой части приложения для других страниц блога.

Примечание: Другая проблема, не упомянутая выше, заключается в том, что вы фактически привязаны к базе данных MySQL. В данной главе этот вопрос не рассматривается, но, тем не менее, Symfony2 изначально интегрирована с ORM Doctrine, библиотекой, отвечающей за абстракцию от баз данных и соответствие данных между СУБД и вашими сущностями (mapping).

Давайте же поработаем над разрешением поставленных выше проблем.

Изоляция представления

При разделении "логики" приложения от кода, который подготавливает HTML "представление" страницы - общая структура приложения сразу же выигрывает:

```
<?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';
```

HTML код теперь расположен в отдельном файле (`templates/list.php`), который главным образом представляет собой HTML-файл, который использует PHP-синтаксис "для шаблонов":

```
<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>
```

По договорённости, файл, который содержит всю логику приложения - `index.php` - называется "контроллер". Термин *controller* - это слово, которое вы будете частенько слышать вне зависимости от языка программирования или же фреймворка, который

используете. В действительности же, речь идёт о части *вашего* кода, который обрабатывает пользовательский ввод и готовит ответ.

В нашем случае, контроллер получает данные из базы и подключает шаблон, для того чтобы отобразить их. С изоляцией контроллера, вы получили возможность поменять *лишь* шаблон, если вам вдруг понадобится отобразить записи блога в другом формате (например `list.json.php` для использования JSON-формата).

Изоляция логики Приложения (Домена)

Пока наше приложение содержало всего одну страницу. Но что же делать, если нужно добавить вторую страницу, которая использует то же подключение к базе данных или даже тот же массив постов из блога? Давайте преобразуем код, изолировав базовую логику от функций доступа к БД - поместим их в новый файл под названием `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;
}
```

Совет: Имя файла `model.php` использовано не случайно - логика и доступ к данным приложения традиционно известен как уровень "модели". В правильно организованном приложении бОльшая часть кода представляющая собой "бизнес-логику" должна быть расположена в модели (в противовес расположению её в контроллере). И, в отличие от нашего примера, лишь часть модели отвечает за доступ к БД (а бывает и вообще не отвечает).

Контроллер (`index.php`) теперь выглядит очень просто:

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

$posts = get_all_posts();

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

Теперь, в обязанности контроллера вменяется получение данных из модели приложения и вызов шаблона для отображения данных. Это очень простой пример паттерна model-view-controller.

Изоляция разметки (Layout)

На текущий момент, приложение разделено на три различных части, предлагающих различные преимущества и возможности по повторному использованию почти любого кода для других страниц.

Пока что мы *не можем* повторно использовать - это разметка страницы (layout). Исправим это упущение, создав файл `layout.php`:

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

Шаблон (`templates/list.php`) может быть упрощён, так как будет "расширять" базовую разметку:

```
<?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' ?>
```

Теперь вы знаете методологию, которая позволяет повторно использовать разметку-layout. К сожалению, для того чтобы достичь этого, вы вынуждены использовать несколько страшненьких PHP-функций (`ob_start()`, `ob_get_clean()`) в шаблоне. Symfony2 использует компонент `Templating`, который позволяет достичь этого просто и прозрачно. Скоро вы увидите - как именно.

2.2.2 Добавляем страницу блога “show”

Страница блога “list” была оптимизирована таким образом, чтобы код был лучше организован и позволял повторное использование. Для того чтобы доказать, что все оптимизации были не зря, добавим страницу “show”, которая отображает один пост идентифицируемый по параметру запроса - `id`.

Для начала, создадим новую функцию в файле `model.php`, которая получает одиночную запись по её id:

Далее, создадим новый файл, который назовем `show.php` - контроллер для нашей новой страницы:

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

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

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

И, наконец, создадим новый шаблон - `templates/show.php` - для отображения одного поста из блога:

```
<?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' ?>
```

Создание второй страницы выполнено легко и непринужденно, и мы избежали дублирования кода. Тем не менее, эта страница добавляет даже больше проблем, которые фреймворк может решить для вас. Например, отсутствующий или неверный параметр `id` вызовет фатальную ошибку приложения. Было бы лучше, если бы в этом случае отображалась страница 404, но сейчас мы не можем легко достичь такого эффекта. И ещё ложка дёгтя - ведь вы забыли "очистить" параметр `id` при помощи функции `mysql_real_escape_string()` - так что вся ваша база данных подвергается риску SQL-инъекции.

Другая серьёзная проблема заключается в том, что каждый файл-контроллер должен подключать файл `model.php`. А что если к каждому контроллеру неожиданно придется подключить дополнительный файл или же выполнить другую глобальную операцию (например, связанную с безопасностью)? При нынешней организации, этот код необходимо добавить в каждый контроллер. Если вы забудете включить что-нибудь в один из файлов, остаётся лишь надеяться, что это не скажется на безопасности приложения...

2.2.3 "Front Controller" вам в помощь

Решение указанных выше проблем является использование *front controller*: единственного PHP-файла, который будет обрабатывать *любой* запрос. При использовании front controller (далее просто фронт-контроллер) URI для вашего приложения изменяются незначительно, но становятся более гибкими:

```
Без фронт-контроллера
/index.php          => Список постов (выполняется index.php)
/show.php           => Отдельный пост (выполняется show.php)

При использовании index.php в качестве фронт-контроллера
/index.php          => Список постов (выполняется index.php)
/index.php/show     => Отдельный пост (выполняется index.php)
```

Примечание: Часть URI, включающая `index.php`, может быть опущена, при использовании rewrite rules веб-сервера Apache (или их эквивалента для прочих веб-серверов). В этом случае результирующий URI для страницы с постом блога будет просто `/show`.

При использовании фронт-контроллера, один PHP файл (`index.php` в нашем случае) обрабатывает *любой* запрос. Для страницы с одним постом `/index.php/show` будет вы-

полнять файл `index.php`, который теперь несёт ответственность за маршрутизацию запроса, основываясь на полном URI. Как вы скоро увидите фронт-контроллер - это очень мощный инструмент.

Создание фронт-контроллера

Внимание! Прямо сейчас вы стоите на пороге **большого** шага для вашего приложения. Имея один файл, который принимает все запросы, вы можете централизованно обрабатывать вопросы, связанные, к примеру, с безопасностью, загрузкой конфигурации, маршрутизацией. В нашем приложении `index.php` теперь должен быть достаточно умён, чтобы отобразить страницу со списком постов *или* страницу отельного поста, основываясь на URI запроса:

```
<?php
// index.php

// Загружаем и инициализируем глобальные библиотеки
require_once 'model.php';
require_once 'controllers.php';

// Внутренняя маршрутизация
$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>';
}
```

Для улучшения структуры приложения, оба контроллера (ранее `index.php` и `show.php`) превратились в функции, и каждая из них была помещена в файл `controllers.php`:

```
<?php
// controllers.php

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

function show_action($id)
{
    $post = get_post_by_id($id);
```

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

Став фронт-контроллером `index.php` получил совершенно новую роль, включая загрузку библиотек ядра и маршрутизацию, которая сейчас заключается в вызове одного из двух контроллеров (функции `list_action()` и `show_action()`). На самом деле, этот фронт-контроллер уже, в плане обработки запросов и маршрутизации, начинает себя вести сходным образом, как и контроллер Symfony2.

Примечание: Другое достоинство фронт-контроллера - это гибкие URL. Обратите внимание, что URL для страницы, отображающей отдельный пост блога, в любой момент может быть изменён с `/show` на `/read`, изменив код всего лишь в одном месте. Ранее же нам бы потребовалось переименовать файл целиком. В Symfony2 URLы ещё более гибки.

К этому времени, приложение разрослось с одного PHP-файла до целой структуры, которая хорошо организована и позволяет повторное использование кода. Вы должны быть счастливы, но до полного удовлетворения ещё далеко. К примеру, система "маршрутизации" ненадёжна и не может определить, что страница list (`/index.php`) должна быть доступна через `/` (если используются Apache rewrite rules). Также, вместо того чтобы разрабатывать блог, куча времени была потрачена на "архитектуру" кода (например, маршрутизация, вызовы контроллеров, шаблоны и т.п.). Еще больше времени нужно, чтобы обрабатывать отправку форм, валидацию введённых данных, логгирование и безопасность. Почему мы должны заново изобретать решения для этих рутинных проблем?

Прикосновение к Symfony2

Symfony2 идёт на помощь. Перед тем, как начать использовать Symfony2, вам нужно указать PHP как и где найти классы Symfony2. Это достигается путём использования автозагрузчика, который предоставляет Symfony. Автозагрузчик - это инструмент, который позволяет использовать PHP-классы, не подключая файлы их содержащие явно.

Во-первых, скачать symfony и поместите файлы в директорию `vendor/symfony/`. Затем, создайте файл `app/bootstrap.php`. Используйте его для подключения (`require`) двух файлов приложения и конфигурирования автозагрузчика:

```
<?php
// bootstrap.php
require_once 'model.php';
require_once 'controllers.php';
require_once 'vendor/symfony/src/Symfony/Component/ClassLoader/UniversalClassLoader.php';

$loader = new Symfony\Component\ClassLoader\UniversalClassLoader();
```

```
$loader->registerNamespaces(array(
    'Symfony' => __DIR__.'/vendor/symfony/src',
));

$loader->register();
```

Это покажет автозагрузчику, где живут классы `Symfony`. Теперь вы можете начать пользоваться классами Symfony, не используя оператор `require` для файлов, содержащих требуемые классы.

Ядром философии Symfony является идея, что основная задача приложения - это интерпретировать каждый запрос и возвратить ответ. Для этого Symfony2 предоставляет два класса: `Symfony\Component\HttpFoundation\Request` и `Symfony\Component\HttpFoundation\Response`. Эти классы являются объектно-ориентированным представлением необработанного HTTP-запроса, который подлежит обработке и соответствующего ему HTTP-ответа, который будет возвращен клиенту. Используйте их для улучшения блога:

```
<?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);
}

// Вывод заголовков и отправка ответа
$response->send();
```

Контроллеры теперь отвечают за возврат объекта `Response`. Для того чтобы упростить процесс создания ответа, вы можете добавить новую функцию `render_template()`, которая, между прочим, действует практически как движок шаблонов Symfony2:

```
<?php
// 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);
}

// Функция-помощник для отображения шаблонов
function render_template($path, array $args)
{
    extract($args);
    ob_start();
    require $path;
    $html = ob_get_clean();

    return $html;
}
```

Получив в помощь небольшую часть Symfony2, приложение стало более гибким и надёжным. `Request` предоставляет надёжный способ получить информацию о запросе. К примеру, метод `getPathInfo()` возвращает "очищенный" URI (всегда возвращает `/show` и никогда `/index.php/show`). Таким образом, даже если пользователь откроет в браузере `/index.php/show`, приложение выполнит `show_action()`.

Объект `Response` предоставляет гибкость в построении HTTP-ответа, позволяя добавлять HTTP заголовки и контент страницы посредством объектно-ориентированного интерфейса. И, хотя в этом приложении пока что ответы весьма просты, эта гибкость выплатит вам дивиденды по мере роста приложения.

Простое приложение на Symfony2

Блог начал свой *длинный* путь, но он всё ещё содержит слишком много кода для такого небольшого приложения. Следуя по пути, мы изобрели простую систему маршрутизации и метод, использующий `ob_start()` и `ob_get_clean()` для отображения шаблонов. Если, по каким-либо соображениям, вы хотите продолжить создание этого "фреймвор-

ка" с нуля, вы можете по крайней мере использовать самостоятельные компоненты Symfony - Routing и Templating, которые решают эти проблемы.

Вместо того чтобы заново решать типовые проблемы, вы можете предоставить Symfony2 заботу о них. Вот пример простого приложения, построенного с использованием 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:Post: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:Post:show.html.php', array('post' => $post));
    }
}
```

Эти два контроллера всё ещё легковесны. Каждый из них использует библиотеку Doctrine ORM для получения объектов из базы данных и компонент `Templating` для отображения шаблона и возврата объекта `Response`. Шаблон list теперь стал ещё немного проще:

```
<!-- 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>
```

Layout практически не изменился:

```
<!-- 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>
```

Примечание: Мы оставляем шаблон show вам в качестве самостоятельного упражнения, так как он будет не сложнее шаблона list.

Когда движок Symfony2 (который называется `Kernel` - ядро) загружается, он нуждается в "карте", по которой он будет узнавать - какой контроллер требуется выполнить, основываясь на информации из запроса. Конфигурация маршрутизатора предоставляет ему эту информацию в следующем формате:

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

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

Теперь, когда Symfony2 берёт на себя повседневные задачи, фронт-контроллер стал предельно простым. Поскольку он теперь делает так мало, вам никогда не придется трогать его после создания (а если вы используете дистрибутив Symfony2, то вам даже не придётся создавать его!):

```
<?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();
```

Единственная забота фронт-контроллера - инициализация движка Symfony2 (`Kernel`) и передача ему объекта `Request` для последующей обработки. Ядро Symfony2 использует карту маршрутизации для определения - какой контроллер необходимо выполнить. Как и раньше, метод контроллера отвечает за возврат конечного объекта `Response`.

Для визуального представления процесса обработки запроса в Symfony2 - посмотрите диаграмму *процесс обработки запроса*.

В чём польза Symfony2

В последующих главах вы узнаете больше обо всех аспектах работы с Symfony и рекомендуемой структуре проекта. Сейчас же давайте посмотрим - как миграция блога с обычного PHP на Symfony2 улучшает жизнь:

- Теперь ваше приложение имеет **простой, понятный и единообразно организованный код** (хотя Symfony не требует этого от вас). Это поощряет **повторное использование** и позволяет новым разработчикам становиться продуктивными быстрее.
- 100% кода, который вы написали - для *вашего* приложения. Вам **не нужно разрабатывать или поддерживать низкоуровневые инструменты**, такие как *автозагрузка*, *маршрутизация*, или рендеринг *контроллеров*.
- Symfony2 предоставляет вам **доступ к инструментам с открытым кодом**, таким как Doctrine, и компонентам Templating, Security, Form, Validation and Translation.
- Приложение теперь использует **гибчайшие URLы** благодаря компоненту `Routing`.
- Архитектура Symfony2, центрированная на HTTP, дает вам доступ к мощным инструментам, таким как **HTTP кеширование**, базирующееся на **внутреннем HTTP-кэше Symfony2** или более ещё более мощным инструментам, таким как Varnish. Об этом будет рассказано в главе о *кэшировании*.

И, возможно самое лучшее, используя Symfony2 вы получаете доступ к целому набору **качественных инструментов с открытым исходным кодом, разработанных**

участниками коммьюнити! Дополнительную информацию вы можете получить на сайте Symfony2Bundles.org

2.2.4 Лучшие шаблоны

Если вы выбрали Symfony2, то приготовьтесь встретиться с шаблонизатором Twig, который делает шаблоны быстрыми в разработке и лёгкие в понимании. Это означает, что приложение будет содержать ещё меньше кода! Давайте, к примеру, взглянем на шаблон списка, написанный на 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 %}
```

Соответствующий шаблон `layout.html.twig` ещё проще:

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

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

Twig отлично интегрирован с Symfony2. В то время, как PHP шаблоны будут всегда поддерживаться в Symfony2, мы также будем продолжать обсуждения преимуществ Twig. Больше информации о Twig вы найдете в *главе о шаблонах*.

2.2.5 Дополнительная информация в Cookbook

- *Как использовать PHP шаблоны вмесро Twig*
- *Как определять Контроллеры в качестве сервисов*

2.3 Установка и настройка Symfony2

Цель этой главы помочь вам настроить и запустить рабочее приложение, созданное при помощи Symfony. К счастью, Symfony предлагает "дистрибутивы", которые представляют собой базовые проекты, которые вы можете загрузить и незамедлительно начать разработку.

Совет: Если вы ищите руководство по созданию нового проекта и размещению его в системе контроля версий, перейдите к секции Использование системы контроля версий.

2.3.1 Загрузка дистрибутива Symfony2

Совет: Прежде всего, удостоверьтесь, что у вас установлен и настроен Web-сервер (например, Apache) и интерпретатор PHP 5.3.2 или более новый. Более подробную информацию о системных требованиях Symfony2 в можете найти в разделе `Системные требования`.

Дистрибутивы Symfony2 представляют собой полнофункциональные приложения, включающие ядро Symfony2, набор полезных пакетов (Bundles), разумную структуру директорий и конфигурацию по умолчанию. Когда вы загружаете дистрибутив Symfony2, вы фактически загружаете скелетон функционирующего приложения, который тут же можно начать использовать как базу для вашего собственного приложения.

Начнём со страницы загрузки Symfony2 http://symfony.com/download. На этой странице вы можете видеть дистрибутив *Symfony Standard Edition*, который является основным дистрибутивом. Теперь вам нужно принять 2 решения:

- Загрузить либо `.tgz` либо `.zip` архив - они идентичны, просто вопрос предпочтений.
- Загрузить дистрибутив, включающий сторонние библиотеки или же не включающий (with/without vendors). Если у вас установлен Git, вы можете загрузить Symfony2 "without vendors", так так это даст вам немного больше возможностей по включению сторонних библиотек/вендоров.

Загрузите один из архивов в root-директорию вашего локального web-сервера и распакуйте его. В командной строке UNIX это можно выполнить при помощи одной из этих команд (заменяя ### актуальным именем файла):

```
# for .tgz file
tar zxvf Symfony_Standard_Vendors_2.0.###.tgz

# for a .zip file
unzip Symfony_Standard_Vendors_2.0.###.zip
```

Когда вы выполните эту операцию, у вас будет директория `Symfony/`, которая будет выглядеть примерно так:

```
www/ <- root директория вашего веб-сервера
    Symfony/ <- распакованный архив
        app/
            cache/
            config/
            logs/
        src/
            ...
        vendor/
            ...
        web/
            app.php
            ...
```

Обновление Вендоров

Далее, если вы загрузили архив "без вендоров" (without vendors), необходимо их установить, выполнив следующую команду:

```
php bin/vendors install
```

Эта команда загрузит все необходимые библиотеки, включая собственно Symfony, в директорию `vendor/`. Более подробную информацию о том, как управлять сторонними библиотеками в Symfony2 вы можете получить в разделе "*Управление внешними библиотеками с помощью bin/vendors и deps*".

Конфигурация и настройка

На текущий момент все необходимые сторонние библиотеки теперь располагаются в директории `vendor/`. Также в директории `app/` расположены настройки по-умолчанию, а в директории `src/` пример кода.

Symfony2 поставляется с визуальным тестером конфигурации веб-сервера, для того чтобы помочь вам определить, подходит ли конфигурация вашего сервера и PHP для Symfony. Используйте следующий URL для проверки конфигурации:

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

Если проверка показывает какие-либо несоответствия - исправьте их, прежде чем двигаться далее.

Настройка прав доступа

Одно из типовых замечаний заключается в том, что директории `app/cache` и `app/logs` должны иметь права на запись как для веб-сервера, так и для пользователя, от имени которого выполняются команды из командной строки. В UNIX-системах, если пользователь, из-под которого запускается веб-сервер отличается от пользователя командной строки, вы можете выполнить следующие команды, для того чтобы быть уверенными, что права доступа настроены верно. Заменяйте `www-data` на пользователя веб-сервера и `yourname` на вашего пользователя командной строки:

1. Использование ACL в системах, которые поддерживают chmod +a

Многие системы позволяют использовать команду `chmod +a`. Попробуйте выполнить её, и если вы получите сообщение об ошибке - пробуйте следующий метод:

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

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

2. Использование Acl на системах, которые не поддерживают chmod +a

Некоторые системы не поддерживают `chmod +a`, но поддерживают другую утилиту, `setfacl`. Возможно, вам потребуется включить поддержку ACL на вашем разделе и установить `setfacl` перед тем как использовать (это может потребоваться, например, если вы используете Ubuntu):

```
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. Без использования ACL

Если у вас нет прав на изменение ACL для директорий, вам потребуется изменить umask таким образом, чтобы директории cache и log были доступны на запись группе или же всем (world-writable) в зависимости от того находятся ли пользователи веб-сервера и командной строки в одной группе или нет. Для этого нужно вставить следующую строчку в начало файлов `app/console`, `web/app.php` и `web/app_dev.php`:

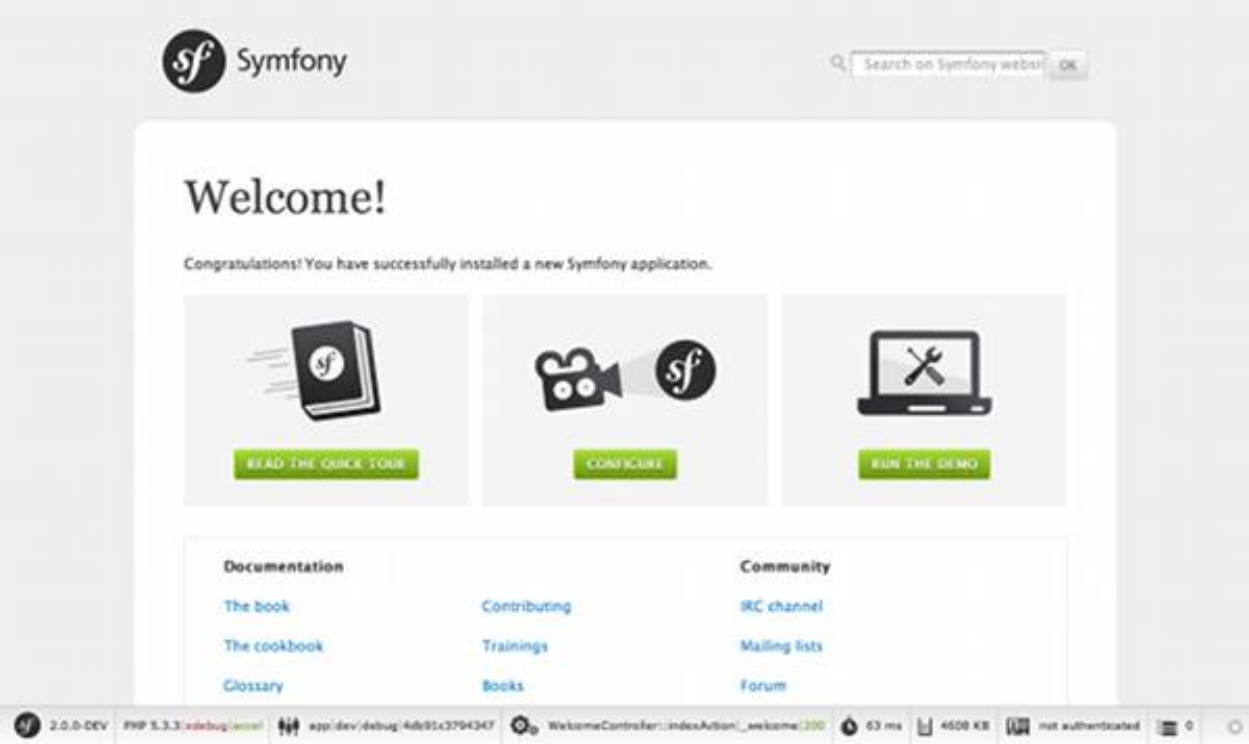
```
umask(0002); // Разрешает использовать права 0775

// или

umask(0000); // Разрешает использовать права 0777
```

Имейте в виду, что использование ACL предпочтительнее, когда вы имеете доступ к ним на сервере, потому что смена umask не является thread-safe.

Когда все необходимые приготовления выполнены, кликните на ссылку “Go to the Welcome page” и перейдите на вашу первую “настоящую” страницу Symfony2:

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

Symfony2 поздоровкается и поздравит вас с проделанной тяжелой работой!!



2.3.2 Начало разработки

Теперь, когда мы имеем настроенное Symfony2 приложение, вы можете начать разработку. Ваш дистрибутив может содержать примеры кода - прочтите файл `README.rst` из дистрибутива (это обычный текстовый файл) для того чтобы ознакомиться с тем, какие примеры включены в данный дистрибутив и как их можно будет удалить позднее.

Если вы новичок в Symfony, ознакомьтесь с руководством "*Создание страниц в Symfony2*", где вы узнаете, как создавать страницы, изменять настройки и вообще делать всё необходимое для создания нового приложения.

2.3.3 Использование системы контроля версий

Если вы используете систему контроля версий типа `Git` или `Subversion`, вы можете настроить вашу систему и начать коммитить ваш проект как вы это делаете обычно. Symfony Standard - это точка отсчёта для вашего нового проекта.

Более подробные инструкции о том, как лучше всего настроить проект для хранения в git, загляните сюда: *Как создать и разместить Проект на Symfony2 в git-репозитории*.

Игнорируем директорию `vendor/`

Если вы загрузили архив *без вендоров* вы можете спокойно игнорить директорию `vendor/` целиком и не коммитить её содержимое в систему контроля версий. В `Git` этого можно добиться, создав файл `.gitignore` и добавив в него следующую строку:

```
vendor/
```

После этого директория vendor не будет участвовать в коммитах. Это здорово (правда-правда!), потому что когда кто-то еще клонирует или выгрузит ваш проект он сможет запросто выполнить скрипт `php bin/vendors install` и загрузить все необходимые библиотеки.

2.4 Создание страниц в Symfony2

Создание новой страницы в Symfony2 это простой процесс, состоящий из 2 шагов:

- *Создание маршрута*: Маршрут определяет URL (например `/about`) для вашей страницы, а также контроллер (PHP функция), который Symfony2 должен выполнить, когда URL входящего запроса совпадет шаблоном маршрута;
- *Создание контроллера*: Контроллер – это PHP функция, которая принимает входящий запрос и преобразует его в объект `Response`, который будет возвращен пользователю.

Нам нравится такой подход, потому что он соответствует тому, как работает Web. Каждое взаимодействие в Web инициализируется HTTP запросом. Забота вашего приложения – интерпретировать запрос и вернуть соответствующий ответ.

Symfony2 следует этой философии и предоставляет вам инструменты и соглашения, для того чтобы ваше приложение оставалось хорошо структурированным при росте его посещаемости и сложности.

Звучит просто? Давайте копнём по глубже!

2.4.1 Страница “Hello Symfony!”

Давайте начнем с классического приложения “Hello World!”. Когда вы закончите работу над ним, пользователь приложения будет иметь возможность получить персональное приветствие, (например “Hello Symfony”), перейдя по следующему URL:

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

Вы также сможете заменить `Symfony` на другое имя и получить новое приветствие. Для создания этой страницы мы пройдем простой путь из двух шагов.

Примечание: Данное руководство подразумевает, что вы уже скачали Symfony2 и настроили ваш веб-сервер. URL, указанный выше, подразумевает, что localhost указывает на web-директорию вашего нового Symfony2 проекта. Если же вы ещё не выполнили этих шагов, рекомендуется их выполнить, прежде чем вы продолжите чтение. Дополнительную информацию вы можете найти в главе *Установка и настройка Symfony2*.

Прежде чем начать: создание Пакета (bundle)

Прежде чем начать, вам необходимо создать пакет (*bundle*). В Symfony2 пакет напоминает plugin, за исключением того, что весь код вашего приложения будет расположен внутри такого пакета.

Вообще говоря, пакет – это не более чем директория, которая содержит все что относится к какой-то специфической функции, включая PHP-классы, настройки и даже стили и файлы Javascript (см. *Система пакетов*).

Для создания пакета с именем `AcmeHelloBundle` (демо-пакет, который вы создадите в ходе прочтения данной статьи), необходимо выполнить следующую команду и следовать инструкциям, которые появятся на экране (установите все опции по-умолчанию):

```
php app/console generate:bundle --namespace=Acme/HelloBundle --format=yml
```

За кулисами же произойдёт вот что: будет создана директория для пакета `src/Acme/HelloBundle`. Также в файл `app/AppKernel.php` автоматически будет добавлена строка, которая зарегистрирует вновь созданный пакет:

```
<?php

// app/AppKernel.php
public function registerBundles()
{
    $bundles = array(
        // ...
        new Acme\HelloBundle\AcmeHelloBundle(),
    );
    // ...

    return $bundles;
}
```

Теперь, когда вы создали и инициализировали пакет, вы можете приступить к созданию вашего приложения.

Шаг 1: Создание маршрута

По умолчанию, конфигурационный файл маршрутизатора в приложении Symfony2, располагается в `app/config/routing.yml`. Для конфигурирования маршрутизатора, а также любых прочих конфигураций Symfony2, вы можете также использовать XML или PHP формат.

Если вы посмотрите в основной конфигурационный файл, вы увидите, что Symfony уже добавил запись для сгенерированного `AcmeHelloBundle`:

- *YAML*

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

- *XML*

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

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

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

- *PHP*

```php
<?php

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

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

return $collection;
```

Эта запись очень проста: она сообщает Symfony, что необходимо загрузить конфигурацию маршрутизатора из файла `Resources/config/routing.yml`, который расположен в пакете `AcmeHelloBundle`. Это означает, что вы можете размешать конфигурацию

маршрутизатора непосредственно в `app/config/routing.yml` или же хранить маршруты внутри пакета и импортировать их оттуда.

Теперь, когда файл `routing.yml` импортирован из пакета, добавьте новый маршрут, который определит URL страницы, которую вы собираетесь создать:

- *YAML*

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

- *XML*

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

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

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

- *PHP*

```php
<?php

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

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

return $collection;
```

Маршрут состоит из двух основных частей: шаблона (`pattern`), с которым сравнивается URL, а также массива параметров по умолчанию (`defaults`), в котором указывается контроллер, который необходимо выполнить. Заполнитель `{name}` в шаблоне – это метасимвол (wildcard). Он означает, что URL `/hello/Ryan`, `/hello/Fabien`, а также прочие, похожие на них, будут соответствовать этому же маршруту. Параметр, определённый заполнителем `{name}`, также будет передан в контроллер, так что вы сможете использовать его, чтобы поприветствовать пользователя.

Примечание: Система маршрутизации имеет еще множество замечательных функций для создания гибких и функциональных структур URL в приложении. За дополнительной информацией вы можете обратиться к главе *Маршрутизация*.

Шаг 2: Создание Контроллера

Когда URI вида `/hello/Ryan` обнаруживается приложением в запросе, маршрут `hello` совпадёт с ним и будет вызван контроллер `AcmeHelloBundle:Hello:index`. Следующим вашим шагом будет создание этого контроллера.

Контроллер `AcmeHelloBundle:Hello:index` - это *логическое* имя контроллера и оно соответствует методу `indexAction` PHP-класса, именуемого `Acme\HelloBundle\Controller\Hello`. Приступим к созданию этого файла внутри `AcmeHelloBundle`:

```
<?php

// src/Acme/HelloBundle/Controller/HelloController.php
namespace Acme\HelloBundle\Controller;

use Symfony\Component\HttpFoundation\Response;

class HelloController
{
}
```

В действительности, контроллер – это не что иное, как метод PHP класса, который вы создаёте, а Symfony выполняет. Это то место, где ваш код, используя информацию из запроса, создает запрошенный ресурс. За исключением некоторых особых случаев, результатом работы контроллера всегда является объект Symfony2 `Response`.

Создайте метод `indexAction`, который Symfony выполнит, когда сработает маршрут `hello`:

```
<?php

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

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

```
    }
}
```

Этот контроллер предельно прост: он создает новый объект `Response`, чьим первым аргументом является контент, который будет использован для создания ответа (в нашем случае это маленькая HTML-страница, код которой мы указали прямо в контроллере).

Примите наши поздравления! После создания всего лишь маршрута и контроллера, вы уже имеете полноценную страницу! Если вы все настроили корректно, ваше приложение должно поприветствовать вас:

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

Совет: Вы также можете отобразить ваше приложение в *"продуктовом (prod)" окружении*, посетив следующий URL:

```
http://localhost/app.php/hello/Ryan
```

Если вы увидите ошибку, то скорее всего вам всего лишь необходимо очистить кэш, выполнив команду:

```
php app/console cache:clear --env=prod --no-debug
```

Не обязательным (но, как правило, востребованным) третьим шагом является создание шаблона.

Примечание: Контроллер – это главная точка входа для вашего кода и ключевой ингредиент при создании страниц. Больше информации о контроллерах вы можете найти в главе *Контроллер*.

Необязательный шаг 3: Создание шаблона

Шаблоны позволяют нам вынести разметку страниц (HTML код, как правило) в отдельный файл и повторно использовать различные части шаблона страницы. Вместо того чтобы писать код внутри контроллера, воспользуемся шаблоном:

```
1 <?php
2
3 // src/Acme/HelloBundle/Controller/HelloController.php
4 namespace Acme\HelloBundle\Controller;
5
6 use Symfony\Bundle\FrameworkBundle\Controller\Controller;
7
8 class HelloController extends Controller
```

```
9  {
10     public function indexAction($name)
11     {
12         return $this->render('AcmeHelloBundle:Hello:index.html.twig', array('name' => $name));
13
14         // render a PHP template instead
15         // return $this->render('AcmeHelloBundle:Hello:index.html.php', array('name' => $name));
16     }
17 }
```

Примечание: Для того, чтобы использовать метод `render()`, необходимо отнаследоваться от класса `Symfony\Bundle\FrameworkBundle\Controller\Controller` (API docs: `Symfony\Bundle\FrameworkBundle\Controller\Controller`), который добавляет несколько методов для быстрого вызова часто употребляемых функций контроллера. В предыдущем примере это достигается путём добавления выражения `use` в строке 6 и, затем, наследованием от класса `Controller` в строке 8.

Метод `render()` создает объект `Response`, заполненный результатом обработки (рендеринга) шаблона. Как и в любом другом контроллере, вы, в конце концов, вернете объект `Response`.

Обратите внимание, что есть две различные возможности рендеринга шаблонов. Symfony2 по умолчанию, поддерживает 2 языка шаблонов: классические PHP-шаблоны и простой, но мощный язык шаблонов Twig. Но не пугайтесь, вы свободны в выборе того или иного из них, кроме того вы можете использовать оба в рамках одного проекта.

Контроллер отображает шаблон `AcmeHelloBundle:Hello:index.html.twig`, который назван с использованием следующих соглашений:

BundleName:**ControllerName**:**TemplateName**

Это, так называемое, логическое имя шаблона, которое соответствует физическому файлу на основании следующих соглашений:

/**путь**/**к**/**BundleName**/Resources/views/**ControllerName**/**TemplateName**

В нашем случае `AcmeHelloBundle` - это наименование пакета, `Hello` - это контроллер и `index.html.twig` - это шаблон:

- *Twig*

```
1  {# src/Acme/HelloBundle/Resources/views/Hello/index.html.twig #}
2  {% extends '::base.html.twig' %}
3
4  {% block body %}
5      Hello {{ name }}!
6  {% endblock %}
```

- *PHP*

```
<!-- src/Acme/HelloBundle/Resources/views/Hello/index.html.php -->
<?php $view->extend('::base.html.php') ?>

Hello <?php echo $view->escape($name) ?>!
```

Давайте рассмотрим подробнее шаблон Twig:

- *строка 2*: Токен `extends` определяет родительский шаблон. Таким образом, сам шаблон однозначным образом определяет родителя (layout) внутрь которого он будет помещен.
- *строка 4*: Токен `block` означает, что всё внутри него будет помещено в блок с именем `body`. Как вы увидите ниже, это уже обязанность родительского шаблона (`base.html.twig`) – полностью отобразить блок `body`.

Родительский шаблон, `::base.html.twig`, не включает в себя ни **имени пакета**, ни **имени контроллера** (отсюда и двойное двоеточие в начале имени (::)). Это означает, что шаблон располагается вне пакета в директории `app`:

- *Twig*

```
{# app/Resources/views/base.html.twig #}
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
        <title>{% block title %}Welcome!{% endblock %}</title>
        {% block stylesheets %}{% endblock %}
        <link rel="shortcut icon" href="{{ asset('favicon.ico') }}" />
    </head>
    <body>
        {% block body %}{% endblock %}
        {% block javascripts %}{% endblock %}
    </body>
</html>
```

- *PHP*

```
<!-- app/Resources/views/base.html.php -->
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
        <title><?php $view['slots']->output('title', 'Welcome!') ?></title>
        <?php $view['slots']->output('stylesheets') ?>
        <link rel="shortcut icon" href="<?php echo $view['assets']->getUrl('favicon.ico')
    </head>
    <body>
```

```
            <?php $view['slots']->output('_content') ?>
            <?php $view['slots']->output('stylesheets') ?>
        </body>
    </html>
```

Базовый шаблон определяет HTML-разметку и отображает блок `body`, который вы определили в шаблоне `index.html.twig`. Так как вы не определили блок `title` в дочернем шаблоне, он останется со значением по умолчанию - "Welcome!".

Шаблоны являются мощным инструментом по организации и отображению контента ваших страниц. Шаблон может отобразить всё что угодно от HTML разметки, до CSS-кода или что контроллеру будет угодно.

В жизненном цикле обработки запроса, шаблонизатор - это всего лишь опциональный инструмент. Не забывайте, что цель каждого контроллера - вернуть объект `Response`. Шаблоны являются мощным инструментом, но они опциональны, всего лишь инструмент для создания контента для объекта `Response`.

2.4.2 Структура директорий

Всего лишь после прочтения нескольких коротких секций вы уже уяснили философию создания и отображения страниц в Symfony2. Поэтому без лишних слов мы приступим к изучению того, как организованы и структурированы проекты Symfony2. К концу этой секции вы будете знать, где найти и куда поместить различные типы файлов. И более того, будет понимать – почему!

По умолчанию каждое Symfony приложение (*application*), изначально созданное быть очень гибким, имеет одну и ту же базовую (и рекомендуемую) структуру директорий:

- `app/`: Эта директория содержит настройки приложения;
- `src/`: Весь PHP код проекта находится в этой директории;
- `vendor/`: Здесь размещаются сторонние библиотеки;
- `web/`: Это корневая директория, видимая web-серверу и содержащая доступные пользователям файлы;

Директория Web

Web-директория – это дом для всех публично-доступных статических файлов, таких как изображения, таблицы стилей и JavaScript файлы. Тут также располагаются все фронт-контроллеры (*front controller*):

```
<?php

// web/app.php
```

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

use Symfony\Component\HttpFoundation\Request;

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

Файл фронт-контроллера (в примере выше – `app.php`)- это PHP файл, который выполняется, когда используется Symfony2 приложение и в его обязанности входит использование Kernel-класса(`AppKernel`), для запуска приложения.

Совет: Наличие фронт-контроллера означает возможность использования более гибких URL, отличных от тех, что используются в типичном "плоском" PHP-приложении. Когда используется фронт-контроллер, URL формируется следующим образом:

```
http://localhost/app.php/hello/Ryan
```

Фронт-контроллер `app.php` выполняется и "внутренний:" URL `/hello/Ryan` направляется внутри приложения с использованием конфигурации маршрутизатора. С использованием правил mod_rewrite для Apache вы можете перенаправлять все запросы (на физически не существующие URL) на app.php, чтобы явно не указывать его в URL:

```
http://localhost/hello/Ryan
```

Хотя фронт-контроллеры имеют важное значение при обработке каждого запроса, вам нечасто придется модифицировать их или вообще вспоминать об их существовании. Мы еще вкратце упомянем о них в секции, где говорится об Окружениях (Окружения).

Директория приложения (app)

Как вы уже видели во фронт-контроллере, класс `AppKernel` – это точка входа приложения и он отвечает за его конфигурацию. Как таковой, этот класс расположен в директории `app/`.

Этот класс должен реализовывать два метода, которые определяют всё, что Symfony необходимо знать о вашем приложении. Вам даже не нужно беспокоиться о реализации этих методов, когда начинаете работу – они уже реализованы и содержат код по умолчанию.

- `registerBundles()`: Возвращает массив всех пакетов, необходимых для запуска приложения (см. *Система пакетов*);
- `registerContainerConfiguration()`: Загружает главный конфигурационный файл (см. секцию Конфигурация приложения).

Изо дня в день вы будете использовать директорию `app/` в основном для того, чтобы модифицировать конфигурацию и настройки маршрутизатора в директории `app/config/` (см. Конфигурация приложения). Также в `app/` содержится кэш (`app/cache`), директория для логов (`app/logs`) и директория для ресурсов уровня приложения (`app/Resources`). Об этих директориях подробнее будет рассказано в других главах.

Автозагрузка

При инициализации приложения подключается особый файл: `app/autoload.php`. Этот файл отвечает за автозагрузку всех файлов из директорий `src/` и `vendor/`. Благодаря автозагрузке, вам больше не придется беспокоиться об использовании выражений `include` или `require`. Вместо этого, Symfony2 использует пространства имен классов, чтобы определить их расположение и автоматически подключить файл класса, в случае если класс вам понадобится.
Автозагрузчик уже настроен для того, чтобы искать ваши классы в директории `src/`. Для того чтобы автозагрузка работала, имя класса и путь к его файлу должны следовать следующему шаблону:

```
Class Name:
    Acme\HelloBundle\Controller\HelloController
Path:
    src/Acme/HelloBundle/Controller/HelloController.php
```

Как правило, единственная ситуация, когда вам необходимо беспокоиться о файле `app/autoload.php` - когда вы добавляете новую стороннюю библиотеку в директорию `vendor/`. Если вы хотите узнать больше об автозагрузке, обратите внимание на статью *Как автоматически загружать классы*.

Директория исходных кодов проекта (`src`)

Если вкратце, то директория `src/` содержит весь код *вашего* приложения (PHP-код, шаблоны, конфигурационные файлы, стили и т.д.). Во время разработки, бОльшую часть работ вы будете выполнять внутри одного или нескольких пакетов, которые вы создадите именно в этой директории.

Но что же собственно из себя представляет сам пакет (*bundle*)?

2.4.3 Система пакетов

Пакет чем-то схож с плагином, но он ещё лучше. Ключевое отличие состоит в том, что *всё есть пакет* в Symfony2, включая функционал ядра и код вашего приложения. Пакеты – это граждане высшего сорта в Symfony2. Они дают вам возможность использовать уже готовые пакеты, которые вы можете найти на сайте symfony2bundles.org.

Вы также можете там выкладывать свои пакеты. Они также дают возможность легко и просто выбрать, какие именно функции подключить в вашем приложении.

Примечание: Здесь мы рассмотрим лишь основы, более детальную информацию по пакетам вы можете найти в статье `пакеты` в "книге рецептов".

Пакет это просто структурированный набор файлов и директорий, который реализует одну конкретную функцию. Вы можете создать `BlogBundle` или `ForumBundle` или же пакет для управления пользователями (такие пакеты уже есть и даже с открытым исходным кодом). Каждая директория содержит все необходимое для реализации этой конкретной функции, включая PHP файлы, шаблоны, стили, клиентские скрипты, тесты и все что ещё потребуется. Каждый аспект реализации функции находится в своём пакете и каждая функция располагается в своем собственном пакете.

Приложение состоит из пакетов, которые объявлены в методе `registerBundles()` класса `AppKernel`:

```
<?php

// app/AppKernel.php
public function registerBundles()
{
    $bundles = array(
        new Symfony\Bundle\FrameworkBundle\FrameworkBundle(),
        new Symfony\Bundle\SecurityBundle\SecurityBundle(),
        new Symfony\Bundle\TwigBundle\TwigBundle(),
        new Symfony\Bundle\MonologBundle\MonologBundle(),
        new Symfony\Bundle\SwiftmailerBundle\SwiftmailerBundle(),
        new Symfony\Bundle\DoctrineBundle\DoctrineBundle(),
        new Symfony\Bundle\AsseticBundle\AsseticBundle(),
        new Sensio\Bundle\FrameworkExtraBundle\SensioFrameworkExtraBundle(),
        new JMS\SecurityExtraBundle\JMSSecurityExtraBundle(),
    );

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

    return $bundles;
}
```

Используя метод `registerBundles()`, вы получаете полный контроль над теми пакетами, которые используются вашим приложением (включая пакеты, входящие в состав ядра Symfony).

Совет: Вообще говоря, пакет может располагаться где угодно, если автозагрузчик (`app/autoload.php`) можно настроить таким образом, чтобы этот пакет мог быть загружен.

Создание пакета

Symfony Standard Edition содержит удобную команду для создания пакета. Тем не менее, создать пакет вручную лишь немногим больше и ничуть не сложнее.

Чтобы показать вам, как проста система пакетов, давайте создадим новый пакет, назовём его `AcmeTestBundle` и активируем его.

Совет: `Acme` это всего-лишь формальное имя, которое должно быть заменено на наименование некоего вендора, которое будет представлять вашу организацию (например `ABCTestBundle` для компании `ABC`).

В первую очередь, создадим директорию `src/Acme/TestBundle/` и добавим в неё файл `AcmeTestBundle.php`:

```
<?php

// src/Acme/TestBundle/AcmeTestBundle.php
namespace Acme\TestBundle;

use Symfony\Component\HttpKernel\Bundle\Bundle;

class AcmeTestBundle extends Bundle
{
}
```

Совет: Наименование класса `AcmeTestBundle` следует стандарту *Именования Пакетов*. Вы также можете сократить наименование пакета до `TestBundle`, назвав класс `TestBundle` (и переименовав файл в `TestBundle.php`).

Этот пустой класс – единственное, что необходимо создать для минимальной комплектации пакета. Не смотря на то, что класс пуст, он обладает большим потенциалом и позволяет настраивать поведение пакета.

Теперь, когда мы создали пакет, его нужно активировать в классе `AppKernel`:

```
<?php

// app/AppKernel.php
```

```
public function registerBundles()
{
    $bundles = array(
        // ...

        // register your bundles
        new Acme\TestBundle\AcmeTestBundle(),
    );
    // ...

    return $bundles;
}
```

И, хотя наш новый пакет пока ничего не делает, `AcmeTestBundle` готов к использованию.

Symfony также предлагает интерфейс для командной строки для создания базового скелетона пакета:

```
php app/console generate:bundle --namespace=Acme/TestBundle
```

Каркас пакета создаёт базовый контроллер, шаблон и маршрут, которые можно настроить впоследствии. Мы еще вернёмся к инструментам командной строки позже.

Совет: Когда создаёте новый пакет, или используете сторонние пакеты, убедитесь, что пакет активирован в registerBundles(). При использовании же команды `generate:bundle` - это действие производится автоматически.

Структура директории пакета

Структура директории пакета проста и гибка. По умолчанию, система пакетов следует некоторым соглашениям, которые помогают поддерживать стилевое единообразие во всех пакетах Symfony2. Давайте взглянем на пакет `AcmeHelloBundle`, так как он содержит наиболее основные элементы пакета:

- `Controller/` содержит контроллеры пакета (например `HelloController.php`);
- `Resources/config/` дом для конфигурационных файлов, включая конфигурацию маршрутизатора (например `routing.yml`);
- `Resources/views/` шаблоны, сгруппированные по имени контроллера (например `Hello/index.html.twig`);
- `Resources/public/` публично доступные ресурсы (картинки, стили...), которые будут скопированы или связаны символической ссылкой с `web/` директорией при помощи консольной команды `assets:install`;
- `Tests/` содержит все тесты пакета.

Пакет может быть как маленьким, так и большим – в зависимости от задачи, которую он реализует. Он содержит лишь те файлы, которые нужны – и ничего более.

В других главах книги вы также узнаете, как работать с базой данных, как создавать и валидировать формы, создавать файлы переводов, писать тесты и много чего ещё. Все эти объекты в пакете имеют определенную роль и место.

2.4.4 Конфигурация приложения

Приложение состоит из набора пакетов, реализующих все необходимые функции вашего приложения. Каждый пакет может быть настроен при помощи конфигурационных файлов, написанных на YAML, XML или PHP. По умолчанию, основной конфигурационный файл расположен в директории `app/config/` и называется `config.yml`, `config.xml` или `config.php`, в зависимости от предпочитаемого вами формата:

- *YAML*

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

framework:
    secret:          %secret%
    charset:         UTF-8
    router:          { resource: "%kernel.root_dir%/config/routing.yml" }
    form:            true
    csrf_protection: true
    validation:      { enable_annotations: true }
    templating:      { engines: ['twig'] } #assets_version: SomeVersionScheme
    session:
        default_locale: %locale%
        auto_start:     true

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

# ...
```

- *XML*

```
<!-- app/config/config.xml -->
<imports>
    <import resource="parameters.yml" />
    <import resource="security.yml" />
```

```
    </imports>

    <framework:config charset="UTF-8" secret="%secret%">
        <framework:router resource="%kernel.root_dir%/config/routing.xml" />
        <framework:form />
        <framework:csrf-protection />
        <framework:validation annotations="true" />
        <framework:templating assets-version="SomeVersionScheme">
            <framework:engine id="twig" />
        </framework:templating>
        <framework:session default-locale="%locale%" auto-start="true" />
    </framework:config>

    <!-- Twig Configuration -->
    <twig:config debug="%kernel.debug%" strict-variables="%kernel.debug%" />

    <!-- ... -->
```

- *PHP*

```
    <?php

    $this->import('parameters.yml');
    $this->import('security.yml');

    $container->loadFromExtension('framework', array(
        'secret'          => '%secret%',
        'charset'         => 'UTF-8',
        'router'          => array('resource' => '%kernel.root_dir%/config/routing.php'),
        'form'            => array(),
        'csrf-protection' => array(),
        'validation'      => array('annotations' => true),
        'templating'      => array(
            'engines' => array('twig'),
            #'assets_version' => "SomeVersionScheme",
        ),
        'session' => array(
            'default_locale' => "%locale%",
            'auto_start'     => true,
        ),
    ));

    // Twig Configuration
    $container->loadFromExtension('twig', array(
        'debug'            => '%kernel.debug%',
        'strict_variables' => '%kernel.debug%',
    ));
```

```
// ...
```

Примечание: Подробнее о том, как загружать каждый файл/формат будет рассказано в следующей секции - Окружения.

Каждый параметр верхнего уровня, например `framework` или `twig`, определяет настройки конкретного пакета. Например, ключ `framework` определяет настройки ядра Symfony `FrameworkBundle` и включает настройки маршрутизации, шаблонизатора и прочих ключевых систем.

Пока же нам не стоит беспокоиться о конкретных настройках в каждой секции. Файл настроек по умолчанию содержит все необходимые параметры. По ходу чтения прочей документации вы ознакомитесь со всеми специфическими настройками.

Форматы конфигураций

Во всех главах книги все примеры конфигураций будут показаны во всех трех форматах (YAML, XML and PHP). Каждый из них имеет свои достоинства и недостатки. Выбор же формата целиком зависит о ваших предпочтений:

- *YAML*: Простой, понятный и читабельный;
- *XML*: В разы более мощный, нежели YAML, к тому же многие современные IDE поддерживают автозавершение в XML;
- *PHP*: Очень мощный, но менее читабельный, чем стандартные форматы конфигурационных файлов.

2.4.5 Окружения

Приложение можно запускать в различных окружениях. Различные окружения используют один и тот же PHP код (за исключением фронт-контроллера), но могут иметь совершенно различные настройки. Например, `dev` окружение ведет лог ошибок и замечаний, в то время как `prod` окружение логгирует только ошибки. В `dev` некоторые файлы пересоздаются при каждом запросе, но кэшируются в `prod` окружении. В то же время, все окружения одновременно доступны на одной и той же машине.

Проект Symfony2 по умолчанию имеет три окружения (`dev`, `test` и `prod`), хотя создать новое окружение не сложно. Вы можете смотреть ваше приложение в различных окружениях просто меняя фронт-контроллеры в браузере. Для того чтобы отобразить приложение в dev окружении, откройте его при помощи фронт контроллера `app_dev.php`:

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

Если же вы хотите посмотреть, как поведёт себя приложение в продуктовом окружении, вы можете вызвать фронт-контроллер `prod`:

```
http://localhost/app.php/hello/Ryan
```

Так как `prod` окружение оптимизировано для скорости, настройки, маршруты и шаблоны Twig компилируются в плоские PHP классы и кэшируются. Когда вы хотите посмотреть изменения в продуктовом окружении, вам потребуется удалить эти файлы чтобы они пересоздались автоматически:

```
php app/console cache:clear --env=prod --no-debug
```

Примечание: Если вы откроете файл `web/app.php`, вы обнаружите, что он однозначно настроен на использование `prod` окружения:

```
$kernel = new AppKernel('prod', false);
```

Вы можете создать новый фронт-контроллер для нового окружения просто скопировав этот файл и изменив `prod` на другое значение.

Примечание: Тестовое окружение (`test`) используется при запуске автотестов и его нельзя напрямую открыть через браузер. Подробнее об это можно почитать в главе *Тестирование*.

Настройка окружений

Класс `AppKernel` отвечает за загрузку конфигурационных файлов:

```
<?php

// app/AppKernel.php
public function registerContainerConfiguration(LoaderInterface $loader)
{
    $loader->load(__DIR__.'/config/config_'.$this->getEnvironment().'.yml');
}
```

Вы уже знаете, что расширение `.yml` может быть изменено на `.xml` или `.php`, если вы предпочитаете использовать XML или PHP для файлов конфигурации. Имейте также в виду, что каждое окружение загружает свои собственные настройки. Рассмотрим конфигурационный файл для `dev` окружения.

- *YAML*

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

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

# ...
```

- *XML*

```
<!-- app/config/config_dev.xml -->
<imports>
    <import resource="config.xml" />
</imports>

<framework:config>
    <framework:router resource="%kernel.root_dir%/config/routing_dev.xml" />
    <framework:profiler only-exceptions="false" />
</framework:config>

<!-- ... -->
```

- *PHP*

```
<?php

// app/config/config_dev.php
$loader->import('config.php');

$container->loadFromExtension('framework', array(
    'router'   => array('resource' => '%kernel.root_dir%/config/routing_dev.php'),
    'profiler' => array('only-exceptions' => false),
));

// ...
```

Ключ `imports` похож по действию на выражение `include` в PHP и гарантирует что главный конфигурационный файл (`config.yml`) будет загружен в первую очередь. Остальной код корректирует конфигурацию по умолчанию для увеличения порога логгирования и прочих настроек, специфичных для процесса разработки.

Оба окружения – `prod` и `test` следуют той же модели: каждое окружение импортирует базовые настройки и модифицирует их значения для своих нужд. Это соглашение позволяет повторно использовать большую часть настроек и изменять лишь те из них, которые требует окружение.

2.4.6 Заключение

Поздравляем! Вы усвоили все фундаментальные аспекты Symfony2 и обнаружили, какими лёгкими и в то же время гибкими они могут быть. И, поскольку на подходе ещё *много интересного*, обязательно запомните следующие положения:

- создание страниц – это три простых шага, включающих **маршрут**, **контроллер** и (опционально) **шаблон**;
- каждое приложение должно содержать 4 основных директории: `web/` (ассеты и фронт-контроллеры), `app/` (настройки), `src/` (ваши пакеты), и `vendor/` (сторонние библиотеки) (также ещё имеется директория `bin/` для обновления вендоров);
- Каждая функция в Symfony2 (включая ядро фреймворка) должна располагаться внутри *пакета*, который представляет собой структурированный набор файлов, реализующих эту функцию;
- *настройки* каждого пакета располагаются в директории `app/config` и могут быть записаны в формате YAML, XML или PHP;
- каждое **окружение** доступно через свой отдельный фронт-контроллер (например `app.php` и `app_dev.php`) и загружает отдельный файл настроек;

Далее, каждая глава книги познакомит вас с все более и более мощными инструментами и более глубокими концепциями. Чем больше вы знаете о Symfony2, тем больше вы будете ценить гибкость его архитектуры и его обширные возможности для быстрой разработки приложений.

2.5 Контроллер

Контроллер - это PHP-функция, которую вы создаёте, чтобы получить информацию из HTTP запроса и на её основе создать HTTP ответ в виде объекта `Response`. Ответ может быть HTML страницей, XML-документом, сериализованным JSON-массивом, изображением, перенаправлением, ошибкой 404, всем чем угодно, о чём вы только могли мечтать. Контроллер содержит любую логику *вашего приложения*, необходимую для того, чтобы отобразить содержимое страницы.

Для того чтобы увидеть, насколько просто этого можно добиться, давайте рассмотрим контроллер Symfony2 в действии. Следующий контроллер отобразит страницу, которая всего-навсего напечатает `Hello world!`:

```
<?php

use Symfony\Component\HttpFoundation\Response;

public function helloAction()
{
```

```
    return new Response('Hello world!');
}
```

Цель у контроллера всегда одна: создать и вернуть объект `Response`. Следуя этой цели, контроллер может читать информацию из запроса, загружать ресурсы из базы данных, отправлять email или же записывать информациюю в сессию пользователя. Но всегда, в конечном итоге, контроллер вернёт объект `Response`, который будет отправлен клиенту.

Здесь нет никакой магии или других требований, о которых стоило бы беспокоиться! Вот несколько типичных примеров:

- *Контроллер A* создаёт объект `Response`, содержащий контент для главной страницы сайта.
- *Контроллер B* получает из запроса параметр `slug` для того, чтобы загрузить запись блога из базы данных и создать объект `Response`, отображающий этот блог. Если указанный `slug` не может быть найден в базе, он создаёт и возвращает объект `Response` со статус-кодом 404 (не найдено).
- *Контроллер C* обрабатывает отправленную форму контактов. Он читает информацию о форме из запроса, сохраняет контактную информацию в базу данных и отправляет сообщение вебмастеру. Наконец, он создаёт объект `Response`, который перенаправляет браузер клиента на страницу "thank you".

2.5.1 Жизненный цикл Запрос-Контроллер-Ответ

Каждый запрос, обрабатываемый проектом Symfony2, следует одному и тому же простому жизненному циклу. Фреймворк берёт на себя повторяющиеся задачи и, в конце концов выполняет контроллер, который содержит код вашего приложения:

1. Каждый запрос обрабатывается одним фронт-контроллером (например `app.php` или `app_dev.php`), который загружает приложение;
2. `Router` читает информацию из запроса (URI к примеру), ищет подходящий маршрут и получает параметр `_controller` из маршрута;
3. Контроллер, соответствующий маршруту, выполняется и его код формирует объект `Response`;
4. HTTP-заголовки и контент объекта `Response` отправляются обратно клиенту, отправившему изначальный запрос.

Создание страницы - это по сути создание контроллера (#3) и маршрута, который ставит в соответствие контроллеру некий URL (#2).

Примечание: Не смотря на то что "фронт-контроллер" и "контроллер" названы похожим образом, они сильно различаются - об этом мы еще поговорим чуть позже в

этой главе. Фронт-контроллер - это короткий PHP-файл, который находится в web-директории и который обрабатывает все входящие запросы. Типичное приложение имеет продуктовый контроллер (`prod`, как правило `app.php`) и контроллер для разработки (`dev`, как правило `app_dev.php`). И вам скорее всего никогда не придется модифицировать или вообще задумываться о фронт-контроллерах в вашем приложении.

2.5.2 Простой контроллер

В то время как контроллер может быть любой PHP-сущностью, которую можно вызвать (функцией, методом объекта, или же замыканием (`Closure`)), в Symfony2 контроллер - это как правило некий метод объекта контроллера. Контроллеры также называются *действиями* (actions).

```
1  <?php
2
3  // src/Acme/HelloBundle/Controller/HelloController.php
4
5  namespace Acme\HelloBundle\Controller;
6  use Symfony\Component\HttpFoundation\Response;
7
8  class HelloController
9  {
10     public function indexAction($name)
11     {
12       return new Response('<html><body>Hello '.$name.'!</body></html>');
13     }
14 }
```

Совет: Обратите внимание, что *контроллер* - это метод `indexAction`, который расположен внутри *класса контроллера* (`HelloController`). Смотрите не путайтесь: *класс контроллера* - это просто удобный способ сгруппировать несколько контроллеров/действий вместе. Обычно класс контроллера содержит несколько контроллеров/действий (например `updateAction`, `deleteAction` и т.д.).

Этом контроллере нет ничего сложного, но давайте разберём подробнее:

- *строка 5*: Symfony2 использует преимущества пространств имён PHP 5.3. Ключевое слово `use` импортирует класс `Response`, который контроллер должен вернуть.
- *строка 8*: Имя класса это результат объединения имени контроллера (`Hello`) и слова `Controller`. Это очередная договорённость, которая позволяет обеспечить единообразие в именовании контроллеров и позволяет ссылаться на класс только по первой части наименования (здесь это будет `Hello`) в конфигурации маршрутизатора.

- *line 10*: Каждое действие в классе контроллера имеет суффикс `Action` и упоминается в настройках маршрутизатора только по имени (`index`). В следующей секции вы создадите маршрут, который привяжет URI к этому действию. Вы узнаете как заполнитель для имени в маршруте - `{name}` - станет аргументом метода действия (`$name`).
- *line 12*: Контроллер создаёт и возвращает объект `Response`.

2.5.3 Соответствие URL Контроллеру

Новый контроллер возвращает простую HTML-страницу. Для того чтобы увидеть эту страницу в вашем браузере, вам надо создать маршрут, который устанавливает соответствие между некоторым шаблоном URL и контроллером:

- *YAML*

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

- *XML*

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

- *PHP*

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

Теперь при запросе URI `/hello/ryan` теперь выполняется контроллер `HelloController::indexAction()` и присваивает переменной `$name` значение `ryan`. Создание страницы по сути подразумевает всего лишь создание метода контроллера и соответствующего маршрута.

Обратите внимание на синтаксис, при помощи которого маршрут ссылается на контроллер: `AcmeHelloBundle:Hello:index`. Symfony2 использует простую строковую нотацию для создания ссылок на различные контроллеры. Этот очень простой синтаксис сообщает Symfony2 что класс контроллера с именем `HelloController` расположен в пакете `AcmeHelloBundle`. Затем выполняется метод `indexAction()`.

Более подробно о формате строк, используемых для создания ссылок на различные контроллеры можно почитать здесь: *Шаблон Именования Контроллера*.

Примечание: В этом примере конфигурация маршрутизатора выполняется непосредственно в директории `app/config/`. На практике более удобен способ, когда ваши маршруты размещаются в пакете, которому соответствуют. Более подробно этот способ рассматривается здесь: *Подключение внешних ресурсов для маршрутизации*.

Совет: Подробно вопросы маршрутизации рассматриваются в главе *Маршрутизация*.

Параметры маршрута в качестве аргументов Контроллера

Вы уже знаете, что параметр `_controller` со значением `AcmeHelloBundle:Hello:index` ссылается на метод `HelloController::indexAction()`, который расположен в пакете `AcmeHelloBundle`. Также интерес представляют аргументы, которые передаются в этот метод:

```
<?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)
    {
      // ...
    }
}
```

Контроллер имеет единственный аргумент - `$name`, который соответствует параметру `{name}` из маршрута (в нашем примере - `ryan`). Фактически, когда контроллер выполняется, Symfony2 каждому аргументу контроллера ставит в соответствие параметр из маршрута. Взгляните на пример:

- *YAML*

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

- *XML*

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

- *PHP*

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

Контроллер для этого примера принимает несколько аргументов:

```
<?php
public function indexAction($first_name, $last_name, $color)
{
    // ...
}
```

Обратите внимание, что оба заполнителя для переменных (`{first_name}`, `{last_name}`), как и переменная по умолчанию `color` - доступны в качестве аргументов в контроллере. Когда совпадает маршрут, заполнители переменных объединяются с `defaults` в один массив, который становится доступен в вашем контроллере.

Настройка соответствия параметров маршрута аргументам контроллера проста, нужно лишь следовать нижеперечисленным рекомендациям во время разработки:

- **Порядок аргументов контроллера не имеет значения**

  Symfony в состоянии установить соответствие между именами параметров маршрута и сигнатурой метода в контроллере. Другими словами, это работает таким образом, что параметр `{last_name}` соответствует аргументу `$last_name`. Аргументы контроллера менять местами и он всё равно будет работать:

  ```
  <?php
  public function indexAction($last_name, $color, $first_name)
  {
      // ..
  }
  ```

- **Каждый обязательный аргумент контроллера должен соответствовать параметру маршрута**

  Следующий пример вызовет исключение `RuntimeException`, так как в маршруте не определён параметр `foo`:

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

Для того чтобы это работало, нужно сделать параметр опциональным. Следующий пример не будет вызывать исключительной ситуации:

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

- **Параметры маршрута не обязательно должны быть представлены в виде аргументов контроллера**

  Если, к примеру, параметр `last_name` не нужен в контроллере, его можно опустить:

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

Совет: Каждый маршрут имеет специализировнный параметр `_route`, который содержит значение равное его имени (например `hello`). Обычно это значение не используется, но, тем не менее, этот параметр также доступен в качестве аргумента контроллера.

`Request` как аргумент Контроллера

Для большего удобства, вы также можете передать объект `Request` в качестве аргумента в ваш контроллер. Это особенно удобно при работе с формами:

```
<?php
use Symfony\Component\HttpFoundation\Request;

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

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

2.5.4 Базовый класс контроллера

Symfony2 включает базовый класс `Controller`, который оказывает помощь в выполнении наиболее типичных задач контроллера и предоставляет вашему контроллеру доступ к любому ресурсу, который может портребоваться. Осуществляя наследование от класса `Controller` вы получите в своё распоряжение некоторое число методов-помощников.

Добавьте выражение `use` в начале класса контроллера и модифицируйте `HelloController`, чтобы он наследовался от `Controller`:

```
<?php
// 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>');
    }
}
```

Эти изменения на самом деле ничего не меняют в логике работы вашего контроллера. В следующей секции вы узнаете о тех методах-помощниках, которые предоставляет базовый класс. Эти методы по сути являются обёртками для базового функционала Symfony2 который доступен вам в любом случае - с использованием базового класса `Controller` или же без него. Самый лучший путь для того чтобы увидеть базовые функции в действии - заглянуть в код класса `Symfony\Bundle\FrameworkBundle\Controller\Controller` самостоятельно.

Совет: Наследование от базового класса совершенно *не обязательно* в Symfony2. Этот касс содержит удобные методы-ярлыки, но ничего обязательного. Вы также можете отнаследоваться от класса `Symfony\Component\DependencyInjection\ContainerAware`. Объект service container'а будет доступен черз свойство `container`.

Примечание: Вы также можете объявить контроллер в качестве сервиса: `</cookbook/controller/service>`.

2.5.5 Контроллер, Базовые операции

Хотя, виртуально контроллер ничего делать не обязан, в основном контроллеры выполняют одни и те же задачи снова и снова. Эти задачи, такие как перенаправление, переадресация, отображение шаблона и доступ к основным сервисам, в Symfony2 выполнять очень легко.

Перенаправление (redirecting)

Если вы хотите перенаправить пользователя на другую страницу, используйте метод `redirect()`:

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

Метод `generateUrl()`, это всего-лишь функция помощник, которая генерирует URL для заданного маршрута. Более подробно этот вопрос рассматривается в главе *Маршрутизация*.

По умолчанию, метод `redirect()` выполняет перенаправление с HTTP статус-кодом 302 (временное перенаправление). Для того, чтобы выполнить постоянное перенаправление (со статус-кодом 301), необходимо добавить второй аргумент:

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

Совет: Метод `redirect()` - это просто ярлычок для операции создания объекта `Response`, который специализируется на перенаправлении пользователя. Он эквивалентен следующему коду:

```
<?php
use Symfony\Component\HttpFoundation\RedirectResponse;

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

Контроллер, Переадресация (forwarding)

Вы также легко можете переадресовать запрос на другой контроллер внутри системы, используя метод `forward()`. Вместо того, чтобы выполнить перенаправление браузера пользователя, этот метод выполняет внутренний подзапрос и вызывает указанный контроллер. Метод `forward()` возвращает объект `Response`, который возвращает контроллер, на который осуществлялась переадресация:

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

    // Здесь можно модифицировать $response или же сразу вернуть его пользователю

    return $response;
}
```

Обратите внимание, что метод *forward()* использует для указания контроллера тот же формат строки, который используется в конфигурации маршрутов. Таким образом, целью переадресации будет `HelloController` из пакета `AcmeHelloBundle`. Массив, передаваемый методу в качестве параметра, будет конвертирован в параметры целевого контроллера. Такой же интерфейс используется при встраивании контроллеров в шаблоны (см. *Внедрение контроллеров*). Метод целевого контроллера должен выглядеть следующим образом:

```
<?php
public function fancyAction($name, $color)
{
    // ... create and return a Response object
}
```

И, как и в случае создания контроллера для маршрута, порядок аргументов для `fancyAction` не имеет значения. Symfony2 устанавливает соответствие по именам ключей (например `name`) и именам параметров (например `$name`). Если вы изменяете порядок следования аргументов, Symfony2 также будет присваивать верные значения каждой переменной.

Совет: Как и прочие методы базового контроллера, метод `forward` - это просто ярлык к базовому функционалу Symfony2. Переадресация может быть выполнена напрямую через сервис `http_kernel`. При переадресации возвращается объект `Response`:

```
<?php
$httpKernel = $this->container->get('http_kernel');
```

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

Рендеринг Шаблонов

Хотя это и не является требованием, большинство контроллеров в конце концов будут отображать (рендерить) шаблон, который отвечает за генерацию HTML (или данных в другом формате) для контроллера. Метод `renderView()` рендерит шаблон и возвращает его содержимое. Контент из шаблона может быть использован для создания объекта `Response`:

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

return new Response($content);
```

Эти операции могут быть выполнены за один шаг при помощи метода `render()`, который возвращает объект `Response`, содержащий контент шаблона:

В обоих случаях, будет отображен шаблон `Resources/views/Hello/index.html.twig` из пакета `AcmeHelloBundle`.

Шаблонизатор Symfony более подробно рассматривается в главе о *Шаблонах*

Совет: Метод `renderView` - это по сути ярлык для быстрого использования шаблонизатора. Шаблонизатор также можно использовать напрямую:

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

Доступ к сервисам

При наследовании от базового контроллера, вы можете получить доступ к любому сервису Symfony2 при помощи метода `get()`. Ниже представлены основные сервисы, которые вам могут быть полезны:

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

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

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

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

В Symfony2 по умолчанию определена куча сервисов и вы вольны определить ещё столько же собственных. Для того чтобы отобразить список доступных сервисов, используйте консольную команду `container:debug`:

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

Больше данных о сервисах вы можете почерпнуть из главы *Service container*.

2.5.6 Разбираемся с ошибками и 404 страница

Когда что-либо не может быть найдено, вы должны вернуть статус-код 404. Для того чтобы это сделать, вы можете сгенерировать особый тип исключения. Если вы унаследовали контроллер от базового, выполните следующее:

```
<?php
public function indexAction()
{
    $product = // тут получаем объект из базы данных
    if (!$product) {
        throw $this->createNotFoundException('Продукт не существует');
    }

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

Метод `createNotFoundException()` создаёт особый объект `NotFoundHttpException`, который в конечном итоге провоцирует возврат HTTP 404 внутри Symfony.

Конечно, вы вольны вызывать любую исключительную ситуацию в вашем контроллере - Symfony2 автоматически вернёт HTTP статус-код 500.

```
throw new \Exception('Что-то пошло не так!');
```

В любом случае, пользователь увидит страницу с той или иной ошибкой, а разработчику (при использовании dev-окружения) будет показана страница с полной отладочной информацией. Эти страницы ошибок могут быть изменены. Более подробно об этом написано в "книге рецептов": "*Как создать собственные страницы ошибок*".

2.5.7 Работа с Сессиями

Symfony2 предоставляет вам объект, для работы с сессиями, который вы можете использовать для хранения информации о пользователе (если он реальный человек, автома-

тический бот или же веб-сервис) между запросами. По умолчанию, Symfony2 сохраняет атрибуты в куках (cookie), используя нативные сессии PHP.

Сохранение и получение информации из сессии можно использовать из любого контроллера:

```
$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');

// set the user locale
$session->setLocale('fr');
```

Эти атрибуты будут соответствовать конкретному пользователю, пока существует его сессия.

Flash-сообщения

Вы также можете сохранять небольшие сообщения, которые сохраняются в пользовательской сессии между двумя запросами. Эти сообщения удобно использовать при обработке форм: вы хотите выполнить перенаправление и отобразить особое сообщение при *следующем* запросе. Такие сообщения называются flash-сообщениями.

Например, представьте, что вы обрабатываете отправку формы:

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

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

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

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

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

После обработки запроса контроллер устанавливает flash-сообщение `notice` и выполняет перенаправление. Имя (`notice`) не устанавливается жёстко - это лишь обозначение

типа сообщения.

В шаблоне следующего действия вы можете использовать следующий код для отображения сообщения `notice`:

- *Twig*

```
{% if app.session.hasFlash('notice') %}
    <div class="flash-notice">
        {{ app.session.flash('notice') }}
    </div>
{% endif %}
```

- *PHP*

```
<?php if ($view['session']->hasFlash('notice')): ?>
    <div class="flash-notice">
        <?php echo $view['session']->getFlash('notice') ?>
    </div>
<?php endif; ?>
```

По умолчанию, flash-сообщения должны жить ровно один запрос. Они разработаны именно для того, чтобы использоваться во время перенаправлениями так как показано в этом примере.

2.5.8 Объект Ответа

К контроллеру предъявляется лишь одно требование - вернуть объект `Response`. Класс `Symfony\Component\HttpFoundation\Response` представляет собой PHP-абстракцию HTTP-ответа - текстового сообщения, состоящего из HTTP-заголовков и контента, который возвращается клиенту:

```
// создаётся простой объект Response со статус-кодом 200 (по умолчанию)
$response = new Response('Hello '.$name, 200);

// создаётся JSON-ответ со статус-кодом 2000
$response = new Response(json_encode(array('name' => $name)));
$response->headers->set('Content-Type', 'application/json');
```

Совет: `headers` - это объект `Symfony\Component\HttpFoundation\HeaderBag`, содержащий методы для чтения и изменения заголовков ответа `Response`. Имена заголовков нормализованы, так что `Content-Type`, `content-type` и даже `content_type` эквивалентны.

2.5.9 Объект запроса

Помимо значений заполнителей из маршрута, контроллер также имеет доступ к объекту `Request`, когда он является наследником базового класса `Controller`:

```
$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
```

Подобно объекту `Response`, заголовки запроса хранятся в объекте `HeaderBag` и также легко доступны.

2.5.10 Заключение

Когда вы создаёте страницу, в конечном итоге должны написать код, который содержит логику этой страницы. В Symfony эта логика называется "контроллером", и представляет собой PHP-функцию, которая выполняет все необходимые действия для того чтобы вернуть объект `Response`, который будет отправлен пользователю.

Для того, чтобы сделать жизнь легче, вы можете отнаследоваться от класса `Controller`, который содержит методы для типичных задач, решаемых контроллером. Например, так как вы должны вернуть HTML код - вы можете использовать метод `render()` и вернуть контент шаблона.

В других главах вы узнаете как контроллер может быть использован для сохранения и получения объектов из базы данных, обрабатывать отправку форм, работать с кэшем и многое другое.

2.5.11 Дополнительно в книге рецептов:

- *Как создать собственные страницы ошибок*
- *Как определять Контроллеры в качестве сервисов*

2.6 Маршрутизация

Каждое серьёзное приложение должно обязательно иметь "красивые" URL. Это означает, что это приложение должно оставить позади страшненькие URL типа

`index.php?article_id=57` в пользу таких `/read/intro-to-symfony`.

Однако гибкость в этом вопросе - ещё более важна, нежели красота. Что, если вам нужно изменить URL `/blog` на `/news`? Сколько ссылок вам придётся отыскать и обновить для этого? Если же вы используете маршрутизатор Symfony, подобные изменения делать легко.

Маршрутизатор Symfony2 позволяет вам определить креативные URL, которые вы сможете привязать к различным областям вашего приложения. По прочтению этой главы вы сможете делать следующее:

- Создавать сложные маршруты, соответствующие контроллерам;
- Генерировать URL в шаблонах и контроллерах;
- Загрузить ресурсы для маршрутизации из пакетов (или из любых других источников);
- Отлаживать маршруты.

2.6.1 Маршрутизация в действии

Маршрут по сути это связующее звено между шаблоном URL и контроллером. Например, предположим, вам нужно искать URL похожие на `/blog/my-post` или `/blog/all-about-symfony` и отправлять их на обработку в контроллер, который найдёт и отобразит эти записи блога. Соответствующий этой задаче маршрут - прост:

- *YAML*

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

- *XML*

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

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

- *PHP*

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

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

return $collection;
```

Шаблон, определяемый маршрутом `blog_show` работает как выражение `/blog/*`, где метасимволом является имя `slug`. Для URL `/blog/my-blog-post` переменная `slug` получает значение `my-blog-post`, которое будет доступно для использования в контроллере.

Параметр `_controller` - это служебный ключ, который сообщает Symfony, какой именно контроллер должен быть выполнен, когда маршрут совпадает с URL. Строка `_controller`, называется *логическим именем*. Логическое имя указывает на некоторый PHP-класс и его метод:

```php
<?php
// 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 = // используйте переменную $slug, для того чтобы получить запись из базы данных

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

Поздравляем! Вы только что создали ваш первый маршрут и связали его с контроллером. Теперь, когда вы посетите страницу `/blog/my-post`, будет выполнен контроллер `showAction` и переменная `$slug` будет равна `my-post`.

Это и есть цель маршрутизатора Symfony2: устанавливать соответствие между URL запроса и контроллером. Далее в этой главе вы узнаете все возможные трюки, которые позволяют легко писать маршруты даже для сложных URL.

2.6.2 Маршрутизация; Что под капотом

Когда создаётся запрос к вашему приложению, он содержит точный адрес ресурса, который запрашивается клиентом. Этот адрес называется URL (или URI) и может выглядеть следующим образом: `/contact`, `/blog/read-me` или ещё каким-то похожим образом. Давайте рассмотрим следующий HTTP-запрос в качестве примера:

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

Цель системы маршрутизации Symfony2 - разбор этого URL и определение того, какой контроллер должен быть выполнен. Процесс целиком выглядит так:

1. Запрос обрабатывается фронт-контроллером Symfony2 (например `app.php`);
2. Ядро Symfony2 (`Kernel`), вызывает маршрутизатор для анализа запроса;
3. Маршрутизатор устанавливает соответствие между входящим URL и некоторым маршрутом и возвращает информацию о маршруте, включая данные о том, какой контроллер требуется выполнить;
4. Ядро Symfony2 выполняет контроллер, который в конечном итоге возвращает объект `Response`.

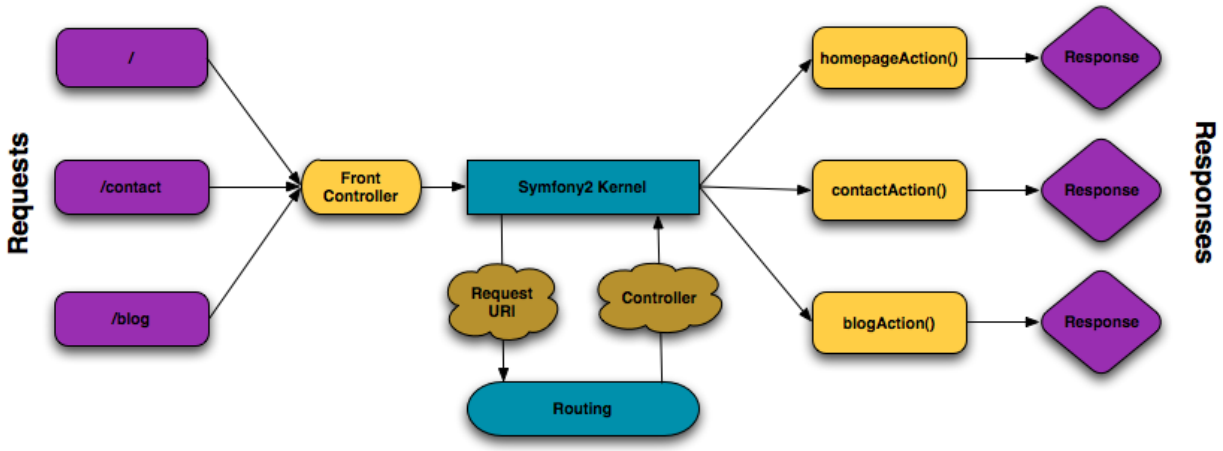


Рис. 2.2: Слой маршрутизации - это инструмент, который транслирует входящий URL в контроллер, который нужно выполнить для его обработки.

2.6.3 Создание маршрутов

Symfony загружает все маршруты, определённые для вашего приложения, из одного файла настроек. Как правило, этот файл называется `app/config/routing.yml`, но при желании наименование файла конфигурации можно изменить на другое (в том числе на файл формата XML или PHP) в конфигурационном файле приложения:

- *YAML*

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

- *XML*

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

- *PHP*

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

Совет: Не смотря на то, что все маршруты загружаются из одного файла, обычной практикой является подключение дополнительных ресурсов внутри этого файла (см. секцию *Подключение внешних ресурсов для маршрутизации*).

Базовая настройка маршрута

Определить новый маршрут не сложно, типичное приложение будет иметь много различных маршрутов. Самый простой маршрут состоит из двух частей: шаблона URL (`pattern`) и массива `defaults`:

- *YAML*

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

- *XML*

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

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

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

</routes>
```

- *PHP*

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

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

return $collection;
```

Этот маршрут соответствует главной странице (/) и ставит ей в соответствие контроллер `AcmeDemoBundle:Main:homepage`. Symfony2 переводит строку `_controller` в имя функции, которую необходимо выполнить. Этот процесс будет объясняться в секции *Шаблон Именования Контроллера*.

Маршрутизация и Заполнители (Placeholders)

Конечно же система маршрутизации поддерживает и более интересные маршруты. Многие маршруты будут содержать один или более заполнителей (placeholders):

- *YAML*

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

- *XML*

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

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

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

- *PHP*

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

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

return $collection;
```

Шаблон будет соответствовать любому URL похожему на `/blog/*`. Что ещё более важно - значение, соответствующее заполнителю `{slug}`, будет доступно в вашем контроллере. Другими словами, если дан URL `/blog/hello-world`, переменная `$slug` со значением `hello-world` будет доступна в контроллере. Эту возможность можно использовать, например, для загрузки записи блога, соответствующей этой строке.

Тем не менее, этот шаблон *не будет соответствовать* URL `/blog`. Это вызвано тем фактом, что заполнитель по умолчанию является обязательным параметром. Однако, если добавить заполнителю значение по умолчанию в массив `defaults`.

Обязательные и Опциональные Заполнители

Для того, чтобы разнообразить процесс, давайте создадим новый маршрут, который отображает список всех записей в блоге для нашего воображаемого приложения:

- *YAML*

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

- *XML*

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

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

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

- *PHP*

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

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

return $collection;
```

Пока что этот маршрут выглядит проще простого - он не содержит заполнителей и соответствует лишь одному URL `/blog`. Ну а если вам потребуется, чтобы данный маршрут поддерживал постраничную навигацию и URL `/blog/2` отображал вторую страницу с записями блога? Добавим к маршруту заполнитель `{page}`:

- *YAML*

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

- *XML*

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

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

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

- *PHP*

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

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

return $collection;
```

Подобно заполнителю {slug}, значение соответствующее {page} будет доступно внутри контроллера. Это значение может быть использовано для того, чтобы определить, какой набор записей блога отобразить для данной страницы.

Но погодите-ка! Так как заполнитель по умолчанию обязателен, маршрут теперь не сможет соответствовать просто /blog. Вместо этого, если вы захотите отобразить первую страницу, вам нужно будет использовать URL /blog/1! Поскольку это совершенно неприемлемо, потребуется изменить параметр {page} и сделать его опциональным. Сделать это можно, включив его в массив defaults:

- *YAML*

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

- *XML*

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

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

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

- *PHP*

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

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

return $collection;
```

Добавив page в массив defaults, вы сделали заполнитель {page} необязательным. URL /blog будет соответствовать маршруту и значение параметра page будет равно 1. URL /blog/2 также будет соответствовать этому маршруту, присваивая параметру page значение 2. Отлично.

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

Добавляем Ограничения

Давайте взглянем на маршруты, которые мы добавили ранее:

- *YAML*

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

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

- *XML*

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

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

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

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

- *PHP*

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

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

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

return $collection;
```

Можете определить тут проблему? Обратите внимание, что оба маршрута имеют похожие шаблоны и соответствуют URL вида `/blog/*`. Маршрутизатор Symfony всегда будет выбирать *первый* совпавший маршрут, который он найдёт. Другими словами, маршрут `blog_show` *никогда* не совпадёт и не будет вызван соответствующий контроллер. Вместо этого URL вида `/blog/my-blog-post` будет соответствовать первому маршруту (`blog`) и возвращать бессмысленное для параметра `{page}` значение `my-blog-post`.

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

Решением этой проблемы является добавление *ограничений* в маршрут. Маршруты в этом примере будут работать, если шаблон `/blog/{page}` будет соответствовать URL лишь в том случае, когда `{page}` будет целым числом. К счастью, ограничения в виде регулярных выражений легко могут быть добавлены к любому параметру. Например:

- *YAML*

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

- *XML*

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

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

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

- *PHP*

```
<?php
use Symfony\Component\Routing\RouteCollection;
```

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

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

return $collection;
```

Ограничение `\d+` - это регулярное выражение, которое сообщает маршрутизатору, что значение параметра `{page}` должно быть числовым. Маршрут `blog` по-прежнему будет соответствовать URL вида `/blog/2` (так как 2 это число), но он уже не будет соответствовать URL вида `/blog/my-blog-post` (так как `my-blog-post` *не является числом*).

В результате URL `/blog/my-blog-post` будет соответствовать маршруту `blog_show`.

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

Более ранний маршрут всегда выигрывает

Что же означает тот факт, что порядок маршрутов очень важен? Если маршрут `blog_show` будет расположен выше маршрута `blog`, то URL `/blog/2` будет соответствовать маршруту `blog_show` вместо маршрута `blog` так как параметр `{slug}` не имеет ограничений. Используя правильный порядок и разумные ограничения вы сможете сделать всё что вам угодно.

Так как ограничения для параметров - это регулярные выражения, сложность и гибкость каждого ограничения лежит на вашей совести. Предположим, что главная страница вашего приложения доступна на двух различных языках, в зависимости от URL:

- *YAML*

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

- *XML*

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

<routes xmlns="http://symfony.com/schema/routing"
```

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

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

- *PHP*

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

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

return $collection;
```

Для входящих запросов, часть URL, соответствующая `{culture}` должна удовлетворять регулярному выражению `(en|fr)`:

/	{culture} = en
/en	{culture} = en
/fr	{culture} = fr
/es	*не соответствует маршруту*

Добавляем Ограничения для HTTP-метода

В дополнение к URL, вы также можете проверять *HTTP-метод* входящего запроса (GET, HEAD, POST, PUT, DELETE). Предположим у вас есть форма контактов с двумя контроллерами - один для отображения формы (GET запрос) и другой - для обработки формы, когда она отправлена пользователем (POST запрос). Ограничения для HTTP-метода можно задать следующим образом:

- *YAML*

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

```
    requirements:
        _method:  GET

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

- *XML*

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

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

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

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

- *PHP*

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

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

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

return $collection;
```

Пренебрегая тем, что оба представленных выше маршрута имеют идентичные шаблоны (`/contact`), первый маршрут будет соответствовать только GET-запросам, а второй, в свою очередь, будет соответствовать только POST-запросам. Это означает, что вы сможете отображать и отправлять форму, используя один и тот же URL и использовать различные контроллеры для каждого из этих действий.

Примечание: Если ограничения на `_method` не указаны, маршрут будет соответствовать *любому* методу.

Как и любые другие ограничения, ограничения для `_method` обрабатываются как регулярные выражения. Для того, чтобы соответствовать как `GET` *так и* `POST` запросам, вы можете использовать ограничение `GET|POST`.

Продвинутая Маршрутизация в Примерах

На текущий момент, вы имеете всю необходимую информацию, для создания сложных структур маршрутизации в Symfony. Ниже мы покажем вам, насколько гибкой может быть система маршрутизации:

- *YAML*

```
article_show:
  pattern:  /articles/{culture}/{year}/{title}.{_format}
  defaults: { _controller: AcmeDemoBundle:Article:show, _format: html }
  requirements:
      culture:  en|fr
      _format:  html|rss
      year:     \d+
```

- *XML*

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

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

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

- *PHP*

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

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

return $collection;
```

Как вы можете видеть, этот маршрут сработает лишь в том случае, если `{culture}` в URL будет либо `en` либо `fr` и `{year}` будет числом. Этот маршрут также показывает, что вы можете использовать помимо слэша (`/`) точку между двумя заполнителями. URL, соответствующий этому маршруту может выглядеть следующим образом:

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

Особый параметр маршрута `_format`

Этот пример также использует особый параметр маршрута - `_format`. При использовании этого параметра, соответствующее значение становится "форматом запроса" в объекте `Request`. В конечном счёте, формат запроса используется для таких действий, как установка `Content-Type` для ответа (например формат запроса `json` трансформируется в `Content-Type application/json`). Этот параметр также может быть использован в контроллере для отображения различных шаблонов для каждого возможного его значения. Параметр `_format` является весьма удобным решением при необходимости отображать один и тот же контент в различных форматах.

Специальные параметры маршрута

Как вы наверное обратили внимание, каждый параметр маршрута или значение по умолчанию в конечном итоге доступен в виде аргумента в методе контроллера. В дополнение к этому есть также три специальных параметра, каждый из которых добавляет уникальные возможности внутри вашего приложения:

- `_controller`: Как вы уже знаете, этот параметр используется для того, чтобы определить какой контроллер будет выполнен, когда маршрут совпадает с URL;
- `_format`: Используется для определения запрашиваемого формата (см. *параметр маршрута _format*);
- `_locale`: Используется для того, чтобы установить локаль в сессии (см. *локаль в URL*);

2.6.4 Шаблон Именования Контроллера

Каждый маршрут должен иметь параметр `_controller`, который определяет, какой именно контроллер будет выполнен, когда соответствующий маршрут совпадёт с URL. Этот параметр использует простой строковый шаблон, именуемый *логическим именем контроллера*, которому Symfony ставит в соответствие PHP метод и класс. Шаблон состоит из трёх частей, разделённых двоеточием:

пакет:**контроллер**:**действие**

Например, если `_controller` имеет значение `AcmeBlogBundle:Blog:show`, то это означает следующее:

Bundle	Controller Class	Method Name
AcmeBlogBundle	BlogController	showAction

Контроллер может выглядеть так:

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

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

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

Обратите внимание, что Symfony добавляет строку `Controller` у имени класса (`Blog` => `BlogController`) и `Action` к имени метода (`show` => `showAction`).

Вы также можете ссылаться на этот класс, используя полное имя класса и метода: `Acme\BlogBundle\Controller\BlogController::showAction`. Но, если вы следуете нескольким простым соглашениям, логическое имя будет более удобно.

Примечание: В дополнение к использованию логического имени и полного имени класса, Symfony поддерживает третий тип ссылок на контроллер. Этот метод использует одно двоеточие в качестве разделителя (например `service_name:indexAction`) и ссылается на котроллер, определённый как сервис (см. *Как определять Контроллеры в качестве сервисов*).

2.6.5 Параметры маршрута и Аргументы контроллера

Параметры маршрута (например `{slug}`) очень важны, так как каждый параметр будет доступен в качестве аргумента в методе-контроллере:

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

Фактически, все `defaults` объединяются со значениями параметров и формируют один массив. Каждый ключ такого массива доступен в качестве аргумента внутри контроллера.

Другими словами, для каждого аргумента вашего метода-контроллера, Symfony ищет параметр маршрута с тем же именем и присваивает его значение этому аргументу. В продвинутом примере ранее любая комбинация (в любом порядке) следующих переменных может быть использована в качестве аргументов метода `showAction()`:

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

Так как заполнители и массив `defaults` объединяются, даже переменная `$_controller` становится доступна. Более подробно это описано в секции *Параметры маршрута в качестве аргументов Контроллера*.

Совет: Вы также можете использовать переменную `$_route`, которая содержит имя соответствующего маршрута.

2.6.6 Подключение внешних ресурсов для маршрутизации

Все маршруты загружаются посредством одного конфигурационного файла, обычно это файл `app/config/routing.yml` (см. Создание маршрутов выше). На практике же вы вероятно захотите загружать маршруты из других мест, например из ваших пакетов. И это становится возможным при помощи "импорта" файла маршрутов:

- *YAML*

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

- *XML*

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

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

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

- *PHP*

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

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

return $collection;
```

Примечание: При импорте ресурсов из YAML, ключ (например `acme_hello`) не имеет практического значения. Просто убедитесь, что этот ключ уникален и нигде далее не переопределяется.

Ключ `resource` загружает указанный ресурс с маршрутами. В данном примере ресурс - это полный путь к файлу, где `@AcmeHelloBundle` это ярлык, означающий путь к пакету. Импортируемый файл может выглядеть следующим образом:

- *YAML*

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

- *XML*

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

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

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

- *PHP*

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

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

return $collection;
```

Маршруты из этого файла анализируются и загружаются также, как и основной файл.

Префикс для импортируемого ресурса

Вы также можете указать “префикс” для импортируемого маршрута. Например, предположим, что вы хотите чтобы маршрут `acme_hello` имел следующий вид: `/admin/hello/{name}` вместо обычного `/hello/{name}`:

- *YAML*

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

- *XML*

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

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

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

- *PHP*

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

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

return $collection;
```

Строка `/admin` теперь будет добавлена вначале каждого маршрута, загружаемого из указанного ресурса:

2.6.7 Отображение и Отладка маршрутов

По мере добавления и настройки маршрутов, было бы удобно иметь возможность визуализировать и получать детальную информацию о них. Для того, чтобы увидеть все маршруты вашего приложения, вы можете воспользоваться консольной командой `router:debug`. Выполните эту команду из корня вашего проекта:

```
php app/console router:debug
```

Эта команда отобразит удобный список *всех* настроенных маршрутов вашего приложения:

```
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}
```

Вы также можете получить более подробную информацию о конкретном маршруте, указав его имя после команды `router:debug`:

```
php app/console router:debug article_show
```

2.6.8 Генерация URL

Система маршрутизации также должна позволять генерировать URL. На практике, маршрутизация - это двунаправленная система: устанавливает как соответствие URL с контроллером (+ параметры), так и обратно - превращает маршрут (+ параметры) в URL. Методы **:method:'Symfony\\Component\\Routing\\Router::match'** и **:method:'Symfony\\Component\\Routing\\Router::generate'** формируют эту двунаправленную систему. Рассмотрим маршрут `blog_show`, описанный выше:

```
<?php
$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
```

Для того, чтобы сгенерировать URL, вам необходимо указать имя маршрута (`blog_show`) и параметры, используемые в шаблоне этого маршрута. Имея эту информацию, можно сгенерировать любой URL:

```
<?php
class MainController extends Controller
{
    public function showAction($slug)
    {
      // ...

      $url = $this->get('router')->generate('blog_show', array('slug' => 'my-blog-post'));
    }
}
```

В следующей секции вы узнаете как генерировать URL в шаблоне.

Совет: Если фронтэнд вашего приложения использует AJAX, вы возможно захотите иметь возможность генерировать URL в JavaScript при помощи вашей конфигурации маршрутизатора. И вы таки можете это делать при помощи пакета FOSJsRoutingBundle:

```
var url = Routing.generate('blog_show', { "slug": 'my-blog-post});
```

Подробнее читайте в документации пакета.

Генерация Абсолютных URL

По умолчанию, маршрутизатор генерирует относительные URL (например `/blog`). Для того, чтобы сгенерировать абсолютный URL, просто укажите "true" в качестве третьего аргумента метода `generate()`:

```
<?php
$router->generate('blog_show', array('slug' => 'my-blog-post'), true);
// http://www.example.com/blog/my-blog-post
```

Примечание: Хост, который используется для генерации абсолютного URL - это хост из текущего объекта `Request`. Этот параметр определяется автоматически, основываясь на информации о сервере, которую предоставляет PHP. При создании абсолютных URL для скриптов, запущенных из командной строки, вам необходимо вручную установить желаемый хост для объекта `Request`:

```
$request->headers->set('HOST', 'www.example.com');
```

Генерация URL содержащих строку запроса (Query String)

Метод `generate` принимает массив значений для генерации URL. Если вы передадите лишний (не указанный в определении маршрута) параметр, он будет добавлен как query string:

```
$router->generate('blog', array('page' => 2, 'category' => 'Symfony'));
// /blog/2?category=Symfony
```

Генерация URL в шаблоне

Типичное место, где вам потребуется генерировать URL - это шаблон. Выполнить эту операцию можно, воспользовавшись функцией-помощником:

- *Twig*

  ```
  <a href="{{ path('blog_show', { 'slug': 'my-blog-post' }) }}">
    Read this blog post.
  </a>
  ```

- *PHP*

  ```
  <a href="<?php echo $view['router']->generate('blog_show', array('slug' => 'my-blog-post'))
      Read this blog post.
  </a>
  ```

Абсолютные URL также можно генерировать, но уже при помощи другой функции:

- *Twig*

```
<a href="{{ url('blog_show', { 'slug': 'my-blog-post' }) }}">
  Read this blog post.
</a>
```

- *PHP*

```
<a href="<?php echo $view['router']->generate('blog_show', array('slug' => 'my-blog-post')
    Read this blog post.
</a>
```

2.6.9 Заключение

Маршрутизатор - это система, ставящая в соответствие URL из входящего запроса контроллеру, который будет вызван для обработки запроса. Он позволяет использовать в приложении "красивые" URL и поддерживать приложение независимым от URL-ов. Маршрутизация - это двунаправленный механизм, и позволяет также генерировать URL.

2.6.10 Дополнительная информация из Книги Рецептов

- *Как заставить маршрутизатор всегда использовать HTTPS или HTTP*

2.7 Создание и использование Шаблонов

Как вам известно, *контроллер* отвечает за обработку запросов, получаемых приложением Symfony2. Фактически же, контроллер делегирует большую часть тяжёлой работы другим частям Фреймворка, чтобы код можно было тестировать и использовать повторно. Когда контроллеру требуется сгенерировать HTML, CSS или любой другой контент, он поручает эту работу шаблонизатору. В этой главе вы узнаете как создавать мощные и функциональные шаблоны, которые могут быть использованы для того, чтобы вернуть контент пользователю, создавать тело email-сообщений и многое другое. Вы узнаете, как наследовать шаблоны и как повторно использовать код шаблонов.

2.7.1 Шаблоны

Шаблон - это просто текстовый файл, который может генерировать любой текстовый формат (HTML, XML, CSV, LaTeX и т.д.). Наиболее простой тип шаблона - это *PHP* шаблон - текстовый файл, обрабатываемый PHP, который содержит как собственно текст, так и PHP-код:

```
<!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>
```

В состав Symfony2 входит мощный язык шаблонов, называемый Twig. Twig позволяет создавать лаконичные и читабельные шаблоны, которые более удобны для веб-дизайнеров и, во многом, более функциональны, нежели PHP шаблоны:

```
<!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 определяет два типа специальных синтаксических конструкций:

- `{{ ... }}`: “Напечатать что-либо”: отображает переменную или результат некоторого выражения;
- `{% ... %}`: “Выполнить что-либо”: **тег**, который контролирует логику шаблона; он используется для выполнения выражений, таких как циклы `for`.

Примечание: Есть также третий тип синтаксической конструкции для создания комментариев: `{# это комментарий #}`. Этот синтаксис может быть использован в качестве многострочного комментария как PHP-аналог `/* comment */`.

Twig также содержит **фильтры**, которые модифицируют контент перед его отображением. Следующий пример выведет переменную `title` в верхнем регистре:

```
{{ title | upper }}
```

Twig по умолчанию содержит много тагов (tags) и фильтров (filters). Кроме этого вы можете также создавать свои собственные расширения Twig, если это потребуется.

Совет: Зарегистрировать расширение Twig просто: нужно создать сервис и указать ему *таг* `twig.extension`.

Как вы увидите далее, Twig также поддерживает функции, и вы сможете легко добавлять новые функции. Например, следующий код использует стандартный таг цикла `for` и функцию `cycle` для того, чтобы вывести десять тагов `div` с css-классами `odd` и `even`:

```
{% for i in 0..10 %}
  <div class="{{ cycle(['odd', 'even'], i) }}">
    <!-- some HTML here -->
  </div>
{% endfor %}
```

На протяжении всей главы, примеры шаблонов будут отображаться как в Twig-формате, так и PHP.

Почему Twig?

Twig шаблоны выглядят проще и не обрабатывают PHP код. Это сделано намеренно: система шаблонов Twig задумана для быстрого создания представления, а не для программной логики. Чем больше вы используете Twig, тем более вы будете ценить его и тем более будете получать пользы от такого разделения. И вас будут уважать все веб-дизайнеры в мире.
Twig также умеет делать то, что не умеет PHP, например полноценное наследование шаблонов (Twig-шаблоны компилируются в PHP-классы, которые наследуются как любые другие классы), контроль пробельных символов, песочница (для контроля выполнения "подозрительного" кода в шаблонах), расширение пользовательскими фильтрами и функциями. Twig имеет много небольших особенностей, которые делают написание шаблонов проще и лаконичнее. Взгляните на следующий пример, который комбинирует цикл с логическим выражением `if`:

```
<ul>
    {% for user in users %}
        <li>{{ user.username }}</li>
    {% else %}
        <li>No users found</li>
    {% endfor %}
</ul>
```

Кэширование Шаблонов в Twig

Twig быстр. Каждый Twig-шаблон компилируется в обычный PHP-класс, который и отображается во время выполнения. Компилированные классы расположены в директории `app/cache/{environment}/twig` (здесь `{environment}` - это название окружения, например `dev` или `prod`) и в некоторых случаях может быть полезен при отладке. Об окружениях подробнее написано в разделе *Окружения*.

Когда активен режим `debug` (как правило, в `dev` окружении), шаблон Twig будет автоматически перекомпилироваться каждый раз, когда в нём были произведены изменения. Это означает, что в процессе разработки вы спокойно можете выполнять изменения в шаблонах и видеть эти изменения без необходимости постоянно чистить кэш.

Когда `debug` отключен (как правило, в `prod` окружении), вы должны очищать кэш в директории Twig для того чтобы шаблоны перекомпилировались. Помните об этом при выкладке вашего приложения на сервер.

2.7.2 Наследование шаблонов и Layout

Обычно в проекте шаблоны используют некоторое количество общих элементов, таких как "шапка" (header), "подвал" (footer), боковые панели и т.п. В Symfony2 мы решаем эту

проблему по другому: шаблон может быть декорирован другим шаблоном. Это работает точно также как с классами PHP: наследование шаблонов позволяет вам создавать базовый шаблон - т.н. **layout**, который содержит все базовые элементы вашего сайта, называемые **блоками** (аналогично "PHP-классу с базовыми методами"). Дочерний шаблон может расширять базовый шаблон и переопределять любой его блок (аналогично "дочерний PHP-класс может переопределять некоторые методы родительского класса").

Сначала создайте файл базового шаблона (layout):

- *Twig*

```
{# app/Resources/views/base.html.twig #}
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
        <title>{% block title %}Test Application{% endblock %}</title>
    </head>
    <body>
        <div id="sidebar">
            {% block sidebar %}
            <ul>
                <li><a href="/">Home</a></li>
                <li><a href="/blog">Blog</a></li>
            </ul>
            {% endblock %}
        </div>

        <div id="content">
            {% block body %}{% endblock %}
        </div>
    </body>
</html>
```

- *PHP*

```
<!-- app/Resources/views/base.html.php -->
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
        <title><?php $view['slots']->output('title', 'Test Application') ?></title>
    </head>
    <body>
        <div id="sidebar">
            <?php if ($view['slots']->has('sidebar')): ?>
                <?php $view['slots']->output('sidebar') ?>
```

```
            <?php else: ?>
                <ul>
                    <li><a href="/">Home</a></li>
                    <li><a href="/blog">Blog</a></li>
                </ul>
            <?php endif; ?>
        </div>

        <div id="content">
            <?php $view['slots']->output('body') ?>
        </div>
    </body>
</html>
```

Примечание: Хотя обсуждение наследования шаблонов будет вестись в терминах Twig, в PHP шаблонах используется та же философия.

Этот шаблон определяет базовый скелетон HTML-документа для простой страницы с двумя колонками. В этом примере определено три области `{% block %}` (`title`, `sidebar` и `body`). Каждый блок может быть переопределён дочерним шаблоном, иначе будет сохранена изначальная реализация этих блоков. Это шаблон может также быть отображен самостоятельно. В этом случае блоки `title`, `sidebar` и `body` будут содержать свои значения по умолчанию.

Дочерний шаблон может выглядеть следующим образом:

- *Twig*

```
{# src/Acme/BlogBundle/Resources/views/Blog/index.html.twig #}
{% extends '::base.html.twig' %}

{% block title %}My cool blog posts{% endblock %}

{% block body %}
    {% for entry in blog_entries %}
        <h2>{{ entry.title }}</h2>
        <p>{{ entry.body }}</p>
    {% endfor %}
{% endblock %}
```

- *PHP*

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

<?php $view['slots']->set('title', 'My cool blog posts') ?>
```

```
<?php $view['slots']->start('body') ?>
    <?php foreach ($blog_entries as $entry): ?>
        <h2><?php echo $entry->getTitle() ?></h2>
        <p><?php echo $entry->getBody() ?></p>
    <?php endforeach; ?>
<?php $view['slots']->stop() ?>
```

Примечание: Родительский шаблон определяется благодаря специальному синтаксису (`::base.html.twig`), который указывает, что шаблон расположен в директории проекта `app/Resources/views`. Этот синтаксис будет рассматриваться в секции *Правила именования и расположения Шаблонов*.

Ключом к наследованию шаблонов является таг `{% extends %}`. Он сообщает движку шаблонизатора, что необходимо сначала выполнить базовый шаблон, который настроит общую разметку и определит некоторое количество блоков. После этого будет отображаться дочерний шаблон, который указывает, что блоки родителя `title` и `body` будут замещаться аналогичными блоками потомка. В зависимости от значение переменной `blog_entries`, результат может быть таким:

```
<!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>
```

Обратите внимание, что дочерний шаблон не определяет блок `sidebar`, поэтому используется значение из родительского шаблона. Контент тага `{% block %}` внутри родительского шаблона всегда будет использоваться по умолчанию.

Вы можете использовать столько уровней наследования, сколько вам требуется. В следующей секции будет разобрано типичное трёхуровневое наследование, а также будет рассказано о том, как шаблоны располагаются внутри Symfony2 проекта.

При работе с наследованием шаблонов есть несколько правил, о которых нужно помнить:

- Если вы используете таг `{% extends %}` в шаблоне, он должен быть первым тагом в этом шаблоне.
- Чем больше тагов `{% block %}` у вас в базовом шаблоне, тем лучше. Запомните, дочерний шаблон не обязан реализовывать все блоки родителя, поэтому создавайте столько блоков в базовом шаблоне, сколько хотите и указывайте для них разумный контент по умолчанию. Чем больше блоков имеет ваш базовый шаблон, тем более гибким будет ваш layout.
- Если вы обнаружите повторяющийся контент в нескольких шаблонах, вероятно это означает, что лучше бы переместить этот контент в `{% block %}` родительского шаблона. В некоторых случаях, более удачным решением будет создание нового шаблона и его подключение (см. *Подключение других шаблонов*).
- Если вам нужен контент блока из родительского шаблона, вы можете использовать функцию `{{ parent() }}`. Это удобно, в случае если вы хотите добавить к контенту родительского блока что-либо, вместо того, чтобы полностью заменять его.

```
{% block sidebar %}
    <h3>Table of Contents</h3>
    ...
    {{ parent() }}
{% endblock %}
```

2.7.3 Правила именования и расположения Шаблонов

По умолчанию, шаблону могут располагаться в двух различных местах:

- `app/Resources/views/`: Директория `views` может содержать шаблоны, общие для всего приложения (например layout приложения), а также шаблоны, которые переопределяют шаблоны пакетов (см. *Переопределение шаблонов пакета*);
- `путь/к/пакету/Resources/views/`: Каждый пакет содержит свои собственные шаблоны в директории `Resources/views` (и её поддиректориях). Большинство шаблонов будет располагаться внутри пакета.

Symfony2 использует синтаксис **bundle:controller:template** для шаблонов. Это позволяет определять место расположения для различных типов шаблонов, каждый из которых располагается в определённом месте:

- `AcmeBlogBundle:Blog:index.html.twig`: Эта форма записи используется для шаблона определённой страницы. Эти три строки, разделённые двоеточием (`:`) означает следующее:
  - `AcmeBlogBundle`: (*пакет*), шаблон расположен внутри пакета `AcmeBlogBundle` (например `src/Acme/BlogBundle`);
  - `Blog`: (*контроллер*), указывает, что шаблон расположен внутри субдиректории `Blog` директории `Resources/views`;
  - `index.html.twig`: (*шаблон*), собственно имя файла - `index.html.twig`.

  При условии что `AcmeBlogBundle` расположен в директории `src/Acme/BlogBundle`, полный путь к файлу шаблона будет следующий: `src/Acme/BlogBundle/Resources/views/Blog/index.html.twig`.
- `AcmeBlogBundle::layout.html.twig`: Эта форма записи сообщает, что это базовый шаблон для пакета `AcmeBlogBundle`. Так как наименование контроллера не указано, шаблон располагается в директории `Resources/views/layout.html.twig` пакета `AcmeBlogBundle`.
- `::base.html.twig`: Эта форма записи ссылается на шаблон или мастер-шаблон (layout) уровня всего приложения. Обратите внимание, что эта строка начинается с двух двоеточий (`::`), что означает следующее: шаблон не принадлежит никакому пакету и расположен он в директории `app/Resources/views/`.

В секции *Переопределение шаблонов пакета* вы узнаете как любой шаблон, находящийся, например, в пакете `AcmeBlogBundle`, может быть переопределён путём размещения шаблона с тем же именем в директории `app/Resources/AcmeBlogBundle/views/`. Это даёт возможность переопределять любые шаблоны любого стороннего пакета.

Совет: Надеемся синтаксис именования шаблонов показался вам знакомым - такой же формат используется и для контроллеров (см. *Шаблон Именования Контроллера*).

Суффиксы Шаблонов

Формат **bundle**:**controller**:**template** имени шаблона указывает *где* шаблон находится. Каждое имя шаблона также имеет два расширения, которые определяют *формат* и *тип шаблонизатора* для этого шаблона.

- **AcmeBlogBundle:Blog:index.html.twig** - HTML формат, шаблонизатор - Twig;
- **AcmeBlogBundle:Blog:index.html.php** - HTML формат, шаблонизатор - PHP;
- **AcmeBlogBundle:Blog:index.css.twig** - CSS формат, шаблонизатор - Twig.

По умолчанию, любой шаблон в Symfony2 может быть написан либо на Twig либо на PHP и последняя часть расширения (`.twig` или `.php`) указывает, какой из этих двух

шаблонизаторов будет использован. Первая часть расширения (`.html`, `.css` и т.д.) - это конечный формат, который шаблон будет генерировать. В отличие от типа шаблонизатора, который определяет как Symfony2 будет анализировать шаблон, указание формата всего лишь способ организации шаблонов, в случае если один ресурс может быть отображен и как HTML (`index.html.twig`), и как XML (`index.xml.twig`) и в любом другом формате, который может потребоваться. Дополнительную информацию ищите в секции *Форматы шаблонов*.

Примечание: Доступные "движки" шаблонизаторов можно настроить и даже добавить новые. Дополнительную информацию ищите в секции о *Настройке шаблонизатора*

2.7.4 Таги и Хелперы

Примечание: Примечание переводчика: здесь и далее функция-"помощник" (`helper`) будет обозначена как **хелпер**.

Вы уже узнали основы создания шаблонов, как они именуются и как работает наследование шаблонов. Самое тяжелое уже позади. В этой секции вы узнаете о множестве инструментов, помогающих выполнять типичные для шаблонов задачи, такие как подключение других шаблонов, создание ссылок на страницы и вставку изображений.

Symfony2 содержит много специализированных тагов и функций Twig, которые упрощают работу дизайнера шаблонов. PHP шаблонизатор предоставляет расширяемую систему *хелперов*, которая предоставляет полезные функции в рамках шаблона.

Мы уже видели несколько встроенных в Twig тагов (`{% block %}` & `{% extends %}`), а также пример PHP-хелпера (`$view['slots']`). Давайте же узнаем и о других.

Подключение других шаблонов

На практике у вас часто будет возникать потребность подключить один и тот же шаблон или же фрагмент кода для многих страниц. Например, в приложении с некоторыми статьями, код шаблона, отображающий одну статью, может быть использован на странице статьи, на странице, отображающей наиболее популярные статьи или же на странице со списком последних статей.

Когда вам необходимо использовать некоторый блок PHP-кода, вы выносите этот код в класс или в функцию. Тоже верно и для шаблонов. Переместив код шаблона, используемый в нескольких местах, в отдельный файл, впоследствии он может быть подключен к любому другому шаблону. Сначала создайте шаблон, который хотите использовать в нескольких местах:

- *Twig*

```
{# src/Acme/ArticleBundle/Resources/views/Article/articleDetails.html.twig #}
<h2>{{ article.title }}</h2>
<h3 class="byline">by {{ article.authorName }}</h3>

<p>
  {{ article.body }}
</p>
```

- *PHP*

```
<!-- src/Acme/ArticleBundle/Resources/views/Article/articleDetails.html.php -->
<h2><?php echo $article->getTitle() ?></h2>
<h3 class="byline">by <?php echo $article->getAuthorName() ?></h3>

<p>
  <?php echo $article->getBody() ?>
</p>
```

Подключить этот шаблон к любому другому несложно:

- *Twig*

```
{# src/Acme/ArticleBundle/Resources/Article/list.html.twig #}
{% extends 'AcmeArticleBundle::layout.html.twig' %}

{% block body %}
    <h1>Recent Articles<h1>

    {% for article in articles %}
        {% include 'AcmeArticleBundle:Article:articleDetails.html.twig' with {'article': a
    {% endfor %}
{% endblock %}
```

- *PHP*

```
<!-- src/Acme/ArticleBundle/Resources/Article/list.html.php -->
<?php $view->extend('AcmeArticleBundle::layout.html.php') ?>

<?php $view['slots']->start('body') ?>
    <h1>Recent Articles</h1>

    <?php foreach ($articles as $article): ?>
        <?php echo $view->render('AcmeArticleBundle:Article:articleDetails.html.php', arra
    <?php endforeach; ?>
<?php $view['slots']->stop() ?>
```

Шаблон подключается при помощи тага `{% include %}`. Обратите внима-

ние, что имя шаблона следует типовым конвенциям об именовании. Шаблон `articleDetails.html.twig` использует переменную `article`. Она передаётся в него из шаблона `list.html.twig` при помощи команды `with`.

Совет: Выражение `{'article': article}` - это стандартный синтаксис для хэшей (ассоциативных массивов) в Twig. Если вам нужно передать много элементов - массив будет выглядеть следующим образом: `{'foo': foo, 'bar': bar}`.

Внедрение контроллеров

В некоторых случаях, вам может потребоваться нечто большее, нежели просто подключение шаблонов. Положим, у вас есть боковая панель в шаблоне, которая содержит три самых последних статьи. Получение этих трёх статей может включать запросы к базе данных или же выполнение некоторых других операций, которые нельзя выполнить непосредственно из шаблона.

Решением в данном случае является встраивание в ваш шаблон результата контроллера целиком. Во-первых, создайте контроллер, который будет отображать некое число последних статей:

```
<?php
// 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'
    }
}
```

Шаблон `recentList` очень прост:

- *Twig*

  ```
  {# src/Acme/ArticleBundle/Resources/views/Article/recentList.html.twig #}
  {% for article in articles %}
    <a href="/article/{{ article.slug }}">
        {{ article.title }}
    </a>
  {% endfor %}
  ```

- *PHP*

```
<!-- src/Acme/ArticleBundle/Resources/views/Article/recentList.html.php -->
<?php foreach ($articles as $article): ?>
    <a href="/article/<?php echo $article->getSlug() ?>">
        <?php echo $article->getTitle() ?>
    </a>
<?php endforeach; ?>
```

Примечание: Обратите внимание, что мы слукавили и захардкодили URL статьи в этом примере (`/article/*slug*`). Это очень плохая практика. В следующей секции вы узнаете, как правильно создавать ссылки на страницы приложения.

Для того чтобы подключить контроллер, вам необходимо сослаться на него, используя стандартный синтаксис логического имени котроллера (**bundle**:**controller**:**action**):

- *Twig*

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

  <div id="sidebar">
      {% render "AcmeArticleBundle:Article:recentArticles" with {'max': 3} %}
  </div>
  ```

- *PHP*

  ```
  <!-- app/Resources/views/base.html.php -->
  ...

  <div id="sidebar">
      <?php echo $view['actions']->render('AcmeArticleBundle:Article:recentArticles', array('
  </div>
  ```

Всякий раз, когда вы понимаете, что вам нужна переменная или данные, к которым вы не можете получить доступ из шаблона, обязательно рассмотрите вариант с встраиванием контроллера. Контроллеры быстро выполняются и способствуют хорошей организации кода и его повторному использованию.

Создание ссылок на страницы

Создание ссылок на другие страницы вашего приложения - это одна из типичных операций в шаблоне. Вместо того, чтобы хардкодить URL в шаблоне, используйте Twig-функцию `path` (в PHP - хелпер `router`) для создания URL, основанных на конфигурации маршрутизатора. Потом, если вы захотите изменить URL некоторой страницы, вам всего лишь потребуется изменить конфигурацию маршрутизатора. Шаблоны автоматически сгенерируют новый URL.

Сначала создадим ссылку на страницу "_welcome", которая определяется следующей конфигураций маршрутизатора:

- *YAML*

```
_welcome:
    pattern:  /
    defaults: { _controller: AcmeDemoBundle:Welcome:index }
```

- *XML*

```
<route id="_welcome" pattern="/">
    <default key="_controller">AcmeDemoBundle:Welcome:index</default>
</route>
```

- *PHP*

```
<?php
$collection = new RouteCollection();
$collection->add('_welcome', new Route('/', array(
    '_controller' => 'AcmeDemoBundle:Welcome:index',
)));

return $collection;
```

Для создания ссылки на страницу используйте функцию path и маршрут:

- *Twig*

```
<a href="{{ path('_welcome') }}">Home</a>
```

- *PHP*

```
<a href="<?php echo $view['router']->generate('_welcome') ?>">Home</a>
```

Как и ожидалось, она сгенерирует URL /. Давайте теперь посмотрим, как это работает для более сложных маршрутов:

- *YAML*

```
article_show:
    pattern:  /article/{slug}
    defaults: { _controller: AcmeArticleBundle:Article:show }
```

- *XML*

```
<route id="article_show" pattern="/article/{slug}">
    <default key="_controller">AcmeArticleBundle:Article:show</default>
</route>
```

- *PHP*

```
<?php
$collection = new RouteCollection();
$collection->add('article_show', new Route('/article/{slug}', array(
    '_controller' => 'AcmeArticleBundle:Article:show',
)));

return $collection;
```

В этом случае, вам нужно указать и имя маршрута (`article_show`) и значение параметра `{slug}`. Используя этот маршрут, давайте вернёмся к шаблону `recentList` из предыдущей секции и создадим ссылку на статью правильно:

- *Twig*

```
{# src/Acme/ArticleBundle/Resources/views/Article/recentList.html.twig #}
{% for article in articles %}
  <a href="{{ path('article_show', { 'slug': article.slug }) }}">
      {{ article.title }}
  </a>
{% endfor %}
```

- *PHP*

```
<!-- src/Acme/ArticleBundle/Resources/views/Article/recentList.html.php -->
<?php foreach ($articles in $article): ?>
    <a href="<?php echo $view['router']->generate('article_show', array('slug' => $article
        <?php echo $article->getTitle() ?>
    </a>
<?php endforeach; ?>
```

Совет: Для генерации абсолютных URL необходимо использовать Twig-функцию `url`:

```
<a href="{{ url('_welcome') }}">Home</a>
```

В PHP-шаблонах для этого нужно передать третий аргумент в метод `generate()`:

```
<a href="<?php echo $view['router']->generate('_welcome', array(), true) ?>">Home</a>
```

Ссылки на ресурсы (assets)

Шаблоны также часто ссылаются на картинки, скрипты, страницы стилей и прочие ресурсы (здесь и далее вместо asset будет использован термин *ресурс*). Конечно, вы можете хардкодить пути к ресурсам (например так `/images/logo.png`), но Symfony2 предлагает использовать более гибкую Twig-функцию `asset`:

- *Twig*

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

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

- *PHP*

```
<img src="<?php echo $view['assets']->getUrl('images/logo.png') ?>" alt="Symfony!" />

<link href="<?php echo $view['assets']->getUrl('css/blog.css') ?>" rel="stylesheet" type="
```

Главная задача функции `asset` - сделать ваше приложение по возможности более переносимым. Если приложение находится в корне домена (http://example.com), тогда будет отображен путь `/images/logo.png`. Но, если ваше приложение находится в поддиректории (http://example.com/my_app), путь к ресурсам будет учитывать эту поддиректорию (`/my_app/images/logo.png`). Функция `asset` берёт на себя заботу по определению того, как именно ваше приложение используется и генерирует соответствующие пути.

В дополнение к этому, если использовать функцию `asset`, Symfony сможет автоматически добавлять строку запроса к ресурсам, чтобы исключить их кеширование при выгрузке на сервер. Например, `/images/logo.png` может выглядеть так: `/images/logo.png?v2`. Подробнее об этом смотрите тут: *ref-framework-assets-version*.

2.7.5 Подключение CSS и Javascript файлов в Twig

Ни один современный сайт не может обойтись без подключения CSS и Javascript файлов. В Symfony, подключение этих ресурсов элегантно обрабатывается, опираясь на возможности наследования шаблонов.

Совет: В этой секции вы узнаете философию подключения CSS и Javascript файлов в Symfony. Symfony также содержит библиотеку Assetic, которая следует той же философии, но позволяет вам также выполнять много интересных операций над этими ресурсами. Дополнительную информацию можно получить в книге рецептов: `/cookbook/assetic/asset_management`.

Давайте начнём с добавления двух блоков к базовому шаблону, который будет подключать ваши ресурсы: один назовём `stylesheets`, располагаться он будет внутри HTML-тага `head`, другой назовём `javascripts` и размещаться он будет перед закрывающим HTML-тагом `body`. Эти блоки будут содержать все стили и скрипты, которые вам требуются для сайта:

```
{# '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>
```

Проще некуда! Но что, если вам потребуется включить дополнительный файл стилей или скрипт в дочернем шаблоне? Например, положим у вас есть страница контактов и вам нужно подключить файл стилей `contact.css` *лишь на одной этой странице*. Внутри шаблона страницы contact необходимо выполнить следующее:

```
{# 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 %}

{# ... #}
```

В дочернем шаблоне вы просто переопределяете блок `stylesheets` и размещаете новый стиль внутри этого блока. Конечно же, поскольку вам нужно добавить стиль, а не изменить его, вы должны использовать функцию `parent()` для того, чтобы получить все стили из родительского шаблона.

Вы также можете включать ресурсы, расположенные в директории `Resources/public` ваших пакетов. Вам нужно будет выполнить команду `php app/console assets:install target [--symlink]`, которая скопирует (или создаст символическую ссылку) файлы в нужное место (target по умолчанию имеет значение “web”.

```
<link href="{{ asset('bundles/acmedemo/css/contact.css') }}" type="text/css" rel="stylesheet" /
```

Конечным результатом является страница, которая включает как `main.css`, так и `contact.css`

2.7.6 Настройка и использование сервиса шаблонизатора

Сердцем системы шаблонов Symfony2 является её "движок" (Engine). Это специализированный объект, который отвечает за отображение шаблонов и возврат их контента. Например, когда вы отображаете шаблон из контроллера, вы используете сервис шаблонизатора:

```
<?php
return $this->render('AcmeArticleBundle:Article:index.html.twig');
```

Этот код эквивалентен следующему:

```
<?php
$engine = $this->container->get('templating');
$content = $engine->render('AcmeArticleBundle:Article:index.html.twig');

return $response = new Response($content);
```

Сервис шаблонизатора предварительно настроен для автоматической работы внутри Symfony2. Естественно он может быть настроен через файл с настройками приложения:

- *YAML*

  ```
  # app/config/config.yml
  framework:
      # ...
      templating: { engines: ['twig'] }
  ```

- *XML*

  ```
  <!-- app/config/config.xml -->
  <framework:templating>
      <framework:engine id="twig" />
  </framework:templating>
  ```

- *PHP*

  ```
  <?php
  // app/config/config.php
  $container->loadFromExtension('framework', array(
      // ...
      'templating'      => array(
          'engines' => array('twig'),
      ),
  ));
  ```

Для настройки доступно много разных опций, которые описаны в Приложении о Конфигурации.

Примечание: `Twig` необходим для использования веб-профайлера (а также многих пакетов от сторонних разработчиков).

2.7.7 Переопределение шаблонов пакета

Сообщество Symfony2 гордится собой за то, что его энтузиастами создано и поддерживается много различных качественных пакетов (см. Symfony2Bundles.org) на любой случай жизни. Если вы используете сторонние пакеты, вам может потребоваться изменять их шаблоны.

Предположим, вы подключили воображаемый пакет с открытым исходным кодом `AcmeBlogBundle`. Вы, в общем-то, всем довольны, но вам хотелось бы заменить страницу "list" для блога, чтобы настроить её отображение под ваше приложение. Если вы залезете в контроллер `Blog` пакета `AcmeBlogBundle`, вы найдёте следующий код:

```
<?php
public function indexAction()
{
    $blogs = // получение записей в блоге

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

Когда отображается шаблон `AcmeBlogBundle:Blog:index.html.twig` Symfony2 на самом деле ищет его в двух местах:

1. `app/Resources/AcmeBlogBundle/views/Blog/index.html.twig`
2. `src/Acme/BlogBundle/Resources/views/Blog/index.html.twig`

Для того чтобы переопределить шаблон из пакета, просто скопируйте его в директорию `app/Resources/AcmeBlogBundle/views/Blog/index.html.twig` (директорию `app/Resources/AcmeBlogBundle` нужно создать, так как по умолчанию её там не будет). Теперь вы можете настраивать шаблон по вашему усмотрению.

Эта логика также применима к базовому шаблону пакета. Положим что каждый шаблон в пакете `AcmeBlogBundle` наследуется от базового шаблона `AcmeBlogBundle::layout.html.twig`. Как и ранее, Symfony2 будет искать

этот шаблон в двух местах:

1. `app/Resources/AcmeBlogBundle/views/layout.html.twig`
2. `src/Acme/BlogBundle/Resources/views/layout.html.twig`

Как и ранее, для переопределения шаблона, просто скопируйте его из пакета `app/Resources/AcmeBlogBundle/views/layout.html.twig`. После этого вы вольны править его по своему усмотрению.

Если вы сделаете шаг назад, вы увидите, что Symfony2 всегда начинает искать файл шаблона в директории `app/Resources/{BUNDLE_NAME}/views/`. Если там его нет, поиск продолжается в директории `Resources/views` пакета. Это значит, что все шаблоны любого пакета могут быть переопределены в директории пакета внутри `app/Resources`.

Переопределение шаблонов ядра

Так как Symfony2 это тоже пакет, его шаблоны также можно переопределить тем же образом. Например, `TwigBundle` содержит несколько шаблонов для различных исключительных ситуаций и ошибок, которые могут быть переопределены, если их скопировать из директории `Resources/views/Exception` пакета `TwigBundle` в директорию `app/Resources/TwigBundle/views/Exception`.

2.7.8 Трёхуровневое наследование

Один из способов использовать наследование - трёхуровневый подход. Этот метод замечательно работает с тремя различными типами шаблонов, которые мы уже рассмотрели:

- Создайте файл `app/Resources/views/base.html.twig`, который содержит базовую разметку приложения (как в предыдущем примере). Внутри приложения такой шаблон называется `::base.html.twig`;
- Создайте файл для каждой секции сайта. Например `AcmeBlogBundle` будет содержать шаблон `AcmeBlogBundle::layout.html.twig`, который включает только элементы специфичные для блога;

```
{# src/Acme/BlogBundle/Resources/views/layout.html.twig #}
{% extends '::base.html.twig' %}

{% block body %}
    <h1>Blog Application</h1>

    {% block content %}{% endblock %}
{% endblock %}
```

- Создайте шаблоны для каждой страницы и унаследуйте из от шаблона соответствующей секции (пакета). Например, страница “index” будет вызывать что-то типа `AcmeBlogBundle:Blog:index.html.twig` и отображать записи блога:

```
{# 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 %}
```

Обратите внимание, что этот шаблон наследуется от шаблона секции `AcmeBlogBundle::layout.html.twig`, который, в свою очередь, наследуется от базового шаблона приложения (`::base.html.twig`). Это и есть типичное трёхуровневое наследование.

При создании приложения вы можете выбрать - будете ли вы следовать этому методу иди же каждый шаблон будет наследоваться напрямую от базового шаблона приложения (`{% extends '::base.html.twig' %}`). Трёхуровневая модель является проверенным и хорошо зарекомендовавшим себя методом в сторонних пакетах, так как базовый шаблон пакета может быть легко переопределён для того чтобы использовать шаблон вашего приложения.

2.7.9 Экранирование

При создании HTML из шаблона, всегда есть риск, что переменная шаблона будет содержать HTML-код или опасный клиентский скрипт. В результате этот контент может сломать HTML разметку страницы или же позволить злоумышленнику выполнить Cross Site Scripting (XSS) атаку. Вот классический пример этого:

- *Twig*

  ```
  Hello {{ name }}
  ```

- *PHP*

  ```
  Hello <?php echo $name ?>
  ```

Представьте, что пользователь ввёл следующий код в качестве имени:

```
<script>alert('hello!')</script>
```

Без экранирования, шаблон отобразит:

```
Hello <script>alert('hello!')</script>
```

а клиент (браузер) выполнит JavaScript код и отобразит окошко alert.

Этот пример выглядит безобидным, но этот же пользователь может также написать скрипт, который выполнит вредоносные действия в защищённой зоне приложения как будто бы у него были права на это.

Ответом на данную проблему является экранирование (output escaping). При наличии экранирования, тот же код будет отображен совершенно безобидно:

```
Hello &lt;script&gt;alert(&#39;helloe&#39;)&lt;/script&gt;
```

Twig и PHP шаблонизаторы решают эту проблему различным образом. Если вы используете Twig, экранирование включено по умолчанию и ваши шаблоны защищены. В PHP экранирование не автоматическое и подразумевает что вы вручную будете экранировать данные при необходимости.

Экранирование в Twig

Если вы используете шаблоны Twig, экранирование включено по умолчанию. Это означает, что ваш код защищён от неожиданных действий пользователей "из коробки". По умолчанию, экранирование подразумевает, что контент будет экранирован для HTML.

В некоторых случаях вам может потребоваться отключить экранирование, когда вы отображаете переменную, которой доверяете и которая содержит HTML-разметку, которую не нужно экранировать. Положим, что администратор имеет возможность писать статьи, которые содержат HTML-код. По умолчанию Twig будет экранировать тело статьи. Для того чтобы отобразить его обычным образом необходимо добавить фильтр `raw`: `{{ article.body | raw }}`.

Вы также можете отключить экранирование внутри блока или же для шаблона целиком. Подробнее это описано в документации Twig: Output Escaping.

Экранирование в PHP

Экранирование в PHP шаблонах не автоматическое. Это означает, что если вы не экранировали переменную - вы не защищены. Для экранирования необходимо использовать специальный метод `escape()`:

```
Hello <?php echo $view->escape($name) ?>
```

По умолчанию, метод `escape()` полагает, что переменная отображается в HTML контексте (и соответственно переменная экранируется чтобы быть безопасной в HTML). Второй аргумент позволяет вам изменить контекст. Например, чтобы вывести что-либо в JavaScript используйте контекст `js`:

```
var myMsg = 'Hello <?php echo $view->escape($name, 'js') ?>';
```

2.7.10 Форматы шаблонов

Шаблоны - это основной способ для отображения контента в *любом* формате. В большинстве случаев вы будете отображать HTML контент, но шаблон также может быть использован для генерации JavaScript, CSS, XML или же любого другого формата на ваш выбор.

Например, один и тот же ресурс может отображаться в разных форматах. Для отображения индексной страницы статей в XML формате, просто добавьте формат в имя шаблона:

- *XML template name*: `AcmeArticleBundle:Article:index.xml.twig`
- *XML template filename*: `index.xml.twig`

По сути это всего лишь соглашение по именованию шаблонов - шаблоны разных форматов не будут отображаться разными способами.

Во многих случаях вам может потребоваться один и тот же контроллер отобразить в нескольких форматах в зависимости от формата запроса. Вы можете выполнить это следующим образом:

```
<?php
public function indexAction()
{
    $format = $this->getRequest()->getRequestFormat();

    return $this->render('AcmeBlogBundle:Blog:index.'.$format.'.twig');
}
```

Метод `getRequestFormat` объекта `Request` по умолчанию возвращает `html`, но может также возвращать любой формат запрошенный пользователем. Формат запроса часто управляется маршрутизатором, где маршрут может быть настроен таким образом чтобы URL `/contact` возвращал HTML а `/contact.xml` устанавливал бы формат запроса XML и контроллер будет возвращать XML. Более подробно этот вопрос рассматривается в главе о Маршрутизации - *Продвинутая маршрутизация*.

Для создания ссылок, которые используют параметр формата - добавьте ключ `_format` к хешу параметров:

- *Twig*

  ```
  <a href="{{ path('article_show', {'id': 123, '_format': 'pdf'}) }}">
      PDF Version
  </a>
  ```

- *PHP*

  ```
  <a href="<?php echo $view['router']->generate('article_show', array('id' => 123, '_format'
      PDF Version
  </a>
  ```

2.7.11 Заключение

Шаблонизатор Symfony - это мощный инструмент, который может быть использован каждый раз, когда вам необходимо сгенерировать контент в HTML, XML или любом

другом формате. И, не смотря на то, что шаблоны - это обычный способ генерации контента в контроллере, он не обязателен. Объект `Response`, возвращаемый контроллером может быть создан как с использованием шаблонизатора, так и без него:

```
<?php
// 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');
```

Шаблонизатор Symfony - гибок и позволяет по умолчанию использовать два различных "визуализатора": традиционные *PHP* шаблоны и мощные *Twig* шаблоны. Оба этих типа шаблонов поддерживают иерархию и укомплектованы богатым набором функций-помощников, предназначенных для выполнения повседневных задач.

В целом, шаблоны следует рассматривать как мощный инструмент в вашем распоряжении. В некоторых случаях вам может и не потребоваться отображать шаблон - и в Symfony2 это тоже совершенно нормальная ситуация.

2.7.12 Читайте в книге рецептов

- *Как использовать PHP шаблоны вмесро Twig*
- *Как создать собственные страницы ошибок*

2.8 Базы данных и Doctrine ("Модель")

Давайте посмотрим правде в глаза, одни из самых распространённых и сложных задач для любого приложения включают хранение и чтение информации из базы данных. К счастью, Symfony поставляется совмещённым с Doctrine - библиотекой, главная цель которой дать мощный инструмент, позволяющий делать это просто. В этой главе вы постигнете основу философии Doctrine и увидите насколько простой может быть работа с базой данных.

Примечание: Doctrine полностью отделёна от Symfony и её использование необязательно. Эта глава о Doctrine ORM, цель которой позволить представить объекты в реляционных базах данных (таких как *MySQL*, *PostgreSQL* или *Microsoft SQL*). Если вы предпочитаете пользоваться необработанными запросами, то это просто и раскрыто в статье "`/cookbook/doctrine/dbal`" среди рецептов.

Также можно хранить данные в MongoDB используя библиотеку Doctrine ODM. За дополнительной информацией обратитесь к статье "`/bundles/DoctrineMongoDBBundle/index`" из документации.

2.8.1 Простой пример: Product

Простейший путь для понимания Doctrine - это увидеть её в действии. В этом разделе вы настроите базу данных, создадите объект `Product` (`Продукт`), поместите его туда и получите обратно.

Код вместе с примером

Если хотите придерживаться примера из этой главы создайте `AcmeStoreBundle`:

```
php app/console generate:bundle --namespace=Acme/StoreBundle
```

Конфигурация базы данных

Перед тем как действительно начать, необходимо настроить соединение с базой данных. По соглашению эта информация обычно указывается в файле `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
```

Примечание: Указание параметров в `parameters.yml` всего лишь соглашение. На них ссылается основной файл конфигурации, когда настраивается Doctrine:

```
doctrine:
    dbal:
        driver:   %database_driver%
        host:     %database_host%
        dbname:   %database_name%
        user:     %database_user%
        password: %database_password%
```

Разделяя информацию о базе данных по отдельным файлам, можно легко хранить различные версии этих файлов на каждом сервере. Также легко можно хранить конфигурацию базы данных (или любую важную информацию) вне проекта, например внутри конфигурации Apache. Дополнительная информация здесь `/cookbook/configuration/external_parameters`.

Теперь, когда Doctrine знает о базе данных, вы хотите чтобы она создала базу данных для вас:

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

Создание сущностного класса

Предположим, создаётся приложение, в котором необходимо показывать продукты. Даже не задумываясь о Doctrine или базах данных, понятно что необходим объект `Product` чтобы представить эти продукты. Создайте его внутри папки `Entity` (`Сущность`) в `AcmeStoreBundle`:

```
// src/Acme/StoreBundle/Entity/Product.php
namespace Acme\StoreBundle\Entity;

class Product
{
    protected $name;

    protected $price;

    protected $description;
}
```

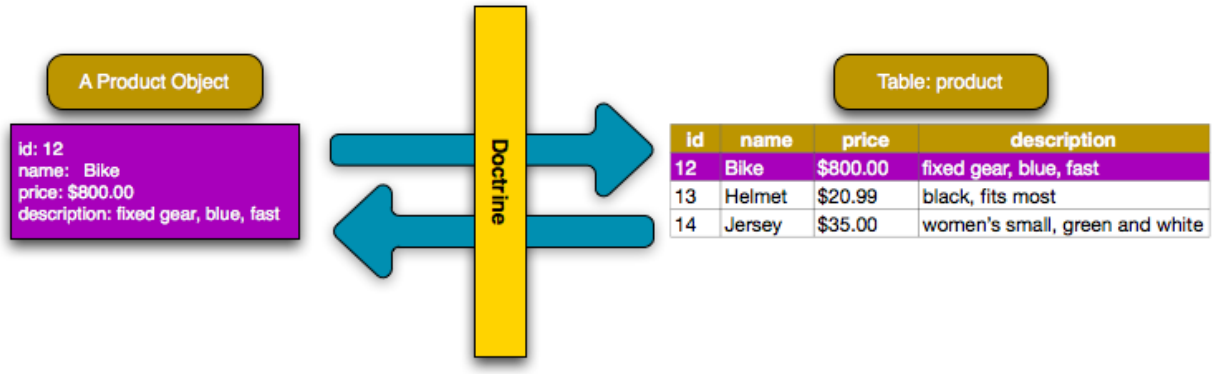
Этот класс - часто называемый “сущность”, что значит *базовый класс, содержащий данные* - простой и помогает выполнять бизнес требования к необходимым продуктам в приложении. Он пока не может хранится в базе данных - он всего лишь простой PHP класс.

Совет: Однажды, когда вы изучите Doctrine, то сможете поручить ей создать этот класс-сущность:

```
php app/console doctrine:generate:entity --entity="AcmeStoreBundle:Product" --fields="name:strin
```

Добавление информации об отображении

Doctrine позволяет работать с базами данных гораздо более интересным способом чем простое получение строк в массив из таблицы, основанной на колонках. Вместо него, Doctrine хранить *объекты* целиком в базе данных и получать целые объекты из неё. Это возможно благодаря отображению PHP класса в таблицу для базы данных и свойств этого PHP класса в колонки этой таблицы:



Чтобы Doctrine могла сделать это, надо просто создать "метаданные" или конфигурацию, которые в точности расскажут ей как класс `Product` и его свойства должны быть *отображены* в базу данных. Эти метаданные могут быть указаны в большом количестве форматов, включая YAML, XML или прямо внутри класса `Product` через аннотации:

Примечание: Bundle может принимать только один формат определения метаданных. Например, нельзя смешивать YAML определения метаданных и определения через аннотациии в классе-сущности PHP.

- *Annotations*

```
// src/Acme/StoreBundle/Entity/Product.php
namespace Acme\StoreBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity
 * @ORM\Table(name="product")
 */
class Product
{
    /**
     * @ORM\Id
     * @ORM\Column(type="integer")
     * @ORM\GeneratedValue(strategy="AUTO")
     */
    protected $id;

    /**
     * @ORM\Column(type="string", length=100)
     */
    protected $name;
```

```
    /**
     * @ORM\Column(type="decimal", scale=2)
     */
    protected $price;

    /**
     * @ORM\Column(type="text")
     */
    protected $description;
}
```

- *YAML*

```
# src/Acme/StoreBundle/Resources/config/doctrine/Product.orm.yml
Acme\StoreBundle\Entity\Product:
    type: entity
    table: product
    id:
        id:
            type: integer
            generator: { strategy: AUTO }
    fields:
        name:
            type: string
            length: 100
        price:
            type: decimal
            scale: 2
        description:
            type: text
```

- *XML*

```
<!-- src/Acme/StoreBundle/Resources/config/doctrine/Product.orm.xml -->
<doctrine-mapping xmlns="http://doctrine-project.org/schemas/orm/doctrine-mapping"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://doctrine-project.org/schemas/orm/doctrine-mapping
                  http://doctrine-project.org/schemas/orm/doctrine-mapping.xsd">

    <entity name="Acme\StoreBundle\Entity\Product" table="product">
        <id name="id" type="integer" column="id">
            <generator strategy="AUTO" />
        </id>
        <field name="name" column="name" type="string" length="100" />
        <field name="price" column="price" type="decimal" scale="2" />
        <field name="description" column="description" type="text" />
    </entity>
```

```
    </doctrine-mapping>
```

Совет: Имя таблицы необязательно и если опущено, то оно будет определено автоматически, исходя из названия класса-сущности.

Doctrine позволяет выбирать из широкого разнообразия различных типов полей, каждый из которых со своими настройками. За информацией о доступных типах обращайтесь к разделу *Справка по типам полей в Doctrine*.

См.также:

Также можно обратиться к Doctrine-овой Basic Mapping Documentation за детальной информацией об отображении. Если будете использовать аннотации, необходимо предварять их, используя `ORM\` (например, `ORM\Column(..)`), об этом не говорится в документации Doctrine. Также надо будет включать `use Doctrine\ORM\Mapping as ORM;` утверждение, которое *импортирует* `ORM` префикс для аннотаций.

Осторожно: Будьте осторожны имена классов и свойств не отображаются в защищённые ключевые слова SQL (такие как `group` или `user`). Например, если имя сущностного класса `Group`, тогда, по умолчанию, таблица будет названа `group`, что вызовет ошибку SQL в некоторых движках. Обратитесь к документации по зарезервированным ключевым словам SQL чтобы узнать как лучше экранировать такие имена.

Примечание: Когда используется другая библиотека или программа (например, Doxygen), использующая аннотации, необходимо поместить в класс аннотацию `@IgnoreAnnotation`, чтобы указать какие из них Symfony должен игнорировать.

Например, чтобы уберечь `@fn` аннотацию от выдачи исключения, добавьте следующее:

```
/**
 * @IgnoreAnnotation("fn")
 *
 */
class Product
```

Создание геттеров и сеттеров

Теперь, когда Doctrine знает как сохранить объект `Product` в базу данных, сам класс пока ещё бесполезен. Так как `Product` всего лишь обычный PHP класс, необходимо создать геттер и сеттер методы (например, `getName()`, `setName()`) чтобы получить доступ к его свойствам (т. к. свойства являются `protected`). К счастью, Doctrine может сделать это по команде:

```
php app/console doctrine:generate:entities Acme/StoreBundle/Entity/Product
```

Эта команда удостоверяется что все геттеры и сеттеры созданы для класса `Product`. Она безопасна - можно запускать её снова и снова: команда лишь создаёт геттеры и сеттеры, которых ещё нет (т. о. она не изменит существующие методы).

Осторожно: Команда `doctrine:generate:entities` сохраняет резервную копию исходного файла `Product.php` в `Product.php~`. В некоторых случаях, присутствие этого файла может вызвать ошибку "Cannot redeclare class". Он может быть безопасно удалён.

Также можно создать все известные сущности (например, любой PHP класс с информацией для отображения Doctrine) для бандла или целого пространства имён:

```
php app/console doctrine:generate:entities AcmeStoreBundle
php app/console doctrine:generate:entities Acme
```

Примечание: Doctrine не интересует являются ли свойства `protected` или `private`, или имеются либо нет функции геттеров или сеттеров для свойства. Геттеры и сеттеры создаются здесь только потому что они понадобятся для взаимодействия с PHP объектом.

Создание таблиц/схемы для базы данных

Теперь есть удобный класс `Product` с информацией для отображения, который Doctrine точно знает как сохранить. Конечно, пока нет соотвествующей таблицы `product` в базе данных. К счастью, Doctrine может автоматически создать все таблицы базы данных, необходимые для всех известных сущностей приложения. Чтобы создать их, выполните:

```
php app/console doctrine:schema:update --force
```

Совет: Эта команда необычайно мощная. Она сравнивает как *должна* выглядеть база данных (основываясь на информации об отображении для сущностей) с тем, как она выглядит *на самом деле*, и создаёт SQL выражения, необходимые для *обновления* базы данных до того вида, какой она должна быть. Другими словами, добавив новое свойство с метаданными отображения в `Product` и запустив её снова, она создаст выражение "alter table", необходимое для добавления этого нового столбца к существующей таблице `product`.

Лучший способ получить преимущества от её функциональности это `миграции`, которые позволяют создавать эти SQL выражения и хранить их в миграционных классах, которые могут систематически запускаться на продакшн сервере чтобы соотвествовать схеме базы данных и изменять её безопасно и надёжно.

Теперь база данных имеет полноценную таблицу `product` со столбцами, соотвествующими указанным метаданным.

Сохранение объектов в базе данных

Теперь, когда есть отображённая сущность `Product` и соотвествующая таблица `product`, всё готово к сохранению данных в базу. Внутри контроллера это очень просто. Добавьте следующий метод в `DefaultController` бандла:

```
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()->getEntityManager();
14     $em->persist($product);
15     $em->flush();
16
17     return new Response('Created product id '.$product->getId());
18 }
```

Примечание: Если вы следуете этому примеру, необходимо создать маршрут, указывающий на это действие, чтобы увидеть его в работе.

Пройдёмся по примеру:

- **строки 8-11** В этой части, берётся экземпляр объекта `$product` и с ним проводится работа как с любым другим нормальным PHP объектом;
- **строка 13** Эта строка получает Doctrine-овый объект *entity manager*, отвественный за управление процессами сохранения и получения объектов из базы данных;
- **строка 14** Метод `persist()` сообщает Doctrine команду на "управление" объектом `$product`. Она не вызывает создание запроса к базе данных (пока).
- **строка 15** Когда вызывается метод `flush()`, Doctrine просматривает все объекты, которыми она управляет, чтобы узнать, надо ли сохранить их в базу данных.

В этом примере объект `$product` ещё не был сохранён, поэтому entity manager выполнит запрос `INSERT` и будет создана строка в таблице `product`.

Примечание: Фактически, т. к. Doctrine знает обо всех управляемых сущностях, когда вызывается метод `flush()`, она прощитывает общий набор изменений и выполняет наиболее эффективный и возможный запрос или запросы. Например, если сохраняется 100 объектов `Product` и впоследствии вызывается `flush()`, то Doctrine создаст *единственное* подготовленное выражение и повторно использует его для каждой вставки. Этот паттерн называется *Unit of Work* и используется потомучто быстр и эффективен.

При создании или обновлении объектов рабочий процесс всегда одинаков. В следующем разделе вы увидите что Doctrine достаточно умна чтобы автоматически выдать запрос `UPDATE` если запись уже существует в базе данных.

Совет: Doctrine предлагает библиотеку, позволяющую программно загружать тестовые данные в проект (т. н. "fixture data"). Информацию можно узнать в `/bundles/DoctrineFixturesBundle/index`.

Получение объектов из базы данных

Получение объекта назад из базы данных ещё проще. Например, представим что настроен маршрут, отображающий определённый `Product`, основываясь на его значении `id`:

```
public function showAction($id)
{
    $product = $this->getDoctrine()
        ->getRepository('AcmeStoreBundle:Product')
        ->find($id);

    if (!$product) {
        throw $this->createNotFoundException('No product found for id '.$id);
    }

    // делает что-нибудь, например передаёт объект $product в шаблон
}
```

Когда запрашивается объект определённого типа, всегда используется так называемый "репозиторий". Можно представить репозиторий как PHP класс, чья работа состоит в предоставлении помощи в получении сущностей определённого класса. Можно получить доступ к объекту-репозиторию для класса-сущности через:

```
$repository = $this->getDoctrine()
    ->getRepository('AcmeStoreBundle:Product');
```

Примечание: Строка `AcmeStoreBundle:Product` - это сокращение, которое можно использовать в Doctrine вместо полного имени класса для сущности (например, `Acme\StoreBundle\Entity\Product`). Оно будет работать пока сущность находится в простанстве имён `Entity` вашего бандла.

Когда имеется репозиторий, у вас есть доступ ко всем видам полезных методов:

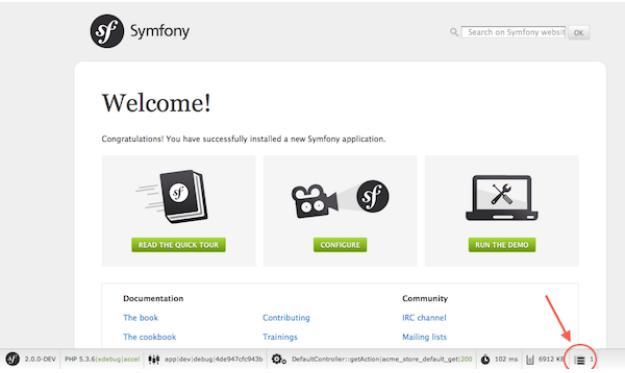
```
// запрос по первичному ключу (обычно "id")
$product = $repository->find($id);

// динамические имена методов, использующиеся для поиска по значению столбцов
$product = $repository->findOneById($id);
$product = $repository->findOneByName('foo');

// ищет *все* продукты
$products = $repository->findAll();

// ищет группу продуктов, основываясь на произвольном значении столбца
$products = $repository->findByPrice(19.99);
```

Примечание: Конечно, также можно задавать сложные запросы, о которых вы узнаете больше в разделе *Запрашивание объектов*.

Также можно использовать преимущества полезных методов `findBy` и `findOneBy` для лёгкого извлечения объектов, основываясь на многочисленных условиях:

```
// запрос одного продукта, подходящего по заданным имени и цене
$product = $repository->findOneBy(array('name' => 'foo', 'price' => 19.99));

// запрос всех продуктов, подходящих по имени и отсортированных по цене
$product = $repository->findBy(
    array('name' => 'foo'),
    array('price' => 'ASC')
);
```

Совет: Когда выдаётся любая страница, можно увидеть сколько запросов было сделано в нижнем правом углу на панели инструментов web debug.



Если кликнуть на иконке, откроется профилировщик, показывающий точные запросы, которые были сделаны.

Обновление объекта

Когда вы получили объект из Doctrine, обновить его также просто. Предположим, есть маршрут, связывающий id продукта с действием обновления в контроллере:

```
public function updateAction($id)
{
    $em = $this->getDoctrine()->getEntityManager();
    $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'));
}
```

Обновление объекта включает три шага:

1. получение объкта из Doctrine;
2. изменение объекта;
3. вызов `flush()` из entity manager

Заметьте, что в вызове `$em->persist($product)` нет необходимости. Вспомните, что этот метод лишь сообщает Doctrine что нужно управлять или "наблюдать" за объектом `$product`. В данной же ситуации, т. к. объект `$product` получен из Doctrine, он уже является управляемым.

Удаление объекта

Удаление объекта очень похоже, но требует вызова метода `remove()` из entity manager:

```
$em->remove($product);
$em->flush();
```

Как и ожидалось, метод `remove()` уведомляет Doctrine о том, что вам хочется удалить указанную сущность из базы данных. Тем не менее, фактический запрос `DELETE` не вызывается до тех пор, пока метод `flush()` не запущен.

2.8.2 Запрашивание объектов

Вы уже видели как объект-репозиторий позволяет выполнять простые запросы без какой-либо работы:

```
$repository->find($id);

$repository->findOneByName('Foo');
```

Конечно, Doctrine также позволяет писать более сложные запросы, используя Doctrine Query Language (DQL). DQL похож на SQL за исключением того, что следует представить что запрашиваются один или несколько объектов из класса-сущности (например, `Product`) вместо строк из таблицы (например, `product`).

Запрашивать из Doctrine можно двумя способами: написанием чистых Doctrine запросов либо использованием Doctrine-ового Query Builder.

Запрашивание объектов через DQL

Представьте что нужно запросить продукты, но вернуть только те, чья цена больше чем `19.99` и по порядку от дешёвого до самого дорогого. Внутри контроллера сделайте следующее:

```
$em = $this->getDoctrine()->getEntityManager();
$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();
```

Если вам удобно с SQL, то DQL должен быть также понятен. Наибольшее различие в том, что надо думать терминами "объектов", а не строк в базе данных. По этой причине, вы выбираете *из* `AcmeStoreBundle:Product` и присваиваете ему псевдоним `p`.

Метод `getResult()` возвращает массив результатов. Если же нужен лишь один объект можно воспользоваться методом `getSingleResult()`:

```
$product = $query->getSingleResult();
```

Осторожно: Метод `getSingleResult()` выбрасывает исключение `Doctrine\ORM\NoResultException` если нет результатов и `Doctrine\ORM\NonUniqueResultException` если возвращается *больше* одного результата. Если используется этот метод, возможно придётся обернуть его в try-catch блок и убедиться в том, что возвращается только один результат (если запрашивается что-то, что может вероятно вернуть более одного результата):

```
$query = $em->createQuery('SELECT ....')
    ->setMaxResults(1);

try {
    $product = $query->getSingleResult();
} catch (\Doctrine\Orm\NoResultException $e) {
    $product = null;
}
// ...
```

Синтаксис DQL невероятно мощный, позволяет легко устанавливать объединения между сущностями (тема *отношений* будет раскрыта позже), группами и т. д. Дополнительная информация в документации Doctrine Doctrine Query Language.

Настройка параметров

Заметка о методе `setParameter()`. Работая с Doctrine, хорошим тоном является указание любых внешних значений через "placeholders", что и было сделанов приведённом выше примере:

```
... WHERE p.price > :price ...
```

Позже можно указать значение `price` placeholder через метод `setParameter()`:

```
->setParameter('price', '19.99')
```

Использование параметров вместо установки значений непосредственно в строку запроса предотвращает атаки через SQL инъекции и должно использоваться *всегда*. При использовании нескольких параметров, можно указать их за один раз воспользовавшись методом `setParameters()`:

```
->setParameters(array(
    'price' => '19.99',
    'name'  => 'Foo',
))
```

Использование Doctrine's Query Builder (Конструктор запросов Doctrine)

Вместо непосредственного написания запросов, можно также использовать Doctrine `QueryBuilder` чтобы сделать ту же работу используя симпатичный, объект-ориентированный интерфейс. Если используется IDE, то можно также получить преимущество от авто-подстановки когда будут вводиться имена методов. Внутри контроллера:

```
$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();
```

Объект `QueryBuilder` содержит все необходимые методы для создания запроса. Вызвав метод `getQuery()`, конструктор запросов вернёт нормальный объект `Query`, являющийся таким же объектом, какой создавался в предыдущем разделе.

За дополнительной информацией о Doctrine's Query Builder, обращайтесь к документации Query Builder.

Custom Repository Classes

В предыдущих разделах вы начали создавать и использовать более сложные запросы изнутри контроллера. Чтобы изолировать, тестировать и повторно использовать их, хорошим тоном будет создать custom repository class для сущности и добавить в него методы с запросами.

Чтобы сделать это добавьте имя репозиторного класса в отбражение.

- *Annotations*

  ```
  // src/Acme/StoreBundle/Entity/Product.php
  namespace Acme\StoreBundle\Entity;

  use Doctrine\ORM\Mapping as ORM;

  /**
   * @ORM\Entity(repositoryClass="Acme\StoreBundle\Repository\ProductRepository")
   */
  class Product
  {
  ```

```
        //...
    }
```

- *YAML*

```
# src/Acme/StoreBundle/Resources/config/doctrine/Product.orm.yml
Acme\StoreBundle\Entity\Product:
    type: entity
    repositoryClass: Acme\StoreBundle\Repository\ProductRepository
    # ...
```

- *XML*

```
<!-- src/Acme/StoreBundle/Resources/config/doctrine/Product.orm.xml -->
<!-- ... -->
<doctrine-mapping>

    <entity name="Acme\StoreBundle\Entity\Product"
            repository-class="Acme\StoreBundle\Repository\ProductRepository">
            <!-- ... -->
    </entity>
</doctrine-mapping>
```

Doctrine может создать репозиторный класс с помощью команды, использованной ранее для создания пропущенных getter и setter методов:

```
php app/console doctrine:generate:entities Acme
```

Затем добавьте новый метод - `findAllOrderedByName()` - к только что созданному репозитороному классу. Он будет запрашивать все сущности `Product`, сортированные в алфавитном порядке.

```
// 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();
    }
}
```

Совет: Менеджер сущностей доступен через `$this->getEntityManager()` внутри ре-

позитория.

Можете использовать этот новый метод как и ранее доступные по умолчанию поисковые методы репозитория:

```
$em = $this->getDoctrine()->getEntityManager();
$products = $em->getRepository('AcmeStoreBundle:Product')
            ->findAllOrderedByName();
```

Примечание: Когда используется custom repository class, всё ещё есть доступ к таким поисковым методам как `find()` и `findAll()`.

2.8.3 Связи/объединения сущностей

Предположим что все продукты в приложении принадлежат единственной "категории". В этом случае, необходим объект `Category` и способ связывания его с объектом `Product`. Начнём с соаздания сущности `Category`. Так как известно что в конечном счёте понадобится сохранить класс с помощью Doctrine, то можно позволить Doctrine создать его для вас.

```
php app/console doctrine:generate:entity --entity="AcmeStoreBundle:Category" --fields="name:stri
```

Это задание создаст сущность `Category` с полями `id`, `name` и связанными getter и setter функциями.

Метаданные отображения связей

Чтобы связать сущности `Category` и `Product`, начните с создания свойства `products` в классе `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();
    }
}
```

Во-первых, т. к. объект `Category` связан со множеством объектов `Product`, то добавленное свойство `products` будет массивом для хранения объектов `Product`. Далее, this isn't done because Doctrine needs it, but instead because it makes sense in the application for each `Category` to hold an array of `Product` objects.

Примечание: Код в методе `__construct()` важен, потому что Doctrine необходимо чтобы свойство `$products` было объектом `ArrayCollection`. Этот объект выглядит и работает почти *также* как массив, но имеет расширенную гибкость. Если это заставляет вас чувствовать неудобство, то не переживайте. Представьте что это просто `массив` и вы будете снова в хорошей форме.

Далее, т. к. каждый класс `Product` может связываться только с одним объектом `Category`, необходимо добавить свойство `$category` к классу `Product`:

```
// src/Acme/StoreBundle/Entity/Product.php
// ...

class Product
{
    // ...

    /**
     * @ORM\ManyToOne(targetEntity="Category", inversedBy="products")
     * @ORM\JoinColumn(name="category_id", referencedColumnName="id")
     */
    protected $category;
}
```

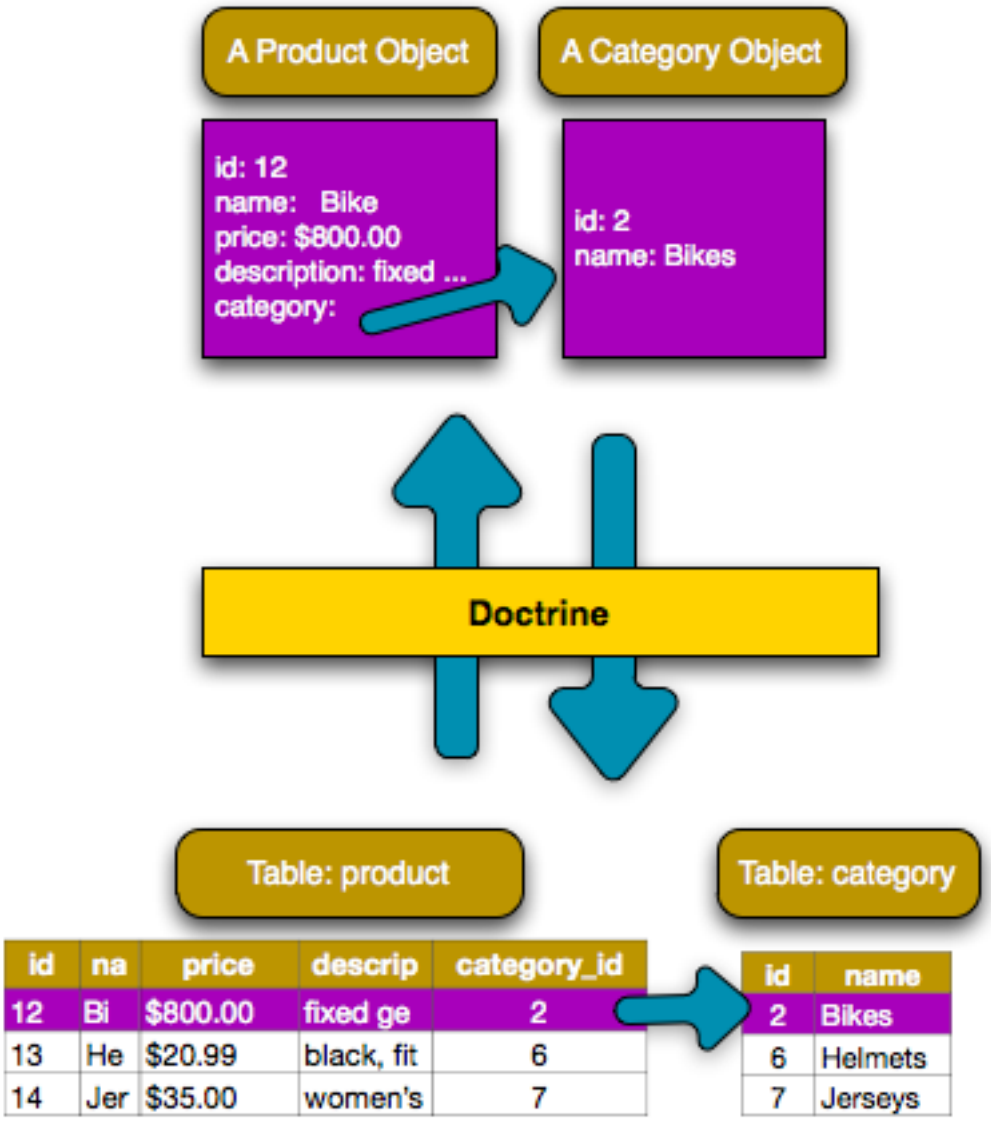
Наконец, когда добавлены новые свойства к обоим классам `Category` и `Product`, сообщите Doctrine что надо создать отсутствующие методы getter и setter:

```
php app/console doctrine:generate:entities Acme
```

Забудьте о метаданных Doctrine на секунду. Имеется два класса - `Category` и `Product` with a natural one-to-many relationship. Класс `Category` holds массив объектов `Product` и объект `Product` может hold один объект `Category`. Другими словами - классы построены таким способом, который имеет смысл для вашей задачи. А тот факт, что данные должны быть сохранены в базу данных, всегда второстепенен.

Теперь взгляните на метаданные над свойством `$category` в классе `Product`. Эта информация сообщает doctrine что связанным классом является `Category` и что он должен

хранить `id` от записи категории в поле `category_id`, находящемся в таблице `product`. Другими словами, связанный объект `Category` будет хранится в свойстве `$category`, но, за кулисами, Doctrine будет хранить эту связь, записывая значение id категории в столбец `category_id` таблицы `product`.



Метаданные над свойством `$products` объекта `Category` менее важны и попросту сообщают Doctrine что нужно посмотреть свойство `Product.category` чтобы вычислить как отображается связь.

Перед тем как продолжить, убедитесь что сообщили Doctrine добавить новые таблицу `category` и столбец `product.category_id`, а также новый внешний ключ:

```
php app/console doctrine:schema:update --force
```

Примечание: Эта задача должна выполняться только во время разработки. Более надёжный способ систематических обновлений производственной базы данных описан в Миграциях Doctrine.

Сохранение связанных сущностей

Теперь давайте посмотрим код в действии. Представьте, что вы внутри контроллера:

```
// ...
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);
        // Связывает этот продукт с категорией
        $product->setCategory($category);

        $em = $this->getDoctrine()->getEntityManager();
        $em->persist($category);
        $em->persist($product);
        $em->flush();

        return new Response(
            'Created product id: '.$product->getId().' and category id: '.$category->getId()
        );
    }
}
```

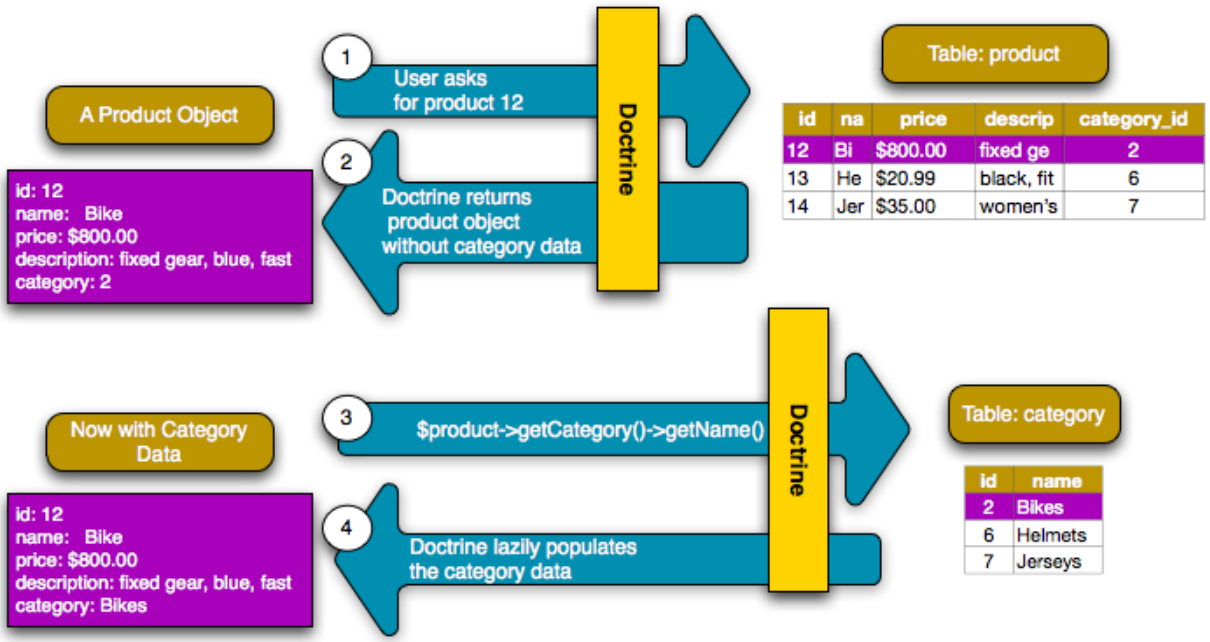
Итак, одна строка добавлена в таблицы `category` и `product`. В столбец `product.category_id` для нового продукта установлен тот `id`, который соотвествует новой категории. Doctrine осуществляет сохранение этой связи для вас.

Получение связанных объектов

Когда необходимо получить объединённые объекты, рабочий процесс выглядит также как и раньше. Сначала получаете объект `$product`, а затем доступ к связанной `Category`:

```
public function showAction($id)
{
    $product = $this->getDoctrine()
        ->getRepository('AcmeStoreBundle:Product')
        ->find($id);

    $categoryName = $product->getCategory()->getName();

    // ...
}
```

В этом примере, сначала запрашивается объект `Product` по `id` продукта. Этот запрос выдаёт ответ *только* для данных о продукте и гидратирует (hydrate) объект `$product` с этими данными. Затем, когда вызовется `$product->getCategory()->getName()`, Doctrine без лишнего шума сделает второй запрос, чтобы найти `Category`, которая связана с этим `Product`. Она подготовит объект `$category` и возвратит его вам.



Важен тот факт, что у вас есть простой доступ к категории, связанной с продуктом, но её данные не извлекаются, пока она вам не понадобится (т. е. это “ленивая загрузка”).

Также можно запросить в другом направлении:

```
public function showProductAction($id)
{
    $category = $this->getDoctrine()
        ->getRepository('AcmeStoreBundle:Category')
        ->find($id);

    $products = $category->getProducts();

    // ...
}
```

В этом случае происходят похожие дела: сначала запрашиваете один объект `Category`, затем Doctrine делает второй запрос для получения связанных объектов `Product`, но только однажды - когда они вам понадобятся (т. е. когда вызывается `->getProducts()`). Переменная `$products` является массивом всех объектов `Product`, связанных с данным объектом `Category` через значение их `category_id`.

Связи и proxy классы

Эта "ленивая загрузка" возможна, когда необходима, потому, что Doctrine возвращает "proxy" объект вместо настоящего объекта. Взгляните снова на пример, приведённый ранее:

```
$product = $this->getDoctrine()
    ->getRepository('AcmeStoreBundle:Product')
    ->find($id);

$category = $product->getCategory();

// prints "Proxies\AcmeStoreBundleEntityCategoryProxy"
echo get_class($category);
```

Этот proxy объект расширяет настоящий объект `Category`, и выглядит и действует так же как и он. Отличие лишь в том, что используя proxy объект, Doctrine может отложить запрос действительных данных о `Category` пока они вам не понадобятся (т. е. пока не вызовете `$category->getName()`).
Proxy классы создаются Doctrine и хранятся в папке cache. И хотя вам, вероятно, никогда не придётся принимать во внимание что объект `$category` на самом деле является proxy объектом, но важно знать об этом.
В следующем разделе будем получать данные о продукте и категории за один заход (через *join*), а Doctrine будет возвращать *настоящий* объект `Category`, т. к. не будет нужды в ленивой загрузке.

Объединение со связанными записями

В предыдущих примерах выполнялось по два запроса - один для исходного объекта (например, `Category`) и один для связанного (например, объекты `Product`).

Совет: Вспомните, что можно увидеть все запросы к базе данных, сделанные во время веб-запроса, через панель инструментов web debug.

Конечно, если заранее известно что будет необходим доступ к обоим объектам, то можно избежать второго запроса, используя join в исходном запросе. Добавьте следующий метод к классу `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;
    }
}
```

Теперь можете использовать этот метод в контроллере чтобы получать объект `Product` и связанную `Category` за один запрос:

```
public function showAction($id)
{
    $product = $this->getDoctrine()
        ->getRepository('AcmeStoreBundle:Product')
        ->findOneByIdJoinedToCategory($id);

    $category = $product->getCategory();

    // ...
}
```

Подробнее об объединениях

Этот раздел является введением к одному общему типу связи сущностей - связи один-ко-многим. За более продвинутыми подробностями и примерами использования других типов связей (напр., `один-к-одному`, `многие-ко-многим`), обращайтесь к Отображениям объединений для Doctrine.

Примечание: Если использовать аннотации, необходимо предварять их упоминаниями об `ORM\` (напр., `ORM\OneToMany`), про это не говорится в документации Doctrine. Также необходимо включить выражение `use Doctrine\ORM\Mapping as ORM;`, которое *внедряет* префикс аннотации `ORM`.

2.8.4 Конфигурация

Doctrine очень гибка, хотя вам, вероятно, никогда не придёться беспокоиться о большей части её опций. Чтобы узнать больше о настройке Doctrine, see the Doctrine section of the `reference manual`.

2.8.5 Lifecycle Callbacks

Иногда требуется выполнить действия сразу же перед или после того как сущность будет вставлена, обновлена или же удалена. Такие типы действий известны как "lifecycle" callbacks, т. к. они вызывают методы, которые необходимо выполнить во время различных стадий жизненного цикла сущности (напр., сущность вставлена, обновлена, удалена и т. д.).

Если для метаданных вы используете аннотации, то начните с включения lifecycle callbacks. В этом нет необходимости если для отображений используются YAML или XML:

```
/**
 * @ORM\Entity()
 * @ORM\HasLifecycleCallbacks()
 */
class Product
{
    // ...
}
```

Теперь можно дать задание Doctrine выполнить метод для любого доступного события жизненного цикла. Например, надо установить текущую дату в колонку `created` только во время первого сохранения сущности (т. е. во время вставки):

- *Annotations*

```
/**
 * @ORM\prePersist
 */
public function setCreatedValue()
{
    $this->created = new \DateTime();
}
```

- *YAML*

```
# src/Acme/StoreBundle/Resources/config/doctrine/Product.orm.yml
Acme\StoreBundle\Entity\Product:
    type: entity
    # ...
    lifecycleCallbacks:
        prePersist: [ setCreatedValue ]
```

- *XML*

```
<!-- src/Acme/StoreBundle/Resources/config/doctrine/Product.orm.xml -->
<!-- ... -->
<doctrine-mapping>

    <entity name="Acme\StoreBundle\Entity\Product">
            <!-- ... -->
            <lifecycle-callbacks>
                <lifecycle-callback type="prePersist" method="setCreatedValue" />
            </lifecycle-callbacks>
    </entity>
</doctrine-mapping>
```

Примечание: Предыдущие примеры предполагают что свойство `created` уже создано и отображено (здесь это не было показано).

Сразу же перед первым сохранением сущности, Doctrine автоматически вызовет этот метод и в поле `created` будет установлена текущая дата.

То же самое можно проделать для любого другого события жизненного цикла, среди которых:

- `preRemove`
- `postRemove`
- `prePersist`
- `postPersist`
- `preUpdate`

- `postUpdate`
- `postLoad`
- `loadClassMetadata`

Дополнительная информация о том, что из себя представляют эти события и вызовы внутри жизненного цикла в общем виде, находится в Документации по Lifecycle Events

Lifecycle Callbacks и Event Listeners

Обратите внимание что метод `setCreatedValue()` не получает аргументов. Это необходимость для lifecylce callbacks и это сделано преднамеренно: lifecycle callbacks должны быть простыми методами, занимающимися внутренними изменениями данных для сущности (напр., установка значений для полей created/updated, создание slug).
Если планируется делать более тяжёлую работу - запись логов или отправка email - необходимо зарегистрировать внешний класс как event listener или subscriber и дать ему доступ к необходимым ресурсам. Дополнительную информацию найдёте в `/cookbook/doctrine/event_listeners_subscribers`.

2.8.6 Расширения для Doctrine: Timestampable, Sluggable и другие

Doctrine расширяема, поэтому доступно множество сторонних решений, позволяющих с лёгкостью выполнять повторяющиеся и общие задачи над сущностями. Среди них есть следующие: *Sluggable*, *Timestampable*, *Loggable*, *Translatable* и *Tree*.

Подробнее о том где найти и как использвать эти расширения расказывает статья `Использование общих расширений Doctrine`.

2.8.7 Справка по типам полей в Doctrine

Doctrine представляет огромное количество типов полей. Каждый из которых отображает тип данных из PHP в установленный тип колонки для любой используемой базы данных. В Doctrine поддерживаются следующие типы:

- **Строки**
  - `string` (используется для коротких строк)
  - `text` (используется для длинных строк)
- **Числа**
  - `integer`

  - `smallint`
  - `bigint`
  - `decimal`
  - `float`
- **Дата и время** (используйте объект DateTime в PHP для этих полей)
  - `date`
  - `time`
  - `datetime`
- **Другие типы**
  - `boolean`
  - `object` (сериализуется и хранится в поле `CLOB`)
  - `array` (сериализуется и хранится в поле `CLOB`)

Дополнительная информация содержится в Отображении типов.

Опции полей

Каждое поле может иметь набор опций, применимых к нему. Доступные опции включают: `type` (стандартный для `string`), `name`, `length`, `unique` и `nullable`. Несколько примеров таких аннотаций:

```
/**
 * Строковое поле длиной 255, которое не должно быть null
 * (это стандартные значения для опций "type", "length" и *nullable*)
 *
 * @ORM\Column()
 */
protected $name;

/**
 * Строковое поле длиной 150, хранящееся в колонке "email_address"
 * и имеющее уникальный индекс.
 *
 * @ORM\Column(name="email_address", unique="true", length="150")
 */
protected $email;
```

Примечание: Существуют ещё опции, о которых здесь не упоминается. За дополнительной информацией обращайтесь к документации Doctrine's Property Mapping documentation

2.8.8 Консольные команды

Интеграция Doctrine2 ORM предлагает несколько консольных команд внутри пространства имён `doctrine`. Чтобы вывести список команд запустите консоль без аргументов:

```
php app/console
```

В выведенном списке доступных команд многие из них начинаются с префикса `doctrine:`. Подробнее о них (или любых других командах для Symfony) можно узнать запустив команду `help`. Например, чтобы получить подробности о процессе `doctrine:database:create`, запустите:

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

Некоторые интересные или примечательные команды включают:

- `doctrine:ensure-production-settings` - проверяет текущее окружение, настроено ли оно эффективно для производственных нужд. Она всегда должна запускаться в окружении `prod`:

  ```
  php app/console doctrine:ensure-production-settings --env=prod
  ```

- `doctrine:mapping:import` - разрешает Doctrine проанализировать существующую базу данных и создать информацию для её отображения. За дополнительной информацией обращайтесь к `/cookbook/doctrine/reverse_engineering`.
- `doctrine:mapping:info` - расскажет обо всех сущностях, которые знает Doctrine, а также есть ли в отображениях какие-нибудь простые ошибки.
- `doctrine:query:dql` и `doctrine:query:sql` - позволяет выполнять DQL или SQL запросы прямо из командной строки.

Примечание: Чтобы иметь возможность загружать fixtures с данными в базу данных, необходимо установить бандл `DoctrineFixturesBundle`. Чтобы узнать как это сделать, прочтите статью "`/bundles/DoctrineFixturesBundle/index`" в документации.

2.8.9 Выводы

Применяя Doctrine, можно сфокусироваться на объектах и их использовании в приложении и только потом заботиться об их сохранении в базу данных. Благодаря тому, что Doctrine позволяет использовать любой объект PHP для хранения данных и применяет информацию метаданных для отображения чтобы отобразить эти данные об объекте в определённую таблицу базы данных.

Хотя в основе Doctrine простая идея, она необычайно мощна, позволяет создавать сложные запросы и подписываться на события, которые дают возможность совершать различные действия когда объекты проходят по своим жизненным циклам во время сохранения.

За дополнительной информацией о Doctrine обращайтесь к разделу *Doctrine* из *Книги рецептов*, который включает следующие статьи:

- `/bundles/DoctrineFixturesBundle/index`
- `/cookbook/doctrine/common_extensions`

2.9 Тестирование

Как только вы пишете новую строку кода, вы также потенциально добавляете новые ошибки. Для того чтобы создавать надёжные приложения, вы должны использовать как функциональные, так и модульные (unit) тесты.

2.9.1 Тестовый фреймворк PHPUnit

В Symfony2 интегрирована поддержка независимой библиотеки - называемой PHPUnit - чтобы предоставить вам отличный тестовый фреймворк. Эта глава не покрывает все нюансы PHPUnit, так как вы всегда можете почитать его подробную документацию.

Примечание: Symfony2 работает с PHPUnit 3.5.11 или старше.

Каждый тест - вне зависимости от того функциональный он или модульный - это PHP класс, который расположен в поддиректории *Tests/* вашиг пакетов. Если вы будете следовать этому правилу, то вы сможете запускать все тесты вашего приложения при помощи команды:

```
# укажите папку с конфигами в командной строке
$ phpunit -c app/
```

Опция `-c` указывает PHPUnit искать конфигурационный файл в директории `app/`. Если вы интересуетесь опциями PHPUnit, обратите внимание на файл `app/phpunit.xml.dist`.

Совет: Покрытие кода может быть получено с помощью опции `--coverage-html`.

2.9.2 Модульные тесты

Модульный тест - это как правило тест одного отдельного PHP класса. Если вы хотите тестировать поведение вашего приложения целиком, обратитесь к секции Функциональные тесты.

Написание модульных тестов в Symfony2 не отличается от написания стандартных модульных тестов PHPUnit. Например, предположим, у вас есть *очень* простой класс `Calculator` в директории `Utility/` вашего пакета:

```
<?php
// src/Acme/DemoBundle/Utility/Calculator.php
namespace Acme\DemoBundle\Utility;

class Calculator
{
    public function add($a, $b)
    {
        return $a + $b;
    }
}
```

Для того, чтобы его протестировать, создайте файл `CalculatorTest` в директории `Tests/Utility` вашего пакета:

```
<?php
// 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);
    }
}
```

Примечание: По соглашению, под-директория `Tests/` должна повторять структуру директорий вашего пакета. Т.о. если вы тестируете класс вашего пакета из директории `Utility/`, поместите тест в директорию `Tests/Utility/`.

Как и в вашем приложении, автозагрузка включается автоматически при помощи файла `bootstrap.php.cache` (это по умолчанию настроено в файле `phpunit.xml.dist`).

Выполнить тесты для заданного файла или папки также просто:

```
# run all tests in the Utility directory
$ phpunit -c app src/Acme/DemoBundle/Tests/Utility/

# run tests for the Calculator class
$ phpunit -c app src/Acme/DemoBundle/Tests/Utility/CalculatorTest.php

# запустить все тесты для целого Bundle
$ phpunit -c app src/Acme/DemoBundle/
```

2.9.3 Функциональные тесты

Функциональные тесты проверяют объединения различных слоёв приложения (от маршрутизации до видов). Они не отличаются от модульных тестов настолько, насколько PHPUnit позволяет это, но имеют конкретный рабочий процесс:

- Сделать запрос;
- Протестировать ответ;
- Кликнуть по ссылке или отправить форму;
- Протестировать ответ;
- Профильтровать и повторить.

Ваш первый функциональный тест

Функциональные тесты - это простые PHP классы, которые, как правило, располагаются в директории пакета `Tests/Controller`. Если вы хотите протестировать страницы, которые содержит ваш класс `DemoController`, создайте новый класс, который расширяет специальный класс `WebTestCase`.

Например, Symfony2 Standard Edition предоставляет простой функциональный тест для его `DemoController` (DemoControllerTest), который выглядит так:

```
<?php
// 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);
    }
}
```

Совет: Для запуска ваших функциональных тестов, класс `WebTestCase` загружает ядро вашего приложения. В большинстве случаев, это происходит автоматически. Тем не менее, если ваше ядро находится в нестандартной директории, вам нужно модифицировать файл `phpunit.xml.dist` и установить переменную среды `KERNEL_DIR` на директорию вашего ядра:

```
<phpunit
    <!-- ... -->
    <php>
        <server name="KERNEL_DIR" value="/path/to/your/app/" />
    </php>
    <!-- ... -->
</phpunit>
```

Метод `createClient()` возвращает клиент, который напоминает браузер, который вы используете для просмотра вашего сайта:

```
$crawler = $client->request('GET', '/demo/hello/Fabien');
```

Метод `request()` (см. *подробнее о методе request*) возвращает объект `Symfony\Component\DomCrawler\Crawler`, который может быть использован для выбора элементов в Response, кликов по ссылкам и отправки форм.

Совет: Crawler может использоваться только в том случае, если содержимое Response это XML или HTML документ. Для других типов нужно получать содержимое Response через `$client->getResponse()->getContent()`.

Давайте кликнем по ссылке, выбрав её при помощи Crawler используя XPath или CSS селектор, затем используем Client для собственно клика. Например, следующий код находит все ссылки с текстом `Greet`, затем выбирает вторую из них и кликает на неё:

```
$link = $crawler->filter('a:contains("Greet")')->eq(1)->link();

$crawler = $client->click($link);
```

Отправка формы происходит схожим образом: выберите кнопку на форме, по желанию переопределите какие-нибудь значения формы, и отправьте её:

```
<?php
$form = $crawler->selectButton('submit')->form();

// устанавливает какие-нибудь значения
$form['name'] = 'Lucas';
$form['form_name[subject]'] = 'Hey there!';

// отправляет форму
$crawler = $client->submit($form);
```

Совет: Форма также поддерживет загрузку файлов и содержит методы для заполнения различных типов полей (например, `select()` и `tick()`). Подробнее читайте в секции Формы ниже.

Теперь, когда вы с лёгкостью можете перемещаться по приложению, воспользуйтесь утверждениями чтобы проверить ожидаемые действия. Воспользуйтесь Crawler чтобы сделать утверждения для DOM:

```
// Утверждает что ответ соответствует заданному CSS селектору.
$this->assertTrue($crawler->filter('h1')->count() > 0);
```

Или проверьте содержимое Response напрямую, если хотите убедиться что его содержимое включает какой-то текст, или что Response не является документом XML/HTML:

```
$this->assertRegExp('/Hello Fabien/', $client->getResponse()->getContent());
```

Подробнее о методе `request()`:

Полная сигнатура метода `request()`:

```
<?php
request(
    $method,
    $uri,
    array $parameters = array(),
    array $files = array(),
    array $server = array(),
    $content = null,
    $changeHistory = true
)
```

Массив `server` - это значения, которые вы как правило ожидаете найти в суперглобальном массиве $_SERVER. Например, для того, чтобы установить HTTP заголовки *Content-Type* и *Referer* вы должны передать следующее:

```
<?php
$client->request(
    'GET',
    '/demo/hello/Fabien',
    array(),
    array(),
    array(
        'CONTENT_TYPE' => 'application/json',
        'HTTP_REFERER' => '/foo/bar',
    )
);
```

Для быстрого старта обратите внимание на список наиболе типовых и полезных утверждений:

```
<?php
// Утверждает что имеется единственный маг h2 с классом "subtitle"
$this->assertTrue($crawler->filter('h2.subtitle')->count() > 0);

// Утверждает что на странице имеется 4 тага h2
$this->assertEquals(4, $crawler->filter('h2')->count());

// Утверждает что заголовок "Content-Type" - "application/json"
$this->assertTrue($client->getResponse()->headers->contains('Content-Type', 'application/js

// Утверждает что тело ответа соответствует регулярному выражению
$this->assertRegExp('/foo/', $client->getResponse()->getContent());

// Утверждает что статус-код ответа 2xx
```

```
$this->assertTrue($client->getResponse()->isSuccessful());
// Утверждает что статус-код ответа 404
$this->assertTrue($client->getResponse()->isNotFound());
// Утверждает что статус-код ответа точно 200
$this->assertEquals(200, $client->getResponse()->getStatusCode());

// Утверждает что ответ - это перенаправление на /demo/contact
$this->assertTrue($client->getResponse()->isRedirect('/demo/contact'));
// или просто проверяет, что ответ - это перенаправление на любой URL
$this->assertTrue($client->getResponse()->isRedirect());
```

2.9.4 Работаем с Тестовым клиентом

Тестовый клиент симулирует HTTP клиент (как правило это браузер) и выполняет запросы к вашему Symfony2 приложению:

```
$crawler = $client->request('GET', '/hello/Fabien');
```

Метод `request()` принимает в качестве аргументов HTTP метод и URL и возвращает экземпляр `Crawler`.

Используйет Crawler для нахождения DOM-элементов в теле Response. После эти элементы могут быть использованы для кликов по ссылкам и отправки форм:

```
<?php
$link = $crawler->selectLink('Go elsewhere...')->link();
$crawler = $client->click($link);

$form = $crawler->selectButton('validate')->form();
$crawler = $client->submit($form, array('name' => 'Fabien'));
```

Методы `click()` и `submit()` возвращают объект `Crawler`. Эти методы - лучший способ просматривать ваше приложение, так как они заботятся о многих вещах, например определении HTTP метода формы, и предоставляют вам удобный API для загрузки файлов.

Совет: Больше узнать об объектах `Link` и `Form` можно в разделе *Crawler*.

Метод `request` может также быть использован для симуляции отправки форм или для выполнения более сложных запросов:

```
<?php
// Прямая отправка формы (можно и так, но легче использовать Crawler!)
$client->request('POST', '/submit', array('name' => 'Fabien'));
```

```
// Отправка формы с загрузкой файла
use Symfony\Component\HttpFoundation\File\UploadedFile;

$photo = new UploadedFile(
    '/path/to/photo.jpg',
    'photo.jpg',
    'image/jpeg',
    123
);
// или
$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)
);

// Выполнение DELETE запросов, и отправка HTTP заголовков
$client->request(
    'DELETE',
    '/post/12',
    array(),
    array(),
    array('PHP_AUTH_USER' => 'username', 'PHP_AUTH_PW' => 'pa$$word')
);
```

И последнее, но не менее важное, можно заставить каждый запрос выполняться в собственном процессе PHP чтобы избежать любых побочных эффектов когда несколько клиентов работают в одном скрипте:

```
$client->insulate();
```

Браузинг

Клиент поддерживает многие операции, свойственные настоящему браузеру:

```
<?php
$client->back();
$client->forward();
```

```
$client->reload();

// Очищает все куки и историю.
$client->restart();
```

Получение внутренних объектов

Когда клиент используется для тестирования приложения, возникает необходимость получить доступ к его внутренним объектам:

```
<?php
$history   = $client->getHistory();
$cookieJar = $client->getCookieJar();
```

Также можно получить объекты, относящиеся к последнему запросу:

```
<?php
$request  = $client->getRequest();
$response = $client->getResponse();
$crawler  = $client->getCrawler();
```

Если запросы не были изолированы, то можно получить доступ к `Container` и `Kernel`:

```
<?php
$container = $client->getContainer();
$kernel    = $client->getKernel();
```

Получение Container

Настоятельно рекомендуется использовать функциональные тесты только для проверки Response. Но в некоторых редких случаях необходимо получить доступ к каким-либо внутренним объектам для написания утверждений. Для этого можно использовать контейнер внедрения зависимости:

```
$container = $client->getContainer();
```

Имейте в виду что это не сработает если вы изолировали клиента или использовали HTTP слой. Для получения списка служб, доступных в вашем приложении, используйте консольную команду `container:debug`.

Совет: Если необходимая для проверки информация доступна из профилировщика, тогда используйте его.

Получение данных профилировщика

Для каждого запроса профайлер Symfony собирает и сохраняет множество данных о том как обрабатывается этот запрос. Например, профайлер может быть использован для верификации, что данная страница выполняет SQL запросов меньше, чем некоторое пороговое значение.

Для получения профайлера для последнего запроса выполните следующий код:

```
$profile = $client->getProfile();
```

Подробнее про использование профайлера в тестах читайте в книге рецептов: *Как использовать профилировщик в Функциональном тесте*.

Перенаправление

Когда запрос возвращает ответ с перенаправлением, клиент автоматически следует ему. Если вы хотите проверить ответ перед перенаправлением, вы можете указать клиенту не следовать перенаправлению припомощи метода `followRedirects()`:

```
$client->followRedirects(false);
```

Если клиент не следует перенаправлениям, вы можете форсировать перенаправление при помощи метода `followRedirect()`:

```
$crawler = $client->followRedirect();
```

2.9.5 Crawler

Экземпляр Crawler возвращается каждый раз когда выполняется запрос посредством клиента. Он позволяет перемещаться по HTML документам, выбирать узлы, искать ссылки и формы.

Перемещения

Как и jQuery, Crawler имеет методы для перемещения по DOM документа HTML/XML. Например, следующий код находит все элементы `input[type=submit]`, выбирает последний на странице и выбирает ближайший родительский элемент:

```
<?php
$newCrawler = $crawler->filter('input[type=submit]')
    ->last()
    ->parents()
    ->first()
;
```

Также доступны следующие методы:

Метод	Описание
`filter('h1.title')`	Ноды, соответствующие CSS селектору
`filterXpath('h1')`	Ноды, соответствующие выражению XPath
`eq(1)`	Ноды с определённым индексом
`first()`	Первый нод
`last()`	Последний нод
`siblings()`	Элементы одного уровня (сёстры)
`nextAll()`	Все последующие сёстры
`previousAll()`	Все предыдущие сёстры
`parents()`	Родительские ноды
`children()`	Потомки
`reduce($lambda)`	Ноды, для которых функция не возвращает false

Так как каждый метод возвращает новый экземпляр `Crawler`, вы можете упростить ваш код путём выстраивания вызовов в цепочку:

```
<?php
$crawler
    ->filter('h1')
    ->reduce(function ($node, $i)
    {
        if (!$node->getAttribute('class')) {
            return false;
        }
    })
    ->first();
```

Совет: Используйте функцию `count()` чтобы получить количество узлов, хранящихся в Crawler: `count($crawler)`

Извлечение информации

Crawler может извлечь информацию из узлов:

```
<?php
// Возвращает значение атрибута для первого узла
$crawler->attr('class');

// Возвращает значение узла для первого узла
$crawler->text();

// Возвращает массив для каждого элемента с его значением и ссылкой
$info = $crawler->extract(array('_text', 'href'));
```

```
// Выполняет lambda для каждого узла и возвращает массив результатов
$data = $crawler->each(function ($node, $i)
{
    return $node->attr('href');
});
```

Ссылки

Можно выбирать ссылки с помощью методов обхода, но сокращение `selectLink()` часто более удобно:

```
$crawler->selectLink('Click here');
```

Оно выбирает ссылки, содержащие указанный текст, либо изображения, по которым можно кликать, содержащие этот текст в атрибуте `alt`.

Клиентский метод `click()` принимает экземпляр `Link`, возвращаемый методом `link()`:

```
$link = $crawler->link();

$client->click($link);
```

Совет: Метод `links()` возвращает массив объектов `Link` для всех узлов.

Формы

Как и ссылки, формы выбирайте методом `selectButton()`:

```
$crawler->selectButton('submit');
```

Заметьте что выбирается кнопка на форме, а не сама форма, т. к. она может иметь несколько кнопок; если используются API перемещений, то помните что надо искать кнопку.

Метод `selectButton()` может выбрать теги `button` и `input` с типом submit; в нём заложено несколько эвристик для их нахождения по:

- значению атрибута `value`;
- значению атрибута `id` или `alt` для изображений;
- значению атрибута `id` или `name` для тегов `button`.

Когда имеется узел, описывающий кнопку, вызовите метод `form()` чтобы получить экземпляр `Form`, формы обёртывающей его:

```
$form = $buttonCrawlerNode->form();
```

При вызове метода `form()` можно передать массив значений для полей, перезаписывающих начальные значения:

```
$form = $buttonCrawlerNode->form(array(
    'name'              => 'Fabien',
    'my_form[subject]'  => 'Symfony rocks!',
));
```

А если надо симулировать определённый HTTP метод для формы, передайте его вторым аргументом:

```
$form = $crawler->form(array(), 'DELETE');
```

Клиент может отправлять эзкемпляры `Form`:

```
$client->submit($form);
```

Значения полей могут быть переданы вторым аргументом метода `submit()`:

```
$client->submit($form, array(
    'name'              => 'Fabien',
    'my_form[subject]'  => 'Symfony rocks!',
));
```

В более сложных случаях, используйте экземпляр `Form` как массив чтобы задать значения каждого поля индивидуально:

```
// Изменяет значение поля
$form['name'] = 'Fabien';
$form['my_form[subject]'] = 'Symfony rocks!';
```

Здесь тоже есть красивый API для управления значениями полей в зависимости от их типов:

```
// Выбирает option или radio
$form['country']->select('France');

// Ставит галочку в checkbox
$form['like_symfony']->tick();

// Загружает файл
$form['photo']->upload('/path/to/lucas.jpg');
```

Совет: Можно получить значения, которые будут отправлены, вызвав метод `getValues()` объекта `Form`. Загружаемые файлы доступны в отдельном массиве, возвращаемом через `getFiles()`. `getPhpValues()` и `getPhpFiles()` тоже возвращают значе-

ния для отправки, но в формате PHP (он преобразует ключи с квадратными скобками - например, `my_form[subject]` - в PHP массивы).

2.9.6 Тестовая конфигурация

PHPUnit конфигурация

Каждое приложение имеет свою собственную конфигурацию PHPUnit, которая хранится в файле `phpunit.xml.dist`. Вы можете редактировать этот файл и менять значения по умолчанию или же вы можете создать файл `phpunit.xml` для подгонки конфигурации на вашей локальной машине.

Совет: Сохраните файл `phpunit.xml.dist` в вашем репозитории и игнорьте файл `phpunit.xml`.

По умолчанию, по команде `phpunit` запускаются только тесты из "стандартных" пакетов (стандартными считаются тесты в директориях `src/*/Bundle/Tests` или `src/*/Bundle/*Bundle/Tests`), но вы можете запросто добавить больше директорий. Например, следующая конфигурация добавляет тесты сторонних пакетов:

```
<!-- hello/phpunit.xml.dist -->
<testsuites>
    <testsuite name="Project Test Suite">
        <directory>../src/*/*Bundle/Tests</directory>
        <directory>../src/Acme/Bundle/*Bundle/Tests</directory>
    </testsuite>
</testsuites>
```

Для того, чтобы включить прочие директории в отчёт по покрытию кода, необходимо отредактировать секцию `<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>
```

2.9.7 Узнайте больше из Рецептов

- *Как смоделировать HTTP аутентификацию в Функциональном тесте*
- *Как тестировать взаимодействие с несколькими клиентами*
- *Как использовать профилировщик в Функциональном тесте*

2.10 Валидация

Валидация - это вполне обычная задача для web-приложения. Данные, вводимые в формы должны быть валидированы (проверены). В то же время, данные должны быть валидированы до того, как они будут записаны в базу данных или же будут переданы далее некоторому web-сервису.

Symfony2 содержит компонент Validator для того, чтобы упростить эту задачу. Этот компонент основан на документе JSR303 Bean Validation specification. Что?! Java спецификация в PHP? Однако же вы всё услышали верно, но всё не так плохо как вам могло показаться. Давайте посмотрим, как мы можем использовать это в PHP.

2.10.1 Основы Валидации

Самый лучший способ понять валидацию - это увидеть её в действии. Для начала, предположим, что вы создали обычный PHP-объект и вам нужно использовать его где-то внутри приложения:

```
<?php
// src/Acme/BlogBundle/Entity/Author.php
namespace Acme\BlogBundle\Entity;

class Author
{
    public $name;
}
```

Итак, это обычный класс, который обслуживает некоторый круг задач внутри вашего приложения. Цель валидации заключается в том, чтобы сообщить вам - являются ли данные объекта корректными (валидными). Для этого, вам нужно настроить перечень правил (называемых *Ограничениями*), которым объект должен удовлетворять, чтобы быть валидным. Эти правила могут быть указаны во многих форматах (YAML, XML, аннотации, или PHP).

Например, для того, чтобы гарантировать, что свойство `$name` не пусто, добавьте следующий код:

- *YAML*

```
# src/Acme/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\Author:
    properties:
        name:
            - NotBlank: ~
```

- *Annotations*

```
// src/Acme/BlogBundle/Entity/Author.php
use Symfony\Component\Validator\Constraints as Assert;

class Author
{
    /**
     * @Assert\NotBlank()
     */
    public $name;
}
```

- *XML*

```
<!-- src/Acme/BlogBundle/Resources/config/validation.xml -->
<?xml version="1.0" encoding="UTF-8" ?>
<constraint-mapping xmlns="http://symfony.com/schema/dic/constraint-mapping"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/dic/constraint-mapping http://symfony.com

    <class name="Acme\BlogBundle\Entity\Author">
        <property name="name">
            <constraint name="NotBlank" />
        </property>
    </class>
</constraint-mapping>
```

- *PHP*

```
<?php
// src/Acme/BlogBundle/Entity/Author.php

use Symfony\Component\Validator\Mapping\ClassMetadata;
use Symfony\Component\Validator\Constraints\NotBlank;

class Author
{
    public $name;

    public static function loadValidatorMetadata(ClassMetadata $metadata)
```

```
        {
            $metadata->addPropertyConstraint('name', new NotBlank());
        }
    }
```

Совет: Защищённые (protected) и закрытые (private) члены класса могут быть также валидированы как и любой `get`-метод (см. validator-constraint-targets).

Использование сервиса `validator`

Далее, чтобы проверить объект `Author`, используйте метод `validate` сервиса `validator` (class `Symfony\Component\Validator\Validator`). Обязанности у класса `validator` простые: прочитать ограничения (т.е. правила), определённые для класса, и определить, удовлетворяют ли данные из объекта этим ограничениям. Если валидация проходит с ошибкой, возвращает массив ошибок. Давайте рассмотрим этот простой пример контроллера:

```
<?php
use Symfony\Component\HttpFoundation\Response;
use Acme\BlogBundle\Entity\Author;
// ...

public function indexAction()
{
    $author = new Author();
    // ... выполняются какие-либо действия с объектом $author

    $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!');
    }
}
```

Если свойство `$name` пустое, вы увидите следующую ошибку:

```
Acme\BlogBundle\Author.name:
    This value should not be blank
```

Если же вы укажете некоторое непустое значение для `name`, появится сообщение об успешной валидации.

Совет: Большую часть времени вы не будете напрямую взаимодействовать с сервисом `validator` и вам не нужно будет беспокоиться об отображении ошибок. Большую часть времени вы будете использовать валидацию косвенно при обработке данных из отправленных приложению форм. Подробнее об этом написано в секции: *Валидация и Формы*.

Вы также можете передать перечень ошибок в шаблон.

```
<?php
if (count($errors) > 0) {
    return $this->render('AcmeBlogBundle:Author:validate.html.twig', array(
        'errors' => $errors,
    ));
} else {
    // ...
}
```

Внутри шаблона вы можете отобразить список ошибок так, как вам нужно:

- *Twig*

  ```
  {# src/Acme/BlogBundle/Resources/views/Author/validate.html.twig #}

  <h3>The author has the following errors</h3>
  <ul>
  {% for error in errors %}
      <li>{{ error.message }}</li>
  {% endfor %}
  </ul>
  ```

- *PHP*

  ```
  <!-- src/Acme/BlogBundle/Resources/views/Author/validate.html.php -->

  <h3>The author has the following errors</h3>
  <ul>
  <?php foreach ($errors as $error): ?>
      <li><?php echo $error->getMessage() ?></li>
  <?php endforeach; ?>
  </ul>
  ```

Примечание: Каждая ошибка валидации (называемая “constraint violation”), представлена объектом класса `Symfony\Component\Validator\ConstraintViolation`

Валидация и Формы

Сервис `validator` может быть использован в любое время для проверки объекта. В жизни же, не смотря такую возможность, вы будете работать с сервисом `validator` косвенно при обработке форм. Библиотека форм Symfony использует сервис валидации для проверки объектов форм после того, как данные были отправлены пользователем и привязаны к форме. Объекты ошибок валидации ("constraint violations") будут конвертированы в объекты `FieldError`, которые могут быть легко отображены вместе с формами. Типичный процесс отправки формы со стороны контроллера выглядит так:

```
<?php
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->getMethod() == 'POST') {
        $form->bindRequest($request);

        if ($form->isValid()) {
            // валидация прошла успешно, можно выполнять дальнейшие действия с объектом $author

            $this->redirect($this->generateUrl('...'));
        }
    }

    return $this->render('BlogBundle:Author:form.html.twig', array(
        'form' => $form->createView(),
    ));
}
```

Примечание: Этот пример использует класс формы `AuthorType`, который в этой главе не описан.

Более подробную информацию о формах вы можете получить в главе *Формы*.

2.10.2 Конфигурирование

Валидатор Symfony2 доступен по умолчанию, но вы должны тщательно настроить его при помощи аннотаций (если вы используете метод аннотаций для настройки ограни-

чений):

- *YAML*

```
# app/config/config.yml
framework:
    validation: { enable_annotations: true }
```

- *XML*

```
<!-- app/config/config.xml -->
<framework:config>
    <framework:validation enable_annotations="true" />
</framework:config>
```

- *PHP*

```
<?php
// app/config/config.php
$container->loadFromExtension('framework', array('validation' => array(
    'enable_annotations' => true,
)));
```

2.10.3 Ограничения

Валидатор создан для того, чтобы проверять объекты на соответствие *ограничениям* (т.е. правилам). Для того чтобы валидировать объект, укажите для его класса одно или более ограничений и передайте его сервису валидации (`validator`).

По сути, ограничение - это PHP объект, который выполняет проверочное выражение. В жизни ограничение может выглядеть так: "пирог не должен подгореть". В Symfony2 ограничения выглядят похожим образом: это утверждения, что некоторое выражение истинно. Учитывая значение, ограничение скажет вам, соответствует ли это значение правилу ограничения.

Поддерживаемые ограничения

Symfony2 содержит большое количество ограничений, необходимых в повседневной работе:

Базовые ограничения

Ниже перечислены базовые ограничения: они используются для самых простых проверок полей классов или значений, возвращаемых методами объектов.

- `NotBlank`

- Blank
- NotNull
- Null
- True
- False
- Type

Строковые ограничения

- Email
- MinLength
- MaxLength
- Url
- Regex
- Ip

Числовые ограничения

- Max
- Min

Ограничения дат

- Date
- DateTime
- Time

Ограничения коллекций

- Choice
- Collection
- UniqueEntity
- Language

- Locale
- Country

Ограничения файлов

- File
- Image

Прочие ограничения

- Callback
- All
- UserPassword
- Valid

Вы также можете создавать свои ограничения. Этот вопрос освещается в топике "/cookbook/validation/custom_constraint" в книге рецептов.

Конфигурация ограничений

Некоторые ограничения, как например NotBlank, просты, в то время как другие, как например Choice, имеют много различных опций. Предположим, что класс Author имеет поле gender, которое может иметь два значения - "male" или "female":

- *YAML*

```
# src/Acme/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\Author:
    properties:
        gender:
            - Choice: { choices: [male, female], message: Choose a valid gender. }
```

- *Annotations*

```
// src/Acme/BlogBundle/Entity/Author.php
use Symfony\Component\Validator\Constraints as Assert;

class Author
{
    /**
     * @Assert\Choice(
     *     choices = { "male", "female" },
     *     message = "Choose a valid gender."
```

```
     * )
     */
    public $gender;
}
```

- *XML*

```
<!-- src/Acme/BlogBundle/Resources/config/validation.xml -->
<?xml version="1.0" encoding="UTF-8" ?>
<constraint-mapping xmlns="http://symfony.com/schema/dic/constraint-mapping"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/dic/constraint-mapping http://symfony.com

    <class name="Acme\BlogBundle\Entity\Author">
        <property name="gender">
            <constraint name="Choice">
                <option name="choices">
                    <value>male</value>
                    <value>female</value>
                </option>
                <option name="message">Choose a valid gender.</option>
            </constraint>
        </property>
    </class>
</constraint-mapping>
```

- *PHP*

```
<?php
// src/Acme/BlogBundle/Entity/Author.php
use Symfony\Component\Validator\Mapping\ClassMetadata;
use Symfony\Component\Validator\Constraints\NotBlank;

class Author
{
    public $gender;

    public static function loadValidatorMetadata(ClassMetadata $metadata)
    {
        $metadata->addPropertyConstraint('gender', new Choice(array(
            'choices' => array('male', 'female'),
            'message' => 'Choose a valid gender.',
        )));
    }
}
```

Опции ограничения всегда представлены в виде массива. Однако, некоторые ограничения также позволяют вам указать одну опцию - основную, а не массив опций. В случае

с ограничением `Choice`, можно указать только варианты выбора (`choices`).

- *YAML*

```
# src/Acme/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\Author:
    properties:
        gender:
            - Choice: [male, female]
```

- *Annotations*

```
// src/Acme/BlogBundle/Entity/Author.php
use Symfony\Component\Validator\Constraints as Assert;

class Author
{
    /**
     * @Assert\Choice({"male", "female"})
     */
    protected $gender;
}
```

- *XML*

```
<!-- src/Acme/BlogBundle/Resources/config/validation.xml -->
<?xml version="1.0" encoding="UTF-8" ?>
<constraint-mapping xmlns="http://symfony.com/schema/dic/constraint-mapping"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/dic/constraint-mapping http://symfony.com

    <class name="Acme\BlogBundle\Entity\Author">
        <property name="gender">
            <constraint name="Choice">
                <value>male</value>
                <value>female</value>
            </constraint>
        </property>
    </class>
</constraint-mapping>
```

- *PHP*

```
<?php
// src/Acme/BlogBundle/Entity/Author.php
use Symfony\Component\Validator\Mapping\ClassMetadata;
use Symfony\Component\Validator\Constraints\Choice;

class Author
```

```
{
    protected $gender;

    public static function loadValidatorMetadata(ClassMetadata $metadata)
    {
        $metadata->addPropertyConstraint('gender', new Choice(array('male', 'female')));
    }
}
```

Такая возможность позволяет сделать настройку базовых опций ограничения короче и быстрее.

Если вы не уверены, как нужно указывать опцию, или же сверьтесь с документацией API для ограничения или же поступайте просто - всегда передавайте массив опций (как показано выше в первом примере).

2.10.4 Цели для ограничений

Ограничение могут быть применены к свойству класса (например, `name`) или же к публичному аксессору (или геттеру, например, `getFullName`). Первый вариант наиболее простой и чаще всего встречающийся, но второй вариант позволяет вам создавать более сложные правила валидации.

Поля класса

Валидация полей класса - это наиболее простая техника валидации. Symfony2 позволяет вам выполнять валидацию приватных, защищённых и публичных полей. Следующий листинг показывает как настроить поле `$firstName` класса `Author`, чтобы оно имело как минимум три символа.

- *YAML*

```
# src/Acme/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\Author:
    properties:
        firstName:
            - NotBlank: ~
            - MinLength: 3
```

- *Annotations*

```
// Acme/BlogBundle/Entity/Author.php
use Symfony\Component\Validator\Constraints as Assert;

class Author
{
```

```
    /**
     * @Assert\NotBlank()
     * @Assert\MinLength(3)
     */
    private $firstName;
}
```

- *XML*

```
<!-- src/Acme/BlogBundle/Resources/config/validation.xml -->
<class name="Acme\BlogBundle\Entity\Author">
    <property name="firstName">
        <constraint name="NotBlank" />
        <constraint name="MinLength">3</constraint>
    </property>
</class>
```

- *PHP*

```
<?php
// src/Acme/BlogBundle/Entity/Author.php
use Symfony\Component\Validator\Mapping\ClassMetadata;
use Symfony\Component\Validator\Constraints\NotBlank;
use Symfony\Component\Validator\Constraints\MinLength;

class Author
{
    private $firstName;

    public static function loadValidatorMetadata(ClassMetadata $metadata)
    {
        $metadata->addPropertyConstraint('firstName', new NotBlank());
        $metadata->addPropertyConstraint('firstName', new MinLength(3));
    }
}
```

Методы класса

Ограничения также могут быть применены к значениям, возвращаемым методами. Symfony2 позволяет вам добавлять ограничения к любому *публичному* методу, если его имя начинается с "get" или "is". В этом руководстве оба этих типа методов называются "геттерами" (от `getters`).

Выгода от этой техники в том, что она позволяет вам валидировать ваш проект динамически. Например, предположим, что вы хотите быть уверенными, что поле пароля не соответствует имени пользователя (по соображениям безопасности конечно, а не от излишнего снобизма)). Вы можете достичь этого, создав метод `isPasswordLegal` и указав,

ограничение, что этот метод должен возвращать true:

- *YAML*

```
# src/Acme/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\Author:
    getters:
        passwordLegal:
            - "True": { message: "The password cannot match your first name" }
```

- *Annotations*

```
// src/Acme/BlogBundle/Entity/Author.php
use Symfony\Component\Validator\Constraints as Assert;

class Author
{
    /**
     * @Assert\True(message = "The password cannot match your first name")
     */
    public function isPasswordLegal()
    {
        // return true or false
    }
}
```

- *XML*

```
<!-- src/Acme/BlogBundle/Resources/config/validation.xml -->
<class name="Acme\BlogBundle\Entity\Author">
    <getter property="passwordLegal">
        <constraint name="True">
            <option name="message">The password cannot match your first name</option>
        </constraint>
    </getter>
</class>
```

- *PHP*

```
<?php
// src/Acme/BlogBundle/Entity/Author.php
use Symfony\Component\Validator\Mapping\ClassMetadata;
use Symfony\Component\Validator\Constraints\True;

class Author
{
    public static function loadValidatorMetadata(ClassMetadata $metadata)
    {
        $metadata->addGetterConstraint('passwordLegal', new True(array(
```

```
                'message' => 'The password cannot match your first name',
            )));
        }
    }
```

Теперь создайте метод `isPasswordLegal()` и реализуйте его логику:

```
public function isPasswordLegal()
{
    return ($this->firstName != $this->password);
}
```

Примечание: Особо внимательные читатели наверняка отметили, что в примере конфигурации опущен префикс геттера ("get" или "is"). Это позволяет вам легко переместить ограничение на поле класса с тем же именем в последствии (или же, наоборот, с поля на метод класса) без изменения логики валидации.

Классы

Некоторые ограничения применяются к целому классу во время валидации. Например ограничение `Callback` - это универсальное ограничение, которое применяется к классу целиком. Когда этот класс валидируется, вызываются методы указанные ограничением, что позволяет выполнять более детальную или же избирательную валидацию.

2.10.5 Валидационные группы

До сих пор вы могли добавлять ограничения к классу и узнавать, удовлетворяет ли класс всем указанным для него ограничениям или же нет. В некоторых случаях, однако, вам может потребоваться валидировать объект, используя лишь некоторые из определённых для него ограничений. Для того чтобы получить такую возможность, вы можете определить каждое ограничение в одну или более "валидационных групп" и после этого выполнять валидацию лишь для одной из этих групп.

Например, положим у вас есть класс `User`, который используется при регистрации пользователя и при обновлении его профайла впоследствии:

- *YAML*

```
# src/Acme/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\User:
    properties:
        email:
            - Email: { groups: [registration] }
        password:
```

```
            - NotBlank: { groups: [registration] }
            - MinLength: { limit: 7, groups: [registration] }
        city:
            - MinLength: 2
```

- *Annotations*

```
// src/Acme/BlogBundle/Entity/User.php
namespace Acme\BlogBundle\Entity;

use Symfony\Component\Security\Core\User\UserInterface;
use Symfony\Component\Validator\Constraints as Assert;

class User implements UserInterface
{
    /**
    * @Assert\Email(groups={"registration"})
    */
    private $email;

    /**
    * @Assert\NotBlank(groups={"registration"})
    * @Assert\MinLength(limit=7, groups={"registration"})
    */
    private $password;

    /**
    * @Assert\MinLength(2)
    */
    private $city;
}
```

- *XML*

```
<!-- src/Acme/BlogBundle/Resources/config/validation.xml -->
<class name="Acme\BlogBundle\Entity\User">
    <property name="email">
        <constraint name="Email">
            <option name="groups">
                <value>registration</value>
            </option>
        </constraint>
    </property>
    <property name="password">
        <constraint name="NotBlank">
            <option name="groups">
                <value>registration</value>
            </option>
```

```
            </constraint>
            <constraint name="MinLength">
                <option name="limit">7</option>
                <option name="groups">
                    <value>registration</value>
                </option>
            </constraint>
        </property>
        <property name="city">
            <constraint name="MinLength">7</constraint>
        </property>
    </class>
```

- *PHP*

```
<?php
// src/Acme/BlogBundle/Entity/User.php
namespace Acme\BlogBundle\Entity;

use Symfony\Component\Validator\Mapping\ClassMetadata;
use Symfony\Component\Validator\Constraints\Email;
use Symfony\Component\Validator\Constraints\NotBlank;
use Symfony\Component\Validator\Constraints\MinLength;

class User
{
    public static function loadValidatorMetadata(ClassMetadata $metadata)
    {
        $metadata->addPropertyConstraint('email', new Email(array(
            'groups' => array('registration')
        )));

        $metadata->addPropertyConstraint('password', new NotBlank(array(
            'groups' => array('registration')
        )));
        $metadata->addPropertyConstraint('password', new MinLength(array(
            'limit'  => 7,
            'groups' => array('registration')
        )));

        $metadata->addPropertyConstraint('city', new MinLength(3));
    }
}
```

При использовании такой конфигурации имеется две валидационные группы:

- `Default` - содержит ограничения, не включённые ни в одну из групп;
- `registration` - содержит ограничения для полей `email` и `password`.

Для того, чтобы явно указать валидатору группу, передайте одно или более наименований групп вторым аргументом в метод `validate()`:

```
$errors = $validator->validate($author, array('registration'));
```

Конечно же, как правило, вы работаете с валидацией косвенно через библиотеку Форм. Чтобы узнать как использовать группы в формах, смотрите раздел *Валидационные группы*.

2.10.6 Валидация простых значений и массивов

Ранее вы увидели как можно валидировать целые объекты. Но иногда, вам всего лишь нужно валидировать простое значение - например, проверить, является ли строка валидным email-адресом. Это также легко сделать. Внутри контроллера это будет выглядеть так:

```
<?php
// add this to the top of your class
use Symfony\Component\Validator\Constraints\Email;

public function addEmailAction($email)
{
    $emailConstraint = new Email();
    // все опции ограничения можно задать таким образом
    $emailConstraint->message = 'Invalid email address';

    // используем валидатор для проверки значения
    $errorList = $this->get('validator')->validateValue($email, $emailConstraint);

    if (count($errorList) == 0) {
        // это ВАЛИДНЫЙ адрес, делаем дело дальше
    } else {
        // это НЕ валидный адрес
        $errorMessage = $errorList[0]->getMessage()

        // делаем что-то с ошибкой
    }

    // ...
}
```

Вызывая метод `validateValue` в валидаторе, вы можете передать ему значение в виде параметра и объект ограничения, которое хотите проверить. Полный список доступных ограничений, а также полные имена классов для каждого ограничения, можно найти в `Справочнике по ограничениям`.

Метод `validateValule` возвращает объект класса `Symfony\Component\Validator\ConstraintViolati`

который, по сути, является массивом ошибок. Каждая ошибка в коллекции - это объект класса `Symfony\Component\Validator\ConstraintViolation`, который содержит сообщение об ошибке, которое можно получить, вызвав метод *getMessage*.

2.10.7 Заключение

Валидатор Symfony2 - это мощный инструмент, который используется для получения гарантий, что данные некоторого объекта "правильные". Сила валидации - в ограничениях, которые являются правилами, которые применяются к полям классов или же их "геттерам". И даже если вам приходится большей частью использовать фреймворк для валидации косвенно - совместно с формами, помните, что он может быть использован где угодно для валидации любого объекта.

2.10.8 Дополнительно в книге рецептов:

- `/cookbook/validation/custom_constraint`

2.11 Формы

Работа с формами - одна из наиболее типичных и проблемных задач для web-разработчика. Symfony2 включает компонент для работы с Формами, который облегчает работу с ними. В этой главе вы узнаете, как создавать сложные формы с нуля, познакомитесь с наиболее важными особенностями библиотеки форм.

Примечание: Компонент для работы с формами - это независимая библиотека, которая может быть использована вне проектов Symfony2. Подробности ищите по ссылке Symfony2 Form Component на ГитХабе.

2.11.1 Создание простой формы

Предположим, вы работаете над простым приложением - списком ToDo, которое будет отображать некоторые "задачи". Поскольку вашим пользователям будет необходимо создавать и редактировать задачи, вам потребуется создать форму. Но, прежде чем начать, давайте создадим базовый класс `Task`, который представляет и хранит данные для одной задачи:

```
<?php
// src/Acme/TaskBundle/Entity/Task.php
namespace Acme\TaskBundle\Entity;
```

```
class Task
{
    protected $task;

    protected $dueDate;

    public function getTask()
    {
        return $this->task;
    }
    public function setTask($task)
    {
        $this->task = $task;
    }

    public function getDueDate()
    {
        return $this->dueDate;
    }
    public function setDueDate(\DateTime $dueDate = null)
    {
        $this->dueDate = $dueDate;
    }
}
```

Примечание: Если вы будете брать код примеров один в один, вам, прежде всего необходимо создать пакет `AcmeTaskBundle`, выполнив следующую команду (и принимая все опции интерактивного генератора по умолчанию):

```
php app/console generate:bundle --namespace=Acme/TaskBundle
```

Этот класс представляет собой обычный PHP-объект и не имеет ничего общего с Symfony или какой-либо другой библиотекой. Это PHP-объект, который выполняет задачу непосредственно внутри *вашего* приложение (т.е. является представлением задачи в вашем приложении). Конечно же, к концу этой главы вы будете иметь возможность отправлять данные для экземпляра `Task` (посредством HTML-формы), валидировать её данные и сохранять в базу данных.

Создание формы

Теперь, когда вы создали класс `Task`, следующим шагом будет создание и отображение HTML-формы. В Symfony2 это выполняется посредством создания объекта формы и отображения его в шаблоне. Теперь, выполним необходимые действия в контроллере:

```php
<?php
// 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)
    {
        // создаём задачу и присваиваем ей некоторые начальные данные для примера
        $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(),
        ));
    }
}
```

Совет: Этот пример показывает, как создать вашу форму непосредственно в коде вашего контроллера. Позднее, в секции *Создание классов форм*, вы также узнаете, как создавать формы в отдельных классах, что является более предпочтительным вариантом и сделает ваши формы доступными для повторного использования.

Создание формы требует совсем немного кода, так как объекты форм в Symfony2 создаются при помощи конструктора форм - "form builder". Цель конструктора форм - облегчить насколько это возможно создание форм, выполняя всю тяжёлую работу.

В этом примере вы добавили два поля в вашу форму - `task` и `dueDate`, соответствующие полям `task` и `dueDate` класса `Task`. Вы также указали каждому полю их типы (например `text`, `date`), которые в числе прочих параметров, определяют - какой HTML таг будет отображен для этого поля в форме.

Symfony2 включает много встроенных типов, которые будут обсуждаться совсем скоро (см. *Встроенные типы полей*).

Отображение формы

Теперь, когда форма создана, следующим шагом будет её отображение. Отобразить форму можно, передав специальный объект "form view" в ваш шаблон (обратите внимание на конструкцию `$form->createView()` в контроллере выше) и использовать ряд функций-помощников в шаблоне:

- *Twig*

```
{# src/Acme/TaskBundle/Resources/views/Default/new.html.twig #}

<form action="{{ path('task_new') }}" method="post" {{ form_enctype(form) }}>
    {{ form_widget(form) }}

    <input type="submit" />
</form>
```

- *PHP*

```
<!-- src/Acme/TaskBundle/Resources/views/Default/new.html.php -->

<form action="<?php echo $view['router']->generate('task_new') ?>" method="post" <?php echo
    <?php echo $view['form']->widget($form) ?>

    <input type="submit" />
</form>
```

Примечание: В этом примере предполагается, что вы создали маршрут `task_new`, который указывает на контроллер `AcmeTaskBundle:Default:new`, который был создан ранее.

Вот и всё! Напечатав `form_widget(form)`, каждое поле формы будет отображено, так же как метки полей и ошибки (если они есть). Это очень просто, но не очень гибко (пока что). На практике вам, скорее всего, захочется отобразить каждое поле формы отдельно, чтобы иметь полный контроль над тем как форма выглядит. Вы узнаете, как сделать это в секции "*Отображение формы в шаблоне*".

Прежде чем двигаться дальше, обратите внимание на то, как было отображено поле `task`, содержащее значение поля `task` объекта `$task` ("Write a blog post"). Это - первая задача форм: получить данные от объекта и перевести их в формат, подходящий для их последующего отображения в HTML форме.

Совет: Система форм достаточно умна, чтобы получить доступ к значению защищённого (protected) поля `task` через методы `getTask()` и `setTask()` класса `Task`. Так как поле не публичное (public), оно должно иметь "геттер" и "сеттер" методы для того, чтобы компонент форм мог получить данные из этого поля и изменить их. Для булевых полей вы также можете использовать "is*" метод (например `isPublished()`) вместо `getPublished()`.

Обработка отправки форм

Второй обязанностью форм является перевод данных, отправленных пользователем, в свойства объекта. Для того чтобы это произошло, отправленные данные должны быть привязаны к форме. Добавьте в контроллер следующие строки:

```
<?php
// ...

public function newAction(Request $request)
{
    // создаём новый объект $task (без данных по умолчанию)
    $task = new Task();

    $form = $this->createFormBuilder($task)
        ->add('task', 'text')
        ->add('dueDate', 'date')
        ->getForm();

    if ($request->getMethod() == 'POST') {
        $form->bindRequest($request);

        if ($form->isValid()) {
            // выполняем прочие действие, например, сохраняем задачу в базе данных

            return $this->redirect($this->generateUrl('task_success'));
        }
    }

    // ...
}
```

Теперь, при отправке формы контроллер привязывает отправленные данные к фор-

ме, которая присваивает эти данные полям `task` и `dueDate` объекта `$task`. Эта задача выполняется методом `bindRequest()`.

Примечание: Как только вызывается метод `bindRequest()`, отправленные данные тут же присваиваются соответствующему объекту формы. Это происходит вне зависимости от того, валидны ли эти данные или нет.

Этот контроллер следует типичному сценарию по обработке форм и имеет три возможных пути:

1. При первичной загрузке страницы в браузер метод запроса будет `GET`, форма лишь создаётся и отображается;
2. Когда пользователь отправляет форму (т.е. метод будет уже `POST`) с неверными данными (вопросы валидации будут рассмотрены ниже, а пока просто предположим что данные не валидны), форма будет привязана к данным и отображена вместе со всеми ошибками валидации;
3. Когда пользователь отправляет форму с валидными данными, форма будет привязана к данным и у вас есть возможность для выполнения некоторых действий, используя объект `$task` (например сохранить его в базе данных) перед тем как перенаправить пользователя на другую страницу (например, “thank you” или “success”).

Примечание: Перенаправление пользователя после успешной отправки формы предотвращает повторную отправку этих же данных, если пользователь обновит страницу.

2.11.2 Валидация форм

В предыдущей секции вы узнали, что форма может быть отправлена с валидными или не валидными данными. В Symfony2 валидация применяется к объекту, лежащему в основе формы (например, `Task`). Другими словами, вопрос не в том, валидна ли форма, а валиден ли объект `$task`, после того как форма передала ему отправленные данные. Выполнив метод `$form->isValid()`, можно узнать валидны ли данные объекта `$task` или же нет.

Валидация выполняется посредством добавление набора правил (называемых ограничениями) к классу. Для того, чтобы увидеть валидацию в действии, добавьте ограничения для валидации того, что поле `task` не может быть пусто, а поле `dueDate` не может быть пусто и должно содержать объект `\DateTime`.

- *YAML*

```
# Acme/TaskBundle/Resources/config/validation.yml
Acme\TaskBundle\Entity\Task:
    properties:
        task:
            - NotBlank: ~
        dueDate:
            - NotBlank: ~
            - Type: \DateTime
```

- *Annotations*

```
// Acme/TaskBundle/Entity/Task.php
use Symfony\Component\Validator\Constraints as Assert;

class Task
{
    /**
     * @Assert\NotBlank()
     */
    public $task;

    /**
     * @Assert\NotBlank()
     * @Assert\Type("\DateTime")
     */
    protected $dueDate;
}
```

- *XML*

```
<!-- Acme/TaskBundle/Resources/config/validation.xml -->
<class name="Acme\TaskBundle\Entity\Task">
    <property name="task">
        <constraint name="NotBlank" />
    </property>
    <property name="dueDate">
        <constraint name="NotBlank" />
        <constraint name="Type">
            <value>\DateTime</value>
        </constraint>
    </property>
</class>
```

- *PHP*

```
<?php
// Acme/TaskBundle/Entity/Task.php
use Symfony\Component\Validator\Mapping\ClassMetadata;
```

```
use Symfony\Component\Validator\Constraints\NotBlank;
use Symfony\Component\Validator\Constraints\Type;

class Task
{
    // ...

    public static function loadValidatorMetadata(ClassMetadata $metadata)
    {
        $metadata->addPropertyConstraint('task', new NotBlank());

        $metadata->addPropertyConstraint('dueDate', new NotBlank());
        $metadata->addPropertyConstraint('dueDate', new Type('\DateTime'));
    }
}
```

Это всё! Если вы отправите форму с ошибочными значениями - вы увидите что соответствующие ошибки будут отображены в форме.

HTML5 Валидация

Начиная с HTML5, многие браузеры могут выполнять некоторые валидационные ограничения на стороне клиента, без отправки формы. Типичным ограничением является указание атрибута `required` для полей, которые будут обязательными. В браузерах, которые поддерживают HTML5, этот атрибут будет позволять отображать браузерное сообщение об ошибке, если пользователь попробует отправить форму с пустым соответствующим полем.
Генерированные формы полностью поддерживают эту возможность, добавляя соответствующие HTML-атрибуты, которые активируют HTML5 клиентскую валидацию. Тем не менее, валидация на стороне клиента может быть отключена путём добавления атрибута `novalidate` к тагу `form` или `formnovalidate` к тагу `submit`. Это бывает необходимо, когда вам нужно протестировать ваши серверные ограничения, но, к примеру, браузер не даёт отправить форму с пустыми полями.

Валидация - это важная функция в составе Symfony2, она описывается в *отдельной главе*.

Валидационные группы

Совет: Если вы не используете *валидационные группы*, вы можете пропустить эту секцию.

Если ваш объект использует возможности *валидационных групп*, вам нужно указать,

какие группы вы хотите использовать:

```
<?php
// ...

$form = $this->createFormBuilder($users, array(
    'validation_groups' => array('registration'),
))->add(...)
;
```

Если вы создаёте *классы форм* (хорошая практика), тогда вам нужно указать следующий код в метод `getDefaultOptions()`:

```
<?php
// ...

public function getDefaultOptions(array $options)
{
    return array(
        'validation_groups' => array('registration')
    );
}
```

В обоих этих примерах, для валидации объекта, для которого создана форма, будет использована *лишь* группа `registration`.

Groups based on Submitted Data

Добавлено в версии 2.1: The ability to specify a callback or Closure in `validation_groups` is new to version 2.1 Если вам требуется дополнительная логика для определения валидационных групп, например, на совновании данных, отправленных пользователем, вы можете установить значением опции validation_groups в массив с callback или замыкание (`Closure`).

```
<?php
public function getDefaultOptions(array $options)
{
    return array(
        'validation_groups' => array('Acme\\AcmeBundle\\Entity\\Client', 'determineValidationGro
    );
}
```

Этот код вызовет статический метод `determineValidationGroups()` класса Client с текущей формой в качестве аргумента, после того как данные будут привязаны (bind) к форме, но перед запуском процесса валидации. Вы также можете определить логику в замыкании `Closure`, например:

```
<?php
public function getDefaultOptions(array $options)
{
    return array(
        'validation_groups' => function(FormInterface $form) {
            $data = $form->getData();
            if (Entity\Client::TYPE_PERSON == $data->getType()) {
                return array('person')
            } else {
                return array('company');
            }
        },
    );
}
```

2.11.3 Встроенные типы полей

В состав Symfony входит большое число типов полей, которые покрывают все типичные поля и типы данных, с которыми вы столкнётесь:

Текстовые поля

- text
- textarea
- email
- integer
- money
- number
- password
- percent
- search
- url

Поля для выбора

- choice
- entity

- `country`
- `language`
- `locale`
- `timezone`

Поля для даты и времени

- *date*
- `datetime`
- `time`
- `birthday`

Прочие поля

- `checkbox`
- `file`
- `radio`

Группы полей

- *collection*
- `repeated`

Скрытые поля

- `hidden`
- `csrf`

Базовые поля

- `field`
- `form`

Вы также можете создавать свои собственные типы полей. Эта возможность подробно описывается в статье книги рецептов "`/cookbook/form/create_custom_field_type`".

Опции полей форм

Каждый тип поля имеет некоторое число опций, которые можно использовать для их настройки. Например, поле `dueDate` сейчас отображает 3 селектбокса. Тем не менее, *поле date* можно настроить таким образом, чтобы отображался один текстбокс (где пользователь сможет ввести дату в виде строки):

```
<?php
// ...
->add('dueDate', 'date', array('widget' => 'single_text'))
```

Все типы полей имеют много различных опций, которые могут быть для них указаны. Многие из этих опций - специфичны для конкретного типа поля и более подробную информацию о них можно получить из справочника.

Опция `required`

Наиболее типичной опцией является опция `required`, которая может быть указана для любого поля. По умолчанию, опция `required` установлена в `true`, что даёт возможность браузерам с поддержкой HTML5 использовать встроенную в них клиентскую валидацию, если поле остаётся пустым. Если вам этого не требуется, или же установите опцию `required` в `false` или же *отключите валидацию HTML5*.
Отметим также, что установка опции `required` в `true` **не влияет** на серверную валидацию. Другими словами, если пользователь отправляет пустое значение для этого поля (при помощи старого браузера или веб-сервиса) оно будет считаться валидным, если вы не используете ограничения `NotBlank` или `NotNull`.
Таким образом, опция `required` - хороша, но серверную валидацию использовать *необходимо всегда*.

2.11.4 Автоматическое определение типов полей

Теперь, когда вы добавили данные для валидации в класс `Task`, Symfony теперь много знает о ваших полях. Если вы позволите, Symfony может определять ("угадывать") тип вашего поля и устанавливать его автоматически. В этом примере, Symfony может определить по правилам валидации, что `task` является полем типа `text` и `dueDate` - полем типа `date`:

```
<?php
public function newAction()
```

```
{
    $task = new Task();

    $form = $this->createFormBuilder($task)
        ->add('task')
        ->add('dueDate', null, array('widget' => 'single_text'))
        ->getForm();
}
```

Автоматическое определение активируется, когда вы опускаете второй аргумент в методе `add()` (или если вы указываете null для него). Если вы передаёте массив опций в качестве третьего аргумента (как в случае `dueDate` выше), эти опции применяются к "угаданному" полю.

Осторожно: Если ваша форма использует особую группу валидации, определитель типов полей будет учитывать *все* ограничения при определении типов полей (включая ограничения, которые не являются частью используемых валидационных групп).

Автоматическое определение опций для полей

В дополнение к определению "типа" поля, Symfony также может попытаться определить значения опций для поля.

Совет: Когда эти опции будут установлены, поле будет отображено с использованием особых HTML атрибутов, которые позволяют выполнять HTML5 валидацию на стороне клиента (например `Assert\MaxLength`). И, поскольку вам нужно будет вручную добавлять правила валидации на стороне сервера, эти опции могут быть угаданы исходя из ограничений, которые вы будете использовать для неё.

- `required`: Опция `required` может быть определена исходя из правил валидации (т.е. если поле `NotBlank` или `NotNull`) или же на основании метаданных Doctrine (т.е. если поле `nullable`). Это может быть очень удобно, так как правила клиентской валидации автоматически соответствуют правилам серверной валидации.
- `min_length`: Если поле является одним из видов текстовых полей, опция `min_length` может быть угадана исходя из правил валидации (если используются ограничения `MinLength` или `Min`) или же из метаданных Doctrine (основываясь на длине поля).
- `max_length`: Аналогично `min_length` с той лишь разницей, что определяет максимальное значение длины поля.

Примечание: Эти опции могут быть определены автоматически, *только* если вы используете автоопределение полей (не указываете или передаёте `null` в качестве второго аргумента в метод `add()`).

Если вы хотите изменить значения, определённые автоматически, вы можете перезаписать их, передавая требуемые опции в массив опций:

```
->add('task', null, array('min_length' => 4))
```

2.11.5 Отображение формы в шаблоне

Ранее вы увидели, как форму целиком можно отобразить при помощи лишь одной строки кода. Конечно же, на практике вам потребуется большая гибкость при отображении:

- *Twig*

```
{# src/Acme/TaskBundle/Resources/views/Default/new.html.twig #}

<form action="{{ path('task_new') }}" method="post" {{ form_enctype(form) }}>
    {{ form_errors(form) }}

    {{ form_row(form.task) }}
    {{ form_row(form.dueDate) }}

    {{ form_rest(form) }}

    <input type="submit" />
</form>
```

- *PHP*

```
<!-- // src/Acme/TaskBundle/Resources/views/Default/newAction.html.php -->

<form action="<?php echo $view['router']->generate('task_new') ?>" method="post" <?php echo
    <?php echo $view['form']->errors($form) ?>

    <?php echo $view['form']->row($form['task']) ?>
    <?php echo $view['form']->row($form['dueDate']) ?>

    <?php echo $view['form']->rest($form) ?>

    <input type="submit" />
</form>
```

Давайте более подробно рассмотрим каждую часть:

- `form_enctype(form)` - если хоть одно поле формы является полем

для загрузки файла, эта функция отобразит необходимый атрибут `enctype="multipart/form-data"`;

- `form_errors(form)` - Отображает глобальные по отношению к форме целиком ошибки валидации (ошибки для полей отображаются после них);
- `form_row(form.dueDate)` - Отображает текстовую метку, ошибки и HTML-виджет для заданного поля (например для `dueDate`) внутри `div` элемента (по умолчанию);
- `form_rest(form)` - Отображает все остальные поля, которые ещё не были отображены. Как правило хорошая идея расположить вызов этого хелпера внизу каждой формы (на случай если вы забыли вывести какое-либо поле или же не хотите вручную отображать скрытые поля). Этот хелпер также удобен для активации автоматической защиты от *CSRF атак*.

Основная часть работы сделана при помощи хелпера `form_row`, который отображает метку, ошибки и виджет для каждого поля внутри тага `div`. В секции *Дизайн форм* вы узнаете, как можно настроить вывод `form_row` на различных уровнях.

Совет: Вы можете получить доступ к данным вашей формы при помощи `form.vars.value`:

- *Twig*

```
{{ form.vars.value.task }}
```

- *PHP*

```
<?php echo $view['form']->get('value')->getTask() ?>
```

Отображение каждого поля вручную

Хелпер `form_row` очень удобен, так как вы можете быстро отобразить каждое поле вашей формы (и разметка, используемая для каждой строки может быть настроена). Но, так как жизнь как правило сложнее, чем нам хотелось бы, вы можете также отобразить каждое поле вручную. В конечном итоге вы получите тоже самое что и с хелпером `form_row`:

- *Twig*

```
{{ form_errors(form) }}

<div>
    {{ form_label(form.task) }}
    {{ form_errors(form.task) }}
    {{ form_widget(form.task) }}
</div>
```

```
<div>
    {{ form_label(form.dueDate) }}
    {{ form_errors(form.dueDate) }}
    {{ form_widget(form.dueDate) }}
</div>

{{ form_rest(form) }}
```

- *PHP*

```
<?php echo $view['form']->errors($form) ?>

<div>
    <?php echo $view['form']->label($form['task']) ?>
    <?php echo $view['form']->errors($form['task']) ?>
    <?php echo $view['form']->widget($form['task']) ?>
</div>

<div>
    <?php echo $view['form']->label($form['dueDate']) ?>
    <?php echo $view['form']->errors($form['dueDate']) ?>
    <?php echo $view['form']->widget($form['dueDate']) ?>
</div>

<?php echo $view['form']->rest($form) ?>
```

Если автоматически созданная метка для поля вам не нравится, вы можете указать её явно:

- *Twig*

```
{{ form_label(form.task, 'Task Description') }}
```

- *PHP*

```
<?php echo $view['form']->label($form['task'], 'Task Description') ?>
```

Наконец, некоторые типы полей имеют дополнительные опции отображения, которые можно указывать виджету. Эти опции документированы для каждого такого типа, но общей для всех опцией является `attr`, которая позволяет вам модифицировать атрибуты элемента формы. Следующий пример добавит текстовому полю CSS класс `task_field`:

- *Twig*

```
{{ form_widget(form.task, { 'attr': {'class': 'task_field'} }) }}
```

- *PHP*

```
<?php echo $view['form']->widget($form['task'], array(
    'attr' => array('class' => 'task_field'),
)) ?>
```

Справочник по функциям Twig

Если вы используете Twig, полная справочная информация о функциях, используемых для отображения форм, доступна в *справочнике*. Ознакомьтесь с этой информацией, для того чтобы узнать больше о доступных хелперах и опциях, которые для них доступны.

2.11.6 Создание классов форм

Как вы уже видели ранее, форма может быть создана и использована непосредственно в контроллере. Тем не менее, лучшей практикой является создание формы в отдельном PHP-классе, который может быть использован повторно в любом месте вашего приложения. Создайте новый класс, который будет содержать логику создания формы `task`:

```
<?php
// src/Acme/TaskBundle/Form/Type/TaskType.php

namespace Acme\TaskBundle\Form\Type;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilder;

class TaskType extends AbstractType
{
    public function buildForm(FormBuilder $builder, array $options)
    {
        $builder->add('task');
        $builder->add('dueDate', null, array('widget' => 'single_text'));
    }

    public function getName()
    {
        return 'task';
    }
}
```

Этот новый класс содержит все необходимые указания для создания формы задачи (обратите внимание, что метод `getName()` должен возвращать уникальный идентификатор для данной формы). Теперь, вы можете использовать этот класс для быстрого создания объекта формы в контроллере:

```php
<?php
// src/Acme/TaskBundle/Controller/DefaultController.php

// добавьте use для класса формы в начале файла контроллера
use Acme\TaskBundle\Form\Type\TaskType;

public function newAction()
{
    $task = // ...
    $form = $this->createForm(new TaskType(), $task);

    // ...
}
```

Размещение логики формы в отдельном классе означает, что теперь форма может быть легко использована в другом месте приложения. Это наилучший способ для создания форм, но выбор конечно же за вами.

Настройка `data_class` для формы

Каждая форма должна знать имя класса, который будет содержать данные для неё (например, `Acme\TaskBundle\Entity\Task`). Как правило, эти данные определяются автоматически по объекту, который передаётся вторым параметром в метод `createForm` (т.е. `$task`). Позднее, когда вы займётесь встраиванием форм, полагаться на автоопределение уже будет нельзя. Таким образом, хоть и не всегда необходимо, но всё же желательно явно указывать опцию `data_class`, добавив следующие строки в класс формы:

```php
<?php
public function getDefaultOptions(array $options)
{
    return array(
        'data_class' => 'Acme\TaskBundle\Entity\Task',
    );
}
```

2.11.7 Формы и Doctrine

Цель любой формы - преобразование данных из объекта (в нашем случае `Task`) в HTML форму и наоборот - преобразование данных, отправленных пользователем, обратно в объект. По существу, тема по сохранению объекта `Task` в базе данных совершенно не относится теме, обсуждаемой в главе "Формы". Тем не менее, если вы сконфигурировали класс `Task` для работы с Doctrine (т.е. вы добавили *метаданные для отображения* (mapping metadata) для него), его сохранение после отправки формы можно выполнить

в случае, если форма валидна:

```
<?php
if ($form->isValid()) {
    $em = $this->getDoctrine()->getEntityManager();
    $em->persist($task);
    $em->flush();

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

Если, по каким-то причинам у вас нет изначального объекта `$task`, вы можете получить его из формы:

```
$task = $form->getData();
```

Больше информации по работе с базами данных вы можете получить в главе *Doctrine ORM*.

Самое главное, что требуется уяснить, когда форма и данные связываются, данные тут же передаются в объект, лежащий в основе формы. Если вы хотите сохранить эти данные, вам нужно просто сохранить объект (который уже содержит отправленные данные).

2.11.8 Встроенные формы

Зачастую, когда вы хотите создать форму, вам требуется добавлять в неё поля из различных объектов. Например, форма регистрации может содержать данные, относящиеся к объекту `User` и к нескольким объектам `Address`. К счастью, с использованием компонента форм сделать это легко и естественно.

Встраивание одного объекта

Предположим, что каждая задача `Task` соответствует некоторому объекту `Category`. Начнём конечно же с создания класса `Category`:

```
<?php
// src/Acme/TaskBundle/Entity/Category.php
namespace Acme\TaskBundle\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Category
{
    /**
     * @Assert\NotBlank()
```

```
     */
    public $name;
}
```

Затем создадим свойство `category` в классе `Task`:

```
<?php
// ...

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;
    }
}
```

Теперь ваше приложение нужно подправить с учётом новых требований. Создайте класс формы для изменения объекта `Category`:

```
<?php
// src/Acme/TaskBundle/Form/Type/CategoryType.php
namespace Acme\TaskBundle\Form\Type;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilder;

class CategoryType extends AbstractType
{
    public function buildForm(FormBuilder $builder, array $options)
    {
        $builder->add('name');
    }
```

```
    public function getDefaultOptions(array $options)
    {
        return array(
            'data_class' => 'Acme\TaskBundle\Entity\Category',
        );
    }

    public function getName()
    {
        return 'category';
    }
}
```

Конечно целью же является изменение `Category` для `Task` непосредственно из задачи. Для того чтобы выполнить это, добавьте поле `category` в форму `TaskType`, которое будет представлено экземпляром нового класса `CategoryType`:

```
<?php
public function buildForm(FormBuilder $builder, array $options)
{
    // ...

    $builder->add('category', new CategoryType());
}
```

Поля формы `CategoryType` теперь могут быть отображены прямо в форме `TaskType`. Отобразите поля `Category` тем же способом как и поля `Task`:

- *Twig*

```
{# ... #}

<h3>Category</h3>
<div class="category">
    {{ form_row(form.category.name) }}
</div>

{{ form_rest(form) }}
{# ... #}
```

- *PHP*

```
<!-- ... -->

<h3>Category</h3>
<div class="category">
    <?php echo $view['form']->row($form['category']['name']) ?>
</div>
```

```
<?php echo $view['form']->rest($form) ?>
<!-- ... -->
```

Когда пользователь отправляет форму, данные для полей `Category` будут использованы для создания экземпляра `Category`, который будет присвоен полю `category` объекта `Task`.

Объект `Category` доступен через метод `$task->getCategory()` и может быть сохранён в базу данных или использован где требуется.

Встраивание коллекций форм

Вы также можете встроить в вашу форму целую коллекцию форм (например форма `Category` с множеством саб-форм `Product`). Этого можно достичь при использовании поля `collection`.

Подробнее этот тип поля описан в книге рецептов "`/cookbook/form/form_collections`" и справочнике: *collection*.

2.11.9 Дизайн форм

Каждая часть отображения формы может быть настроена в соответствии с вашими требованиями. Вы можете изменить как отображается каждая строка формы, изменить разметку отображения ошибок и даже настроить как должен отображаться таг `textarea`. Вы ничем не ограничены и разные настройки могут быть использованы в разных местах.

Symfony использует шаблоны для отображения всех частей форм, таких как метки, таги, input таги, сообщения об ошибках и многое другое.

В Twig каждый такой фрагмент представлен блоком Twig. Для настройки любой части отображения формы вам просто надо заменить нужный блок.

В PHP каждый фрагмент формы отображается посредством индивидуального файла шаблона. Для настройки отображения любой части формы вам нужно заменить существующий шаблон новым.

Для того чтобы понять, как это работает, давайте настроим отображение фрагмента `form_row` и добавим атрибут class для элемента `div`, который содержит каждую строку. Для того чтобы выполнить это, создайте новый файл шаблона, который будет содержать новую разметку:

- *Twig*

  ```
  {# src/Acme/TaskBundle/Resources/views/Form/fields.html.twig #}

  {% block field_row %}
  ```

```
{% spaceless %}
    <div class="form_row">
        {{ form_label(form) }}
        {{ form_errors(form) }}
        {{ form_widget(form) }}
    </div>
{% endspaceless %}
{% endblock field_row %}
```

- *PHP*

```
<!-- src/Acme/TaskBundle/Resources/views/Form/field_row.html.php -->

<div class="form_row">
    <?php echo $view['form']->label($form, $label) ?>
    <?php  echo $view['form']->errors($form) ?>
    <?php  echo $view['form']->widget($form, $parameters) ?>
</div>
```

Фрагмент `field_row` используется при отображении большинства полей при помощи функции `form_row`. Для того, чтобы сообщить компоненту форм, чтобы он использовал новый фрагмент `field_row`, определённый выше, добавьте следующую строку в начале шаблона, отображающего форму:

- *Twig*

```
{# src/Acme/TaskBundle/Resources/views/Default/new.html.twig #}

{% form_theme form 'AcmeTaskBundle:Form:fields.html.twig' %}

<form ...>
```

- *PHP*

```
<!-- src/Acme/TaskBundle/Resources/views/Default/new.html.php -->

<?php $view['form']->setTheme($form, array('AcmeTaskBundle:Form')) ?>

<form ...>
```

Таг `form_theme` (в Twig) как бы "импортирует" фрагменты, определённые в указанном шаблоне и использует их при отображении формы. Другими словами, когда вызывается функция `form_row` ниже в этом шаблоне, она будет использовать блок `field_row` из вашей темы (вместо блока `field_row` по умолчанию используемого в Symfony).

Для того чтобы настроить любую часть формы, вам всего лишь нужно переопределить все необходимые фрагменты. О том, какие блоки или файлы могут быть переопределены, мы поговорим в следующей секции.

Дополнительную информацию о кастомизации форм ищите в книге рецептов: `/cookbook/form/form_customization`.

Именование фрагментов форм

В Symfony, каждая отображаемая часть формы - HTML элементы форм, ошибки, метки и т.д. - определены в базовой теме, которая представляет из себя набор блоков в Twig и набор шаблонов в PHP.

В Twig все блоки определены в одном файле (form_div_layout.html.twig), который располагается внутри Twig Bridge. В этом файле вы можете увидеть любой из блоков, необходимых для отображения любого стандартного поля.

В PHP каждый фрагмент расположен в отдельном файле. По умолчанию, они располагаются в директории *Resources/views/Form* в составе пакета framework (см. на GitHub).

Наименование каждого фрагмента следует одному базовому правилу и разбито на две части, разделённых подчерком (_). Несколько примеров:

- `field_row` - используется функцией `form_row` для отображения большинства полей;
- `textarea_widget` - используется функцией `form_widget` для отображения полей типа `textarea`;
- `field_errors` - используется функцией `form_errors` для отображения ошибок.

Каждый фрагмент подчиняется простому правилу: `type_part`. Часть `type` соответствует типу поля, которое будет отображено (например, `textarea`, `checkbox`, `date` и т.д.), часть `part` соответствует же тому, что именно будет отображаться (`label`, `widget`, `errors`, и т.д.). По умолчанию есть четыре возможных типов *parts*, которые отображаются:

`label`	(`field_label`)	отображает метку для поля
`widget`	(`field_widget`)	отображает HTML-представление для поля
`errors`	(`field_errors`)	отображает ошибки для поля
`row`	(`field_row`)	отображает цельную строку для поля (label, widget & errors)

Примечание: Есть также ещё три типа *parts* - `rows`, `rest`, и `enctype` - но заменять их вам вряд ли потребуется, так что и заботиться этом не стоит.

Зная тип поля (например `textarea`), а также какую часть вы хотите изменить (например, `widget`), вы можете составить имя фрагмента, который должен быть переопределён (например, `textarea_widget`).

Наследование фрагментов шаблона форм

В некоторых случаях фрагмент, который вам нужно настроить, не будет существовать. Например, в базовой теме нет фрагмента `textarea_errors`. Но как же отображаются ошибки для полей textarea?

Ответ на этот вопрос такой: отображаются они при помощи фрагмента `field_errors`. Когда Symfony отображает ошибки для textarea, он ищет фрагмент `textarea_errors`, прежде чем использовать стандартный фрагмент `field_errors`. Любой тип поля имеет *родительский* тип (для `textarea` это `field`) и Symfony использует фрагмент от родительского типа, если базовый фрагмент не существует.

Таким образом, чтобы переопределить фрагмент ошибок только для полей `textarea`, скопируйте фрагмент `field_errors`, переименуйте его в `textarea_errors` и измените его как вам требуется. Для того, чтобы изменить отображение ошибок для *всех* полей, скопируйте и измените сам фрагмент `field_errors`.

Совет: Родительские типы для всех типов полей можно узнать из справочника: *типы полей*.

Глобальная тема для форм

В примере выше вы использовали хелпер `form_theme` (для Twig), чтобы "импортировать" изменённые фрагменты форм *только* в одну форму. Вы также можете указать Symfony тему форм для всего проекта в целом.

Twig

Для того, чтобы автоматически подключить переопределённые блоки из ранее созданного шаблона `fields.html.twig`, измените ваш файл конфигурации следующим образом:

- *YAML*

```
# app/config/config.yml

twig:
    form:
        resources:
            - 'AcmeTaskBundle:Form:fields.html.twig'
    # ...
```

- *XML*

```
<!-- app/config/config.xml -->

<twig:config ...>
        <twig:form>
            <resource>AcmeTaskBundle:Form:fields.html.twig</resource>
        </twig:form>
        <!-- ... -->
</twig:config>
```

- *PHP*

```
<?php
// app/config/config.php

$container->loadFromExtension('twig', array(
    'form' => array('resources' => array(
        'AcmeTaskBundle:Form:fields.html.twig',
     ))
    // ...
));
```

Любой блок внутри шаблона `fields.html.twig` будет использован глобально в рамках проекта для определения формата отображения форм.

Настройка отображения форм в файле формы при использовании Twig

При использовании Twig, вы также можете изменить блок формы непосредственно внутри шаблона, где требуется изменение стиля отображения:

```
{% 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 %}
```

Таг `{% form_theme form _self %}` позволяет изменять блоки формы непосредственно внутри того шаблона, который требует изменений. Используйте этот метод для быстрой настройки отображения формы, если данное изменение нигде больше не потребуется.

PHP

Для того, чтобы автоматически подключить изменённые шаблоны из директории `Acme/TaskBundle/Resources/views/Form`, созданной ранее, для *всех* шаблонов, измените конфигурацию вашего приложения следующим образом:

- *YAML*

  ```
  # app/config/config.yml

  framework:
      templating:
          form:
              resources:
                  - 'AcmeTaskBundle:Form'
  # ...
  ```

- *XML*

  ```
  <!-- app/config/config.xml -->
  ```

```
<framework:config ...>
    <framework:templating>
        <framework:form>
            <resource>AcmeTaskBundle:Form</resource>
        </framework:form>
    </framework:templating>
    <!-- ... -->
</framework:config>
```

- *PHP*

```
<?php
// app/config/config.php

$container->loadFromExtension('framework', array(
    'templating' => array('form' =>
        array('resources' => array(
            'AcmeTaskBundle:Form',
     )))
    // ...
));
```

Все фрагменты, определённые в директории `Acme/TaskBundle/Resources/views/Form` теперь будут использованы во всём приложении для изменения стиля отображения форм.

2.11.10 Защита от CSRF атак

CSRF - или же Подделка межсайтовых запросов это вид атак, позволяющий злоумышленнику выполнять запросы от имени пользователей вашего приложения, которые они делать не собирались (например перевод средств на счёт хакера). К счастью, такие атаки можно предотвратить, используя CSRF токен в ваших формах.

Хорошие новости! Заключаются они в том, что Symfony по умолчанию добавляет и валидирует CSRF токен для вас. Это означает, что вы получаете защиту от CSRF атак не прилагая к этому никаких усилий. Фактически, все формы в этой главе были защищены от подобных атак.

Защита от CSRF атак работает за счёт добавления в формы скрытого поля, называемого по умолчанию `_token`, которое содержит значение, которое знаете только вы и пользователь вашего приложения. Это гарантирует, что пользователь - и никто более - отправил данные, которые пришли к вам. Symfony автоматически валидирует наличие и правильность этого токена.

Поле `_token` - это скрытое поле и оно автоматически отображается, если вы используете функцию `form_rest()` в вашем шаблоне, которая отображает все поля, которые ещё не были отображены в форме.

CSRF токен можно настроить уровне формы. Например:

```
<?php
class TaskType extends AbstractType
{
    // ...

    public function getDefaultOptions(array $options)
    {
        return array(
            'data_class'      => 'Acme\TaskBundle\Entity\Task',
            'csrf_protection' => true,
            'csrf_field_name' => '_token',
            // уникальный ключ для генерации секретного токена
            'intention'       => 'task_item',
        );
    }

    // ...
}
```

Для того, чтобы отключить CSRF защиту, установите опцию `csrf_protection` в `false`. Настройки также можно выполнить на уровне всего проекта. Дополнительную информацию можно найти в *справочнике по настройке форм*.

Примечание: Опция `intention` (`намерение`) не обязательна, но значительно увеличивает безопасность сгенерированного токена, делая его различным для всех форм.

2.11.11 Использование форм без класса

В большинстве случаев форма привязывается к объекту и поля формы получают и сохраняют данные в поля этого объекта. Это ровно то, с чем вы работали ранее в этой главе.

Тем не менее, вам возможно потребуется использовать форму без соответствующего класса и получать массив отправленных данных, а не объект. Этого просто достичь:

```
<?php
// удостоверьтесь, что вы добавили use для пространства имён Request:
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->getMethod() == 'POST') {
            $form->bindRequest($request);

            // data is an array with "name", "email", and "message" keys
            $data = $form->getData();
        }

    // ... render the form
}
```

По умолчанию, форма полагает, что вы хотите работать с массивами данных, а не с объектами. Есть два способа изменить это поведение и связать форму с объектом:

1. Передать объект при создании формы (первый аргумент `createFormBuilder`) или второй аргумент `createForm`);
2. Определить опцию `data_class` для вашей формы.

Если вы этого *не сделали*, тогда форма будет возвращать данные в виде массива. В этом примере, так как `$defaultData` не является объектом (и не установлена опция `data_class`), `$form->getData()` в конечном итоге вернёт массив.

Совет: Вы также можете получить доступ к значениям POST (в данном случае "name") напрямую через объект запроса, например так:

```
$this->get('request')->request->get('name');
```

Тем не менее, в большинстве случаев рекомендуется использовать метод getData(), так как он возвращает данные (как правило объект) после того как он был преобразован фреймворком форм.

Добавление валидации

А как же быть с валидацией? Обычно, когда вы используете вызов `$form->isValid()`, объект валидировался на основании ограничений, которые вы добавили в этот класс. Но когда класса нет, как добавить ограничения для данных из формы?

Ответом является настройка ограничений вручную и передача их в форму. Полностью этот подход описан в главе о *Валидации*, мы же рассмотрим лишь небольшой пример:

```
<?php
// импорт пространств имён
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')),
));

// создание формы без значений по умолчанию и с явным указанием ограничений для валидации
$form = $this->createFormBuilder(null, array(
    'validation_constraint' => $collectionConstraint,
))->add('email', 'email')
    // ...
;
```

Теперь, когда вы вызываете *$form->isValid()*, ограничения, указанные выше, выполняются для данных формы. Если вы используете класс формы, переопределите метод `getDefaultOptions`:

```
<?php
namespace Acme\TaskBundle\Form\Type;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilder;
use Symfony\Component\Validator\Constraints\Email;
use Symfony\Component\Validator\Constraints\MinLength;
use Symfony\Component\Validator\Constraints\Collection;

class ContactType extends AbstractType
{
    // ...

    public function getDefaultOptions(array $options)
    {
        $collectionConstraint = new Collection(array(
            'name' => new MinLength(5),
            'email' => new Email(array('message' => 'Invalid email address')),
        ));

        $options['validation_constraint'] = $collectionConstraint;
    }
}
```

Теперь вы можете создавать формы с валидацией, которые возвращают массив данных вместо объекта. В большинстве случаев же лучше - да и более правильно - привязывать

объекты к вашим формам. Но для простых форм вы можете этого и не делать.

2.11.12 Заключение

Теперь вы знаете всё необходимое для создания сложных форм для вашего приложения. При создании форм, не забывайте что первой целью формы является транслирование данных из объекта (`Task`) в HTML форму, чтобы пользователь мог модифицировать эти данные. Второй целью формы является получение отправленных пользователем данных и передача их обратно в объект.

Есть ещё много вещей, которые стоит узнать о прекрасном мире форм, таких как `загрузка файлов при помощи Doctrine`, или же как создание формы с динамически меняемым числом вложенных форм (например, список todo, где вы можете добавлять новые поля при помощи Javascript перед отправкой). Ищите ответы в книге рецептов. Также изучите *справочник по типам полей*, который включает примеры использования полей и их опций.

2.11.13 Читайте также в книге рецептов

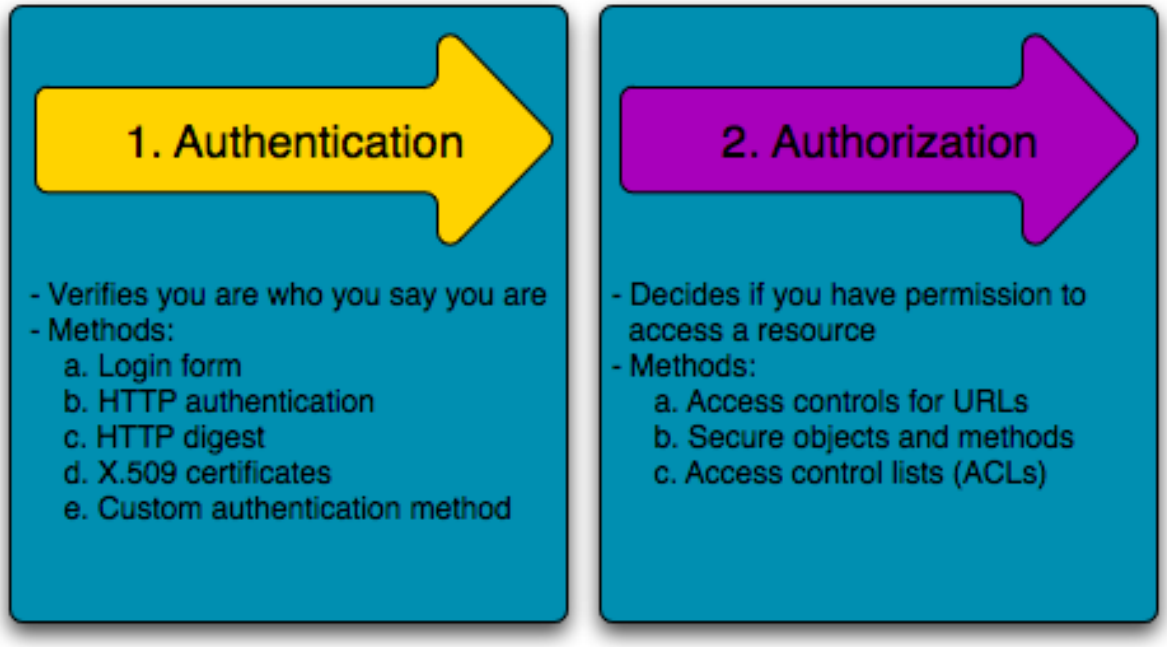
- `/cookbook/doctrine/file_uploads`
- `Работа с полем File`
- `Создание пользовательского поля`
- `/cookbook/form/form_customization`
- `/cookbook/form/dynamic_form_generation`
- `/cookbook/form/data_transformers`

2.12 Безопасность

Обеспечение безопасности (Security) - это двух шаговый процесс, целью которого является предотвращение доступа пользователя к ресурсам, получить которые он не имеет права.

В первую очередь, система безопасности идентифицирует пользователя, запрашивая у него ту или иную информацию. Этот процесс называется **аутентификацией** и означает, что система пытается понять, кто вы есть такой.

После того как система идентифицирует вас, следующим шагом требуется определить, имеете ли вы права на доступ к затребованному ресурсу. Эта часть процесса называется **авторизацией** и подразумевает проверку ваших привилегий на выполнение того или иного действия.



Поскольку лучший путь изучить что-либо - увидеть на примере, давайте углубимся в вопросы безопасности web-приложений.

Примечание: security component Symfony доступен как самостоятельная PHP-библиотека и может быть использован в любом PHP-проекте.

2.12.1 Простой пример: базовая HTTP аутентификация

Компонент безопасности может быть настроен при помощи конфигурации приложения. На самом деле, наиболее стандартные сценарии безопасности можно настроить непосредственно через конфигурацию. Следующая конфигурация подскажет Symfony, что нужно защитить любой URL, соответствующий шаблону `/admin/*`, и запрашивать пользовательские данные при помощи базовой HTTP-аутентификации (т.е. суровый олдскульный бокс username/password):

- *YAML*

```
# app/config/security.yml
security:
    firewalls:
        secured_area:
            pattern:    ^/
            anonymous: ~
            http_basic:
```

```
                realm: "Secured Demo Area"

    access_control:
        - { path: ^/admin, roles: ROLE_ADMIN }

    providers:
        in_memory:
            users:
                ryan:  { password: ryanpass, roles: 'ROLE_USER' }
                admin: { password: kitten, roles: 'ROLE_ADMIN' }

    encoders:
        Symfony\Component\Security\Core\User\User: plaintext
```

- *XML*

```
<!-- app/config/security.xml -->
<srv:container xmlns="http://symfony.com/schema/dic/security"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:srv="http://symfony.com/schema/dic/services"
    xsi:schemaLocation="http://symfony.com/schema/dic/services http://symfony.com/schema/di

    <config>
        <firewall name="secured_area" pattern="^/">
            <anonymous />
            <http-basic realm="Secured Demo Area" />
        </firewall>

        <access-control>
            <rule path="^/admin" role="ROLE_ADMIN" />
        </access-control>

        <provider name="in_memory">
            <user name="ryan" password="ryanpass" roles="ROLE_USER" />
            <user name="admin" password="kitten" roles="ROLE_ADMIN" />
        </provider>

        <encoder class="Symfony\Component\Security\Core\User\User" algorithm="plaintext" />
    </config>
</srv:container>
```

- *PHP*

```
<?php
// app/config/security.php
$container->loadFromExtension('security', array(
    'firewalls' => array(
        'secured_area' => array(
```

```
            'pattern' => '^/',
            'anonymous' => array(),
            'http_basic' => array(
                'realm' => 'Secured Demo Area',
            ),
        ),
    ),
    'access_control' => array(
        array('path' => '^/admin', 'role' => 'ROLE_ADMIN'),
    ),
    'providers' => array(
        'in_memory' => array(
            'users' => array(
                'ryan' => array('password' => 'ryanpass', 'roles' => 'ROLE_USER'),
                'admin' => array('password' => 'kitten', 'roles' => 'ROLE_ADMIN'),
            ),
        ),
    ),
    'encoders' => array(
        'Symfony\Component\Security\Core\User\User' => 'plaintext',
    ),
));
```

Совет: Стандартный дистрибутив Symfony выделяет настройку безопасности в отдельный файл (по умолчанию `app/config/security.yml`). Если вам не нужен отдельный конфигурационный файл для настройки безопасности, вы можете переместить его контент непосредственно в основной конфигурационный файл (по умолчанию `app/config/config.yml`).

В результате использования такой конфигурации вы получите полнофункциональную систему безопасности, которая выглядит следующим образом:

- Есть два пользователя системы (`ryan` and `admin`);
- Пользователи аутентифицируются при помощи базовой HTTP-аутентификации;
- Любой URL, соответствующий шаблону `/admin/*`, будет защищен, и лишь пользователь `admin` сможет попасть туда;
- Любой URL, *НЕ* соответствующий шаблону `/admin/*`, будет доступен любому пользователю без ввода логина/пароля.

Давайте взглянем на то, как работает безопасность и как каждая часть конфигурации вступает в игру.

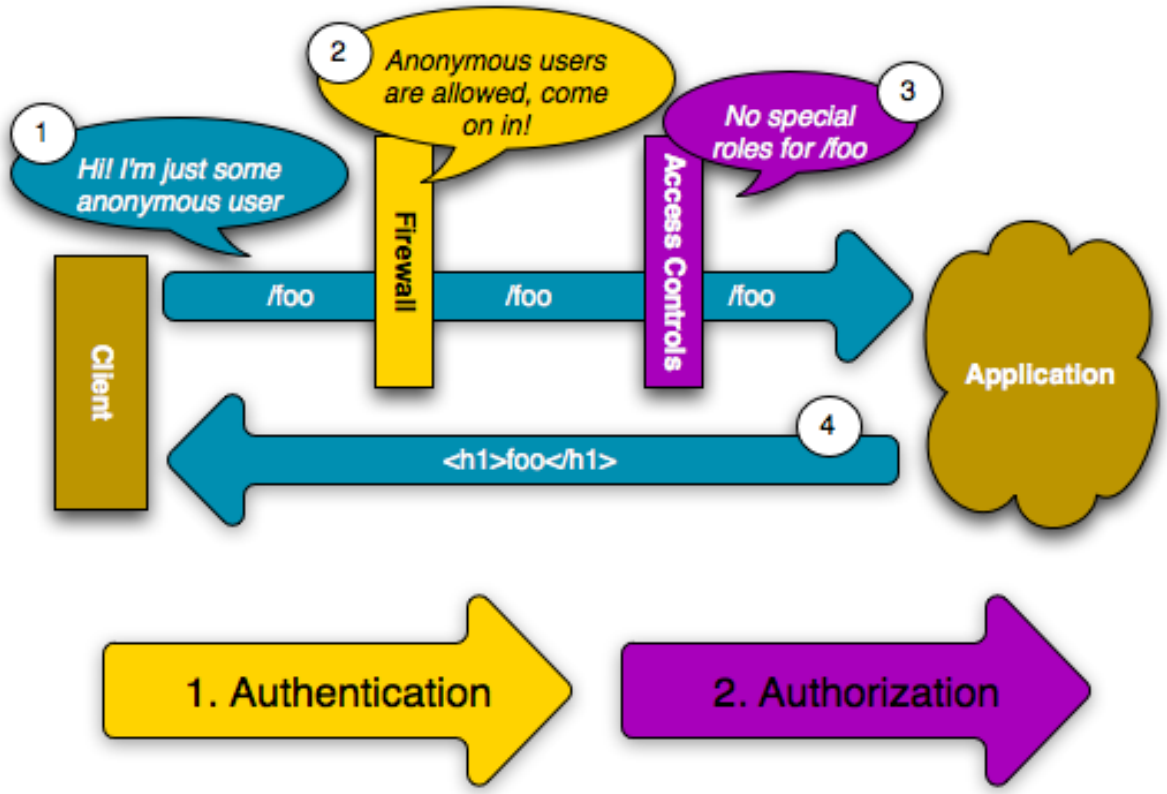
2.12.2 Как работает безопасность: Аутентификация и Авторизация

Система безопасности Symfony работает, определяя “личность” пользователя (аутентификация) и? затем, проверяя, имеет ли этот пользователь доступ к конкретному ресурсу или URL.

Брандмауэры (Аутентификация)

Когда пользователь выполняет запрос к URL, защищённому брандмауэром, активируется система безопасности. Работа брандмауэра заключается в определении того, требуется ли аутентификация пользователя и, если требуется, отправить обратно ответ, инициирующий процесс аутентификации.

Брандмауэр активируется, когда URL входящего запроса соответствует регулярному выражению `pattern`, которое было указано в конфигурации. В данном примере шаблон `pattern` (`^/`) будет соответствовать *любому* входящему запросу. То, что брандмауэр активируется, *не означает*, что HTTP аутентификация (бокс с логином/паролем) будет требоваться для каждого URL. К примеру, пользователь может получить доступ к `/foo` без запроса аутентификации:

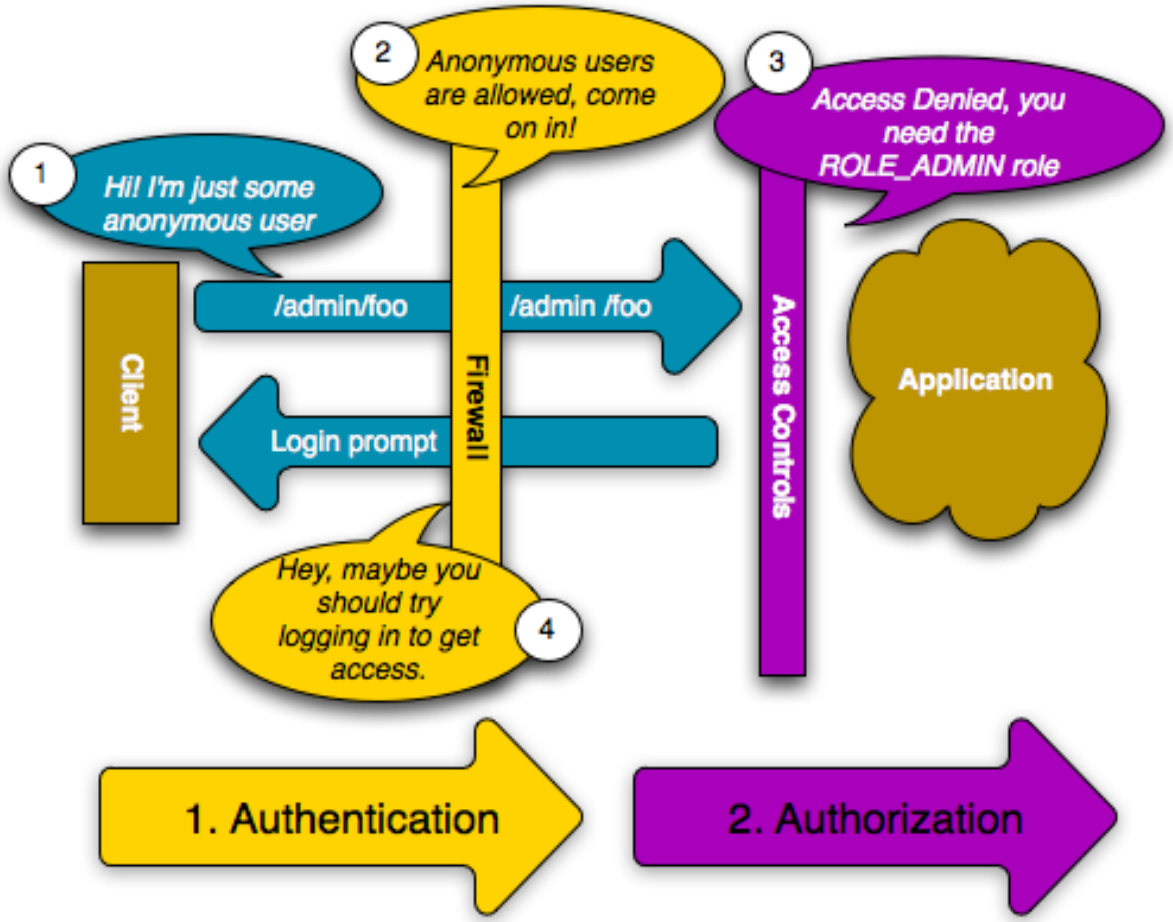


Это работает, так как брандмауэр позволяет доступ *анонимному пользователю* на основании параметра `anonymous` в настройках безопасности. Другими словами, брандмауэр не требует немедленной аутентификации. И, поскольку доступ к `/foo` не требует никакой особой роли (`role`) (это указано в секции `access_control`), запрос будет выполнен без аутентификации пользователя.

Если вы удалите ключ `anonymous`, брандмауэр будет *всегда* запрашивать у пользователя немедленной аутентификации.

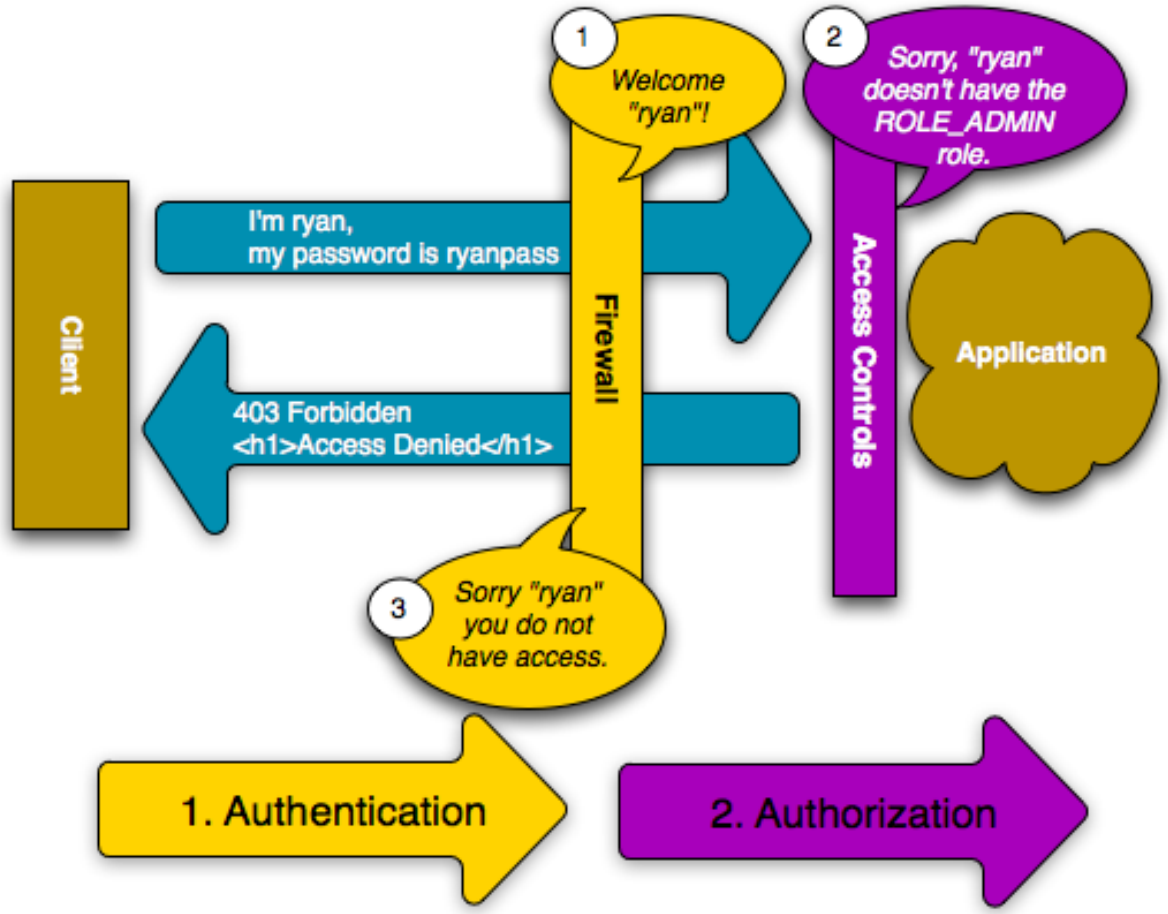
Контроль доступа (Авторизация)

Если пользователь запрашивает `/admin/foo`, процесс ведёт себя иным образом. Это обусловлено тем, что в секции `access_control` указано, что любой URL, соответствующий шаблону `^/admin` (т.е. `/admin` или всё прочее, что соответствует `/admin/*`) требует наличия у пользователя роли `ROLE_ADMIN`. Роли являются основой авторизации: пользователь может получить доступ к `/admin/foo` лишь тогда, когда у него есть роль `ROLE_ADMIN`.



Как и ранее, когда пользователь выполняет запрос, брандмауэр не требует идентификации пользователя. Тем не менее, как только контроль доступа отказывает пользователю в действии (так как анонимный пользователь не имеет роли `ROLE_ADMIN`), брандмауэр вступает в игру и инициирует процесс аутентификации. Процесс аутентификации зависит от механизма аутентификации, который вы используете. Например, если вы используете аутентификацию с использованием формы логина, пользователь будет перенаправлен на страницу логина. Если используется HTTP-аутентификация, пользователю будет направлен ответ с HTTP статус кодом 401 и в браузере будет отображён бокс с полями username и password.

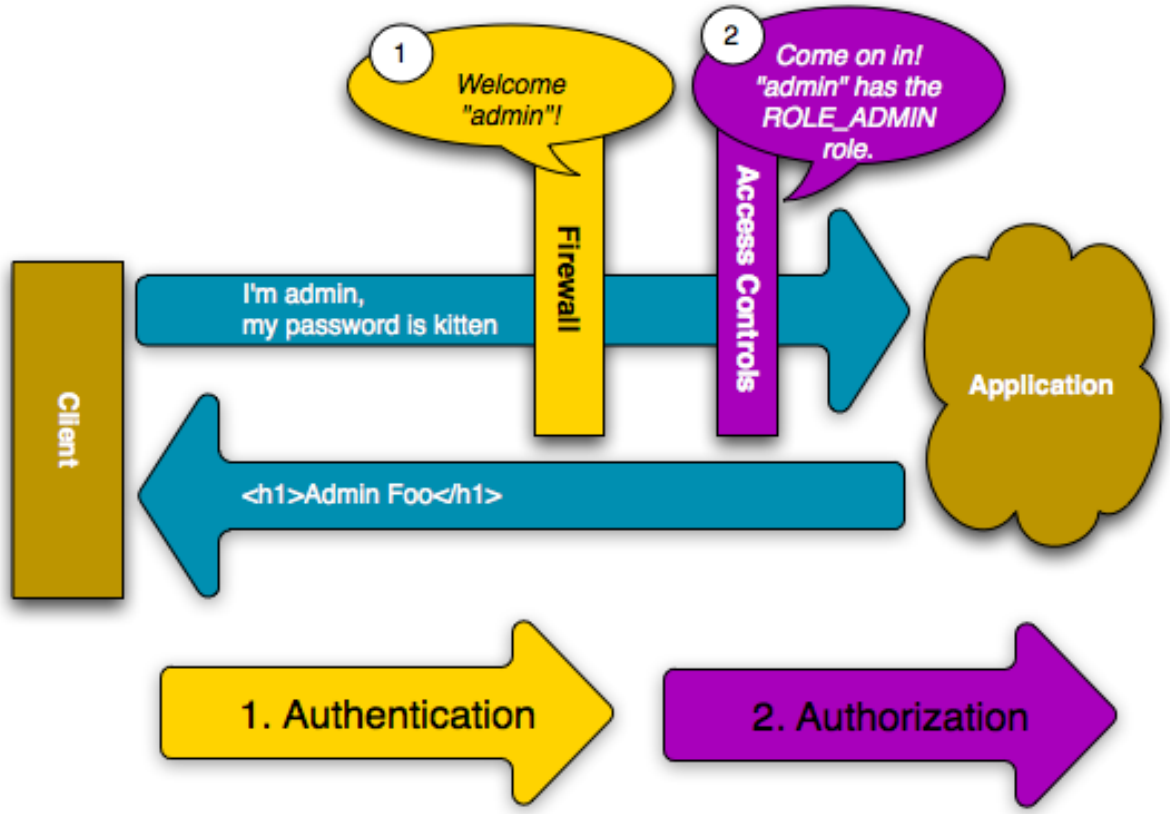
Пользователь теперь имеет возможность отправить свои данные обратно приложению. Если эти данные будут валидными, оригинальный запрос пользователя будет обработан.



В этом примере, пользователь `ryan` успешно проходит аутентификацию в брандмауэре, но, так как `ryan` не имеет роли `ROLE_ADMIN`, он по-прежнему не имеет доступа к `/admin/foo`. В конечном итоге, это означает, что пользователь увидит некое сообщение, о том, что ему отказано в доступе.

Совет: Когда Symfony запрещает доступ пользователю, ему отображается страница с ошибкой и возвращается HTTP статус код 403 (`Forbidden`). Вы можете изменить дизайн страницы с ошибкой 403, следуя руководству из книги рецептов *Error Pages*.

Наконец, если пользователь `admin` запрашивает `/admin/foo`, имеет место схожий процесс, за тем исключением, что система контроля доступа разрешит прохождение этого запроса:



Путь запроса, когда пользователь запрашивает защищённый ресурс, прямолинеен, но вместе с тем гибок. Как вы увидите позднее, аутентификация может быть выполнена различными способами, включая форму логина, сертификат X.509 или же посредством Twitter. Вне зависимости от метода аутентификации, запрос проходит следующий путь:

1. Пользователь запрашивает защищённый ресурс;
2. Приложение перенаправляет пользователя на форму логина (или её аналог);
3. Пользователь отправляет свои данные (например username/password);
4. Брандмауэр производит аутентификацию пользователя;
5. Аутентифицированный пользователь получает оригинальный запрос.

Примечание: Процесс аутентификации *целиком* зависит от типа аутентификации, который вы используете. Например, при использовании формы логина, пользователь отправляет свои данные по URL-адресу, который обрабатывает форму (например, `/login_check`) и после этого он перенаправляется на изначально запрошенный URL (например, `/admin/foo`). Но при использовании HTTP аутентификации пользователь отправляет свои данные не уходя с запрошенного URL (например, `/admin/foo`) и после этого страница отправляется пользователю без перенаправлений.

Эти особенности не должны вам причинять проблем, но лучше о них знать заранее.

Совет: В дальнейшем вы также узнаете, как можно защитить *любой* объект в Symfony2, включая отдельные контроллеры, объекты и даже PHP-методы.

2.12.3 Используем традиционную форму логина

До сих пор вы узнали, как защитить ваше приложение при помощи брандмауэра и, затем, ограничить доступ к отдельным разделам при помощи ролей. Используя HTTP аутентификацию, вы можете, не прилагая усилий, воспользоваться нативным окошком для аутентификации при помощи логина и пароля. Помимо этого, Symfony поддерживает много механизмов аутентификации "из коробки". Подробнее с ними вы можете ознакомиться в разделе справочной информации `Настройка параметров безопасности`.

В этой секции вы улучшите процесс аутентификации, дав пользователю возможность воспользоваться традиционной формой логина.

Во-первых, активируйте форму в брандмауэре:

- *YAML*

```yaml
# app/config/security.yml
security:
    firewalls:
        secured_area:
            pattern:    ^/
            anonymous: ~
            form_login:
                login_path:  /login
                check_path:  /login_check
```

- *XML*

```xml
<!-- app/config/security.xml -->
<srv:container xmlns="http://symfony.com/schema/dic/security"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:srv="http://symfony.com/schema/dic/services"
```

```
        xsi:schemaLocation="http://symfony.com/schema/dic/services http://symfony.com/schema/di

        <config>
            <firewall name="secured_area" pattern="^/">
                <anonymous />
                <form-login login_path="/login" check_path="/login_check" />
            </firewall>
        </config>
    </srv:container>
```

- *PHP*

```
<?php
// app/config/security.php
$container->loadFromExtension('security', array(
    'firewalls' => array(
        'secured_area' => array(
            'pattern' => '^/',
            'anonymous' => array(),
            'form_login' => array(
                'login_path' => '/login',
                'check_path' => '/login_check',
            ),
        ),
    ),
));
```

Совет: Если вы не хотите изменять значения `login_path` или `check_path` используемые по умолчанию, вы можете упростить конфигурацию:

- *YAML*

```
form_login: ~
```

- *XML*

```
<form-login />
```

- *PHP*

```
'form_login' => array(),
```

Теперь, когда система безопасности инициирует процесс аутентификации, она перенаправляет пользователя на форму логина (`/login` по умолчанию). Как эта форма будет выглядеть - это ваша забота. Сначала создайте два маршрута: один для отображения формы (т.е. `/login`) другой будет обрабатывать отправку формы логина (т.е. `/login_check`):

- *YAML*

```yaml
# app/config/routing.yml
login:
    pattern:   /login
    defaults:  { _controller: AcmeSecurityBundle:Security:login }
login_check:
    pattern:   /login_check
```

- *XML*

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

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

    <route id="login" pattern="/login">
        <default key="_controller">AcmeSecurityBundle:Security:login</default>
    </route>
    <route id="login_check" pattern="/login_check" />

</routes>
```

- *PHP*

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

$collection = new RouteCollection();
$collection->add('login', new Route('/login', array(
    '_controller' => 'AcmeDemoBundle:Security:login',
)));
$collection->add('login_check', new Route('/login_check', array()));

return $collection;
```

Примечание: Вам *не требуется* реализовывать контроллер для URL `/login_check`, так как брандмауэр будет автоматически перехватывать и обрабатывать формы, отправленные на этот URL. Не обязательно, но полезно, будет создание маршрута, который вы будете использовать для генерации URL отправки формы в шаблоне логина ниже.

Обратите внимание, что наименование маршрута `login` не обязательно. Действитель-

но же важным является URL этого маршрута (`/login`), соответствующий значению параметра `login_path`, так как на него система безопасности будет перенаправлять пользователя, которому нужно залогиниться.

Затем, создайте контроллер, который будет отображать форму логина:

```
<?php
// 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();

        // получить ошибки логина, если таковые имеются
        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(
            // имя, введённое пользователем в последний раз
            'last_username' => $session->get(SecurityContext::LAST_USERNAME),
            'error'         => $error,
        ));
    }
}
```

Не дайте этому коду запутать вас. Как вы увидите, когда пользователь отправляет форму, система безопасности автоматически обрабатывает её. Если юзер отправил неверные имя и пароль, этот контроллер получает ошибки от системы безопасности, чтобы вы могли их отобразить пользователю.

Другими словами, ваша работа заключается в отображении формы логина и ошибок, которые могут возникнуть, в то время как система безопасности берёт на себя заботу по проверке введённых имени и пароля и аутентификации пользователя.

Наконец, создадим шаблон формы:

- *Twig*

```
{# src/Acme/SecurityBundle/Resources/views/Security/login.html.twig #}
{% if error %}
    <div>{{ error.message }}</div>
{% endif %}

<form action="{{ path('login_check') }}" method="post">
    <label for="username">Username:</label>
    <input type="text" id="username" name="_username" value="{{ last_username }}" />

    <label for="password">Password:</label>
    <input type="password" id="password" name="_password" />

    {#
        Если вы хотите контролировать URL, на который перенаправить пользователя:
        <input type="hidden" name="_target_path" value="/account" />
    #}

    <input type="submit" name="login" />
</form>
```

- *PHP*

```
<?php // src/Acme/SecurityBundle/Resources/views/Security/login.html.php ?>
<?php if ($error): ?>
    <div><?php echo $error->getMessage() ?></div>
<?php endif; ?>

<form action="<?php echo $view['router']->generate('login_check') ?>" method="post">
    <label for="username">Username:</label>
    <input type="text" id="username" name="_username" value="<?php echo $last_username ?>"

    <label for="password">Password:</label>
    <input type="password" id="password" name="_password" />

    <!--
        Если вы хотите контролировать URL, на который перенаправить пользователя:
        <input type="hidden" name="_target_path" value="/account" />
    -->

    <input type="submit" name="login" />
</form>
```

Совет: Переменная `error`, передаваемая в шаблон, это экземпляр класса `Symfony\Component\Security\Core\Exception\AuthenticationException`. Этот объект может содержать дополнительную информацию - даже секретную - об ошибке аутентификации, так что используйте его с умом!

Форма имеет немного требований. Во-первых, отправляя форму на `/login_check` (маршрут `login_check`), система безопасности перехватит отправленную форму и обработает её автоматически. Во вторых, система безопасности ожидает, что отправленные поля будут называться `_username` и `_password` (эти наименования также можно *настроить*).

Вот и всё! Когда вы отправляете форму, система безопасности автоматически проверит данные пользователя и, либо выполнить его аутентификацию, либо отправить пользователя обратно на форму логина, где он увидит возникшие ошибки.

Давайте ещё раз взглянем на процесс целиком:

1. Пользователь пытается получить доступ к защищённому ресурсу;
2. Брандмауэр инициирует процесс аутентификации, перенаправляя пользователя на форму логина (`/login`);
3. Страница `/login` отображает форму логина при помощи маршрута и контроллера, созданных в этом примере;
4. Пользователь отправляет форму логина на URL `/login_check`;
5. Система безопасности перехватывает запрос, проверяет данные, отправленные пользователем, аутентифицирует пользователя, если данные верны или же возвращает пользователю страницу логина, если данные не верны.

По умолчанию, если данные пользователя верны, пользователь будет перенаправлен на ту же страницу, которую и запрашивал (например, `/admin/foo`). Если пользователь сразу открыл страницу логина, то он будет перенаправлен на главную страницу (`homepage`). Это поведение можно настроить, к примеру разрешить перенаправление пользователя на фиксированный URL.

Дополнительную информацию о том, как настраивается форма логина, смотрите статью в книге рецептов `/cookbook/security/form_login`.

Типичные ошибки

При настройке вашей формы логина, избегайте следующих типичных ошибок.

1. Создавайте корректные маршруты

Во-первых, удостоверьтесь, что вы корректно определили маршруты `/login` и `/login_check` и что они соответствуют конфигурационным параметрам `login_path` и `check_path`. Ошибки здесь будут вызывать перенаправление на страницу 404 вместо страницы логина или же отправленная форма не будет обрабатываться (вы будете видеть форму логина снова и снова).

2. Удостоверьтесь, что страница логина НЕ защищена

Также, удостоверьтесь, что страница логина *НЕ* требует никаких ролей для доступа к ней. Например, следующая конфигурация - которая требует роли `ROLE_ADMIN` для всех URL (включая URL `/login`), будет вызывать зацикливание перенаправлений:

- *YAML*

```
access_control:
    - { path: ^/, roles: ROLE_ADMIN }
```

- *XML*

```
<access-control>
    <rule path="^/" role="ROLE_ADMIN" />
</access-control>
```

- *PHP*

```
'access_control' => array(
    array('path' => '^/', 'role' => 'ROLE_ADMIN'),
),
```

Удаление контроля доступа для URL `/login` исправляет проблему:

- *YAML*

```
access_control:
    - { path: ^/login, roles: IS_AUTHENTICATED_ANONYMOUSLY }
    - { path: ^/, roles: ROLE_ADMIN }
```

- *XML*

```
<access-control>
    <rule path="^/login" role="IS_AUTHENTICATED_ANONYMOUSLY" />
    <rule path="^/" role="ROLE_ADMIN" />
</access-control>
```

- *PHP*

```
'access_control' => array(
    array('path' => '^/login', 'role' => 'IS_AUTHENTICATED_ANONYMOUSLY'),
    array('path' => '^/', 'role' => 'ROLE_ADMIN'),
),
```

Также, если ваш брандмауэр *не* разрешает доступ анонимным пользователям, вам необходимо создать особый брандмауэр, который позволяет анонимному пользователю получить доступ к странице логина:

- *YAML*

```
firewalls:
```

2.12.4 Авторизация

Первым шагом в обеспечении безопасности всегда является аутентификация: процесс идентификации пользователя. В Symfony аутентификацию можно выполнять различными способами, начиная с базовой HTTP аутентификации и формы логина и заканчивая Facebook.

После того как пользователь аутентифицирован, начинается процесс авторизации. Авторизация является стандартным путём определения имеет ли пользователь право доступа к какому-либо ресурсу (URL, объект модели, вызов метода...). В основе этого процесса лежит присвоение некоторых ролей каждому пользователю и после этого для различных ресурсов можно требовать наличия различных ролей.

Авторизация имеет две различные грани:

1. Пользователю назначен некоторый набор ролей;
2. Ресурс требует наличия некоторых ролей для получения доступа к нему.

В этой секции вы узнаете о том, как защитить различные ресурсы (например, URL, вызов метода и т.д.) при помощи различных ролей. Затем, вы узнаете о том, как создаются роли и как их можно присвоить пользователю.

Защищаем URL по шаблону

Наиболее простой и понятный способ защиты вашего приложения - защита некоторого набора URL по шаблону. Вы уже видели ранее, в первом примере этой главы, где все URL, что соответствовали регулярному выражению `^/admin`, требовали роли `ROLE_ADMIN`.

Вы можете определить столько URL, сколько вам нужно - каждый при помощи шаблона для регулярного выражения:

- *YAML*

```yaml
# app/config/config.yml
security:
    # ...
    access_control:
        - { path: ^/admin/users, roles: ROLE_SUPER_ADMIN }
        - { path: ^/admin, roles: ROLE_ADMIN }
```

- *XML*

```xml
<!-- app/config/config.xml -->
<config>
    <!-- ... -->
    <rule path="^/admin/users" role="ROLE_SUPER_ADMIN" />
```

```
        <rule path="^/admin" role="ROLE_ADMIN" />
    </config>
```

- *PHP*

```
<?php
// app/config/config.php
$container->loadFromExtension('security', array(
    // ...
    'access_control' => array(
        array('path' => '^/admin/users', 'role' => 'ROLE_SUPER_ADMIN'),
        array('path' => '^/admin', 'role' => 'ROLE_ADMIN'),
    ),
));
```

Совет: Добавление в начало пути символа ^ гарантирует, что этому шаблону будут соответствовать лишь URL, которые *начинаются* с него. Например, путь `/admin` (без ^ в начале) будет соответствовать как URL `/admin/foo`, так и URL `/foo/admin`.

Для каждого входящего запроса, Symfony2 пытается найти соответствующее правило контроля доступа (используется первое найденное правило). Если пользователь ещё не прошёл аутентификацию, инициируется процесс аутентификации (т.е. пользователю предоставляется возможность залогиниться в систему). Если же пользователь уже прошёл аутентификацию, но не имеет требуемой роли, будет брошено исключение `Symfony\Component\Security\Core\Exception\AccessDeniedException`, которое вы можете обработать и показать пользователю красивую страничку "access denied". Подробнее читайте в книге рецептов: *Как создать собственные страницы ошибок*

Так как Symfony использует первое найденное правило, URL вида `/admin/users/new` будет соответствовать первому правилу и требовать наличия роли `ROLE_SUPER_ADMIN`. Любой URL вида `/admin/blog` будет соответствовать второму правилу и требовать наличия роли `ROLE_ADMIN`.

Защита по IP

В жизни могут возникать различные ситуации, в которых вам будет необходимо ограничить доступ для некоего маршрута по IP. Это особенно важно в случае использования *Edge Side Includes* (ESI), которые, например, используют маршрут под названием "_internal". Когда используются ESI, маршрут _internal необходим кэширующему шлюзу для подключения различных опций кэширования субсекций внутри указанной страницы. Этот маршрут по умолчанию использует префикс ^/_internal в Symfony Standard Edition (предполагается также что вы раскомментировали эти строки в файле маршрутов).

Ниже приводится пример того, как вы можете защитить этот маршрут от доступа извне:

- *YAML*

```
# app/config/security.yml
security:
    # ...
    access_control:
        - { path: ^/_internal, roles: IS_AUTHENTICATED_ANONYMOUSLY, ip: 127.0.0.1 }
```

- *XML*

```
<access-control>
    <rule path="^/_internal" role="IS_AUTHENTICATED_ANONYMOUSLY" ip="127.0.0.1" />
</access-control>
```

- *PHP*

```
'access_control' => array(
    array('path' => '^/_internal', 'role' => 'IS_AUTHENTICATED_ANONYMOUSLY', 'ip' => '127.0
),
```

Использование защищённого канала

Как и защита на основании IP, требование использования SSL добавляется очень просто:

- *YAML*

```
# app/config/security.yml
security:
    # ...
    access_control:
        - { path: ^/cart/checkout, roles: IS_AUTHENTICATED_ANONYMOUSLY, requires_channel: h
```

- *XML*

```
<access-control>
    <rule path="^/cart/checkout" role="IS_AUTHENTICATED_ANONYMOUSLY" requires_channel="http
</access-control>
```

- *PHP*

```
'access_control' => array(
    array('path' => '^/cart/checkout', 'role' => 'IS_AUTHENTICATED_ANONYMOUSLY', 'requires_
),
```

Защита Контроллера

Защищать ваше приложение на основании шаблонов URL легко, но в некоторых случаях может быть не слишком удобным. При необходимости, вы также можете легко форсировать авторизацию внутри контроллера:

```
<?php
use Symfony\Component\Security\Core\Exception\AccessDeniedException;
// ...

public function helloAction($name)
{
    if (false === $this->get('security.context')->isGranted('ROLE_ADMIN')) {
        throw new AccessDeniedException();
    }

    // ...
}
```

Вы также можете использовать опциональный пакет `JMSSecurityExtraBundle`, который поможет вам защитить контроллер с использованием аннотаций:

```
<?php
use JMS\SecurityExtraBundle\Annotation\Secure;

/**
 * @Secure(roles="ROLE_ADMIN")
 */
public function helloAction($name)
{
    // ...
}
```

Дополнительную информацию об этом пакете вы можете получить из документации JMSSecurityExtraBundle. Если вы используете дистрибутив Symfony Standard Edition, этот пакет уже доступен вам по умолчанию. В противном случае вам необходимо загрузить и установить его.

Защита прочих сервисов

Фактически, всё что угодно в Symfony может быть защищено при помощи стратегии, описанной в предыдущей секции. Например, предположим у вас есть сервис (т.е. PHP класс), который отсылает email-сообщения от одного пользователя другому. Вы можете ограничить использование этого класса - неважно где он будет использован - для пользователей с определённой ролью.

Подробнее о том как вы можете использовать компонент безопасности для защиты

различных сервисов и методов в вашем приложении, смотрите статью в книге рецептов: `/cookbook/security/securing_services`.

Списки контроля доступа (ACL): Защита отдельных объектов базы данных

Представьте, что вы проектируете блог, где пользователи могут создавать комментарии к вашим постам. Теперь вы хотите, чтобы пользователь имел возможность редактировать его собственный комментарий, но не мог редактировать комментарии других пользователей. Также, в качестве администратора, вы хотите иметь возможность редактировать комментарии *всех* пользователей.

Компонент безопасности содержит опциональную систему "списков контроля доступа" (ACL), которую вы можете использовать при необходимости контроля доступа к отдельным экземплярам объектов в вашей системе. *Без* использования ACL, вы можете защитить свою систему таким образом, что лишь некоторые пользователи смогут иметь возможность редактирования комментариев. Но с помощью ACL, вы можете ограничить ли разрешить доступ к каждому конкретному комментарию.

Подробнее читайте в книге рецептов: `/cookbook/security/acl`.

2.12.5 Пользователи

В предыдущей секции вы узнали как можно защитить различные ресурсы, требуя для них наличия одной или нескольких *ролей*. В этой секции вы узнаете о другой грани авторизации: пользователях.

Откуда берутся пользователи? (*Провайдеры Пользователей*)

Во время аутентификации, пользователь отправляет некоторые данные (как правило имя и пароль). Работа системы аутентификации заключается в том, чтобы проверить эти данные на некотором наборе пользователей. Откуда же берутся эти пользователи?

В Symfony2 пользователи могут появляться отовсюду - из файла конфигурации, базы данных, веб сервиса или откуда вашей душе угодно будет. Всё, что предоставляет одного или более пользователей системе аутентификации называется "провайдером пользователя" (user provider). Symfony2 поставляется с двумя, наиболее простыми провайдерами: один из них загружает пользователей из конфигурационного файла, другой загружает пользователей из таблицы в базе данных.

Определение пользователей в файле конфигурации

Наиболее простой способ создать пользователей - определить их прямо в файле конфигурации. Фактически вы уже видели этот способ ранее в одном из примеров этой

главы.

- *YAML*

```yaml
# app/config/config.yml
security:
    # ...
    providers:
        default_provider:
            users:
                ryan:  { password: ryanpass, roles: 'ROLE_USER' }
                admin: { password: kitten, roles: 'ROLE_ADMIN' }
```

- *XML*

```xml
<!-- app/config/config.xml -->
<config>
    <!-- ... -->
    <provider name="default_provider">
        <user name="ryan" password="ryanpass" roles="ROLE_USER" />
        <user name="admin" password="kitten" roles="ROLE_ADMIN" />
    </provider>
</config>
```

- *PHP*

```php
<?php
// app/config/config.php
$container->loadFromExtension('security', array(
    // ...
    'providers' => array(
        'default_provider' => array(
            'users' => array(
                'ryan' => array('password' => 'ryanpass', 'roles' => 'ROLE_USER'),
                'admin' => array('password' => 'kitten', 'roles' => 'ROLE_ADMIN'),
            ),
        ),
    ),
));
```

Такой провайдер называется провайдером "в памяти" (in-memory), так как пользователи не сохранены где-либо в базе данных. В итоге предоставляется объект класса `Symfony\Component\Security\Core\User\User`.

Совет: Любой провайдер пользователей может загружать пользователей непосредственно из конфигурации, если для него указан параметр `users` и определены пользователи.

Осторожно: Если имя вашего пользователя полностью цифровое (например, 77) или содержит тире (например, `user-name`), вы должны использовать альтернативный синтаксис при создании пользователей в YAML файле:

```
users:
    - { name: 77, password: pass, roles: 'ROLE_USER' }
    - { name: user-name, password: pass, roles: 'ROLE_USER' }
```

Для небольших сайтов этот метод быстр и прост в настройке. Для более сложных систем вы вероятно захотите загружать пользователей из базы данных.

Загрузка пользователей из базы данных

Если вы хотите загружать пользователей из базы данных при помощи Doctrine ORM, вы можете этого легко достичь, создав класс `User` и настроив провайдер `entity`.

При таком подходе вы сначала создаёте свой собственный класс `User`, который будет сохраняться в базе данных:

```
<?php
// 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;

    // ...
}
```

Что же качается системы безопасности, единственным её требованием к вашему классу пользователя является имплементация им интерфейса `Symfony\Component\Security\Core\User\UserInterface`. Это означает, что ваша концепция пользователя может быть какой угодно, коль скоро класс имплементирует этот интерфейс.

Примечание: Объект пользователя будет сериализован и сохранён в сессии во время обработки запроса, поэтому рекомендуется также имплементировать интерфейс Serializable для вашего пользователя. Это особенно важно, если ваш класс `User` имеет родителя с приватными свойствами.

Далее, настроим провайдер `entity` и укажем для него класс `User`:

- *YAML*

```yaml
# app/config/security.yml
security:
    providers:
        main:
            entity: { class: Acme\UserBundle\Entity\User, property: username }
```

- *XML*

```xml
<!-- app/config/security.xml -->
<config>
    <provider name="main">
        <entity class="Acme\UserBundle\Entity\User" property="username" />
    </provider>
</config>
```

- *PHP*

```php
<?php
// app/config/security.php
$container->loadFromExtension('security', array(
    'providers' => array(
        'main' => array(
            'entity' => array('class' => 'Acme\UserBundle\Entity\User', 'property' => 'user
        ),
    ),
));
```

Добавив этот новый провайдер, система аутентификации будет пытаться загрузить объект `User` из базы данных, используя его поле `username`.

Примечание: Этот пример предназначен чтобы показать вам основную идею провайдера `entity`. Полный рабочий пример приводится в книге рецептов: `/cookbook/security/entity_provider`.

Больше информации о создании вашего собственного провайдера (например, если вам нужно загружать пользователей из веб-сервиса), смотрите статью `/cookbook/security/custom_provider`.

Шифрование пароля пользователя

Ранее, для упрощения, все примеры хранили пароли пользователей в виде текста (вне зависимости от того где эти пользователи были определены - в файле настроек или в базе данных). Конечно, в настоящем приложении вы захотите шифровать пароли пользователей из соображений безопасности. Этого легко достичь, связав ваш класс User с одним из нескольких встроенных "процедур шифрования". Например, при хранении ваших пользователей в памяти, чтобы скрывать их пароли при помощи функции `sha1`, выполните следующие настройки:

- *YAML*

```
# app/config/config.yml
security:
    # ...
    providers:
        in_memory:
            users:
                ryan:  { password: bb87a29949f3a1ee0559f8a57357487151281386, roles: 'ROLE_U
                admin: { password: 74913f5cd5f61ec0bcfdb775414c2fb3d161b620, roles: 'ROLE_A

    encoders:
        Symfony\Component\Security\Core\User\User:
            algorithm:   sha1
            iterations: 1
            encode_as_base64: false
```

- *XML*

```
<!-- app/config/config.xml -->
<config>
    <!-- ... -->
    <provider name="in_memory">
        <user name="ryan" password="bb87a29949f3a1ee0559f8a57357487151281386" roles="ROLE_U
        <user name="admin" password="74913f5cd5f61ec0bcfdb775414c2fb3d161b620" roles="ROLE_
    </provider>

    <encoder class="Symfony\Component\Security\Core\User\User" algorithm="sha1" iterations=
</config>
```

- *PHP*

```
<?php
// app/config/config.php
$container->loadFromExtension('security', array(
    // ...
    'providers' => array(
        'in_memory' => array(
```

```
            'users' => array(
                'ryan' => array('password' => 'bb87a29949f3a1ee0559f8a57357487151281386', '
                'admin' => array('password' => '74913f5cd5f61ec0bcfdb775414c2fb3d161b620',
            ),
        ),
    ),
    'encoders' => array(
        'Symfony\Component\Security\Core\User\User' => array(
            'algorithm'         => 'sha1',
            'iterations'        => 1,
            'encode_as_base64'  => false,
        ),
    ),
));
```

Присвоив параметру `iterations` значение 1 и параметру `encode_as_base64` - false, пароль будет просто прогоняться один раз через алгоритм шифрования `sha1` без дополнительного шифрования. Теперь вы можете вычислить хэш пароля програмно (`hash('sha1', 'ryanpass')`) или же при помощи онлайн-инструментов типа functions-online.com.

Если вы создаёте ваших пользователей динамически (и храните их в базе данных), вы можете использовать более сложные алгоритмы шифрования, а затем передавать оригинал пароля объекту-шифровальщику для хеширования паролей. Например, предположим что ваш объект User - это экземпляр класса `Acme\UserBundle\Entity\User` (как в примере выше). Сначала настройте шифрование для этого класса пользователя:

- *YAML*

```
# app/config/config.yml
security:
    # ...

    encoders:
        Acme\UserBundle\Entity\User: sha512
```

- *XML*

```
<!-- app/config/config.xml -->
<config>
    <!-- ... -->

    <encoder class="Acme\UserBundle\Entity\User" algorithm="sha512" />
</config>
```

- *PHP*

```
<?php
// app/config/config.php
$container->loadFromExtension('security', array(
    // ...

    'encoders' => array(
        'Acme\UserBundle\Entity\User' => 'sha512',
    ),
));
```

В этом случае вы используете более стойкий алгоритм `sha512`. Также, поскольку вы просто указали алгоритм шифрования в виде строки (`sha512`), система будет по умолчанию хэшировать ваш пароль 5000 раз подряд и затем шифровать его в base64. Другими словами, пароль будет многократно зашифрован и пароль не сможет быть декодирован (т.е. будет невозможно определить оригинал пароля по его хэшу).

Если у вас предусмотрена некая регистрация для пользователей, вам потребуется определить хэш пароля, чтобы присвоить его пользователю. Вне зависимости от алгоритма шифрования, который вы настроили для объекта пользователя, в контроллере получить хэш пароля можно следующим образом:

```
<?php
// ...

$factory = $this->get('security.encoder_factory');
$user = new Acme\UserBundle\Entity\User();

$encoder = $factory->getEncoder($user);
$password = $encoder->encodePassword('ryanpass', $user->getSalt());
$user->setPassword($password);
```

Получение объекта пользователя

После аутентификации, объект `User` для текущего юзера можно получить через сервис `security.context`. В контроллере это будет выглядеть следующим образом:

```
<?php
public function indexAction()
{
    $user = $this->get('security.context')->getToken()->getUser();
}
```

В контроллере можно использвать шорткат:

```
<?php
public function indexAction()
{
```

```
    $user = $this->getUser();
}
```

Примечание: Анонимные пользователи технически считаются также аутентифицированными, т.е. метод `isAuthenticated()` анонимного пользователя будет возвращать `true`. Для того, чтобы действительно убедиться, что ваш пользователь прошёл аутентификацию, необходимо проверить наличие роли `IS_AUTHENTICATED_FULLY`.

Использование нескольких провайдеров пользователей

Любой механизм аутентификации (HTTP аутентификация, форма логина и т.п.) использует только один провайдер и будет по умолчанию использовать первый указанный. Но что, если вы хотите указать несколько пользователей при помощи конфигурации и остальных пользователей сохранять в базу данных? Можно создать новый chain-провайдер, который позволит добиться этого:

- *YAML*

```yaml
# app/config/security.yml
security:
    providers:
        chain_provider:
            providers: [in_memory, user_db]
        in_memory:
            users:
                foo: { password: test }
        user_db:
            entity: { class: Acme\UserBundle\Entity\User, property: username }
```

- *XML*

```xml
<!-- app/config/config.xml -->
<config>
    <provider name="chain_provider">
        <provider>in_memory</provider>
        <provider>user_db</provider>
    </provider>
    <provider name="in_memory">
        <user name="foo" password="test" />
    </provider>
    <provider name="user_db">
        <entity class="Acme\UserBundle\Entity\User" property="username" />
    </provider>
</config>
```

- *PHP*

```
<?php
// app/config/config.php
$container->loadFromExtension('security', array(
    'providers' => array(
        'chain_provider' => array(
            'providers' => array('in_memory', 'user_db'),
        ),
        'in_memory' => array(
            'users' => array(
                'foo' => array('password' => 'test'),
            ),
        ),
        'user_db' => array(
            'entity' => array('class' => 'Acme\UserBundle\Entity\User', 'property' => 'user
        ),
    ),
));
```

Теперь, любой механизм аутентификации будет использовать `chain_provider`, так как он указан первым. В свою очередь, `chain_provider` будет пытаться получить пользователя как из провайдера `in_memory`, так и из `user_db`.

Совет: Если вам не требуется разделять пользователей `in_memory` от пользователей `user_db`, вы можете достигнуть того же эффекта ещё быстрее, скомбинировав эти два ресурса в один провайдер:

- *YAML*

```
# app/config/security.yml
security:
    providers:
        main_provider:
            users:
                foo: { password: test }
            entity: { class: Acme\UserBundle\Entity\User, property: username }
```

- *XML*

```
<!-- app/config/config.xml -->
<config>
    <provider name=="main_provider">
        <user name="foo" password="test" />
        <entity class="Acme\UserBundle\Entity\User" property="username" />
    </provider>
</config>
```

- *PHP*

```
<?php
// app/config/config.php
$container->loadFromExtension('security', array(
    'providers' => array(
        'main_provider' => array(
            'users' => array(
                'foo' => array('password' => 'test'),
            ),
            'entity' => array('class' => 'Acme\UserBundle\Entity\User', 'property' => 'user
        ),
    ),
));
```

Вы также можете настроить брандмауэр или индивидуальный механизм аутентификации на использование конкретного провайдера. Напоминаем, если провайдер явно не указан, будет использоваться первый по списку:

- *YAML*

```
# app/config/config.yml
security:
    firewalls:
        secured_area:
            # ...
            provider: user_db
            http_basic:
                realm: "Secured Demo Area"
                provider: in_memory
            form_login: ~
```

- *XML*

```
<!-- app/config/config.xml -->
<config>
    <firewall name="secured_area" pattern="^/" provider="user_db">
        <!-- ... -->
        <http-basic realm="Secured Demo Area" provider="in_memory" />
        <form-login />
    </firewall>
</config>
```

- *PHP*

```
<?php
// app/config/config.php
$container->loadFromExtension('security', array(
```

```
        'firewalls' => array(
            'secured_area' => array(
                // ...
                'provider' => 'user_db',
                'http_basic' => array(
                    // ...
                    'provider' => 'in_memory',
                ),
                'form_login' => array(),
            ),
        ),
    ));
```

В этом примере, если пользователь пытается залогиниться при помощи HTTP аутентификации - будет использоваться провайдер `in_memory`, но если пользователь попытается залогиниться при помощи формы логина, будет использован провайдер `user_db` (так как этот провайдер является провайдером по умолчанию для всего брандмауэра).

Подробную информацию о провайдерах пользователей и настройках брандмауэра вы можете прочитать в справочнике: `/reference/configuration/security`.

2.12.6 Роли

Роль имеет ключевое значение в процессе авторизации. Каждый пользователь получает набор ролей и каждый ресурс требует наличие одной или нескольких ролей. Если пользователь имеет необходимую роль - доступ будет разрешён. В противном случае - доступ будет запрещён.

Роли, по сути, очень просты, это строки, которые вы можете создавать и использовать по мере надобности (тем не менее, внутри системы роли это всё-таки объекты). Например, если вам нужно ограничить доступ к админке блога на вашем сайте, вы можете защитить эту секцию, используя роль `ROLE_BLOG_ADMIN`. Эта роль не должна быть нигде определена, вы просто начинаете её использовать и всё.

Примечание: Все роли в Symfony2 **должны** начинаться с префикса `ROLE_`. Если вы определяете ваши роли в отдельном классе `Role` (продвинутый вариант), использовать префикс `ROLE_` не нужно.

Иерархические роли

Вместо того, чтобы присваивать пользователю много ролей, вы можете определить правила наследования ролей, создав их иерархию:

- *YAML*

```
# app/config/security.yml
security:
    role_hierarchy:
        ROLE_ADMIN:       ROLE_USER
        ROLE_SUPER_ADMIN: [ROLE_ADMIN, ROLE_ALLOWED_TO_SWITCH]
```

- *XML*

```
<!-- app/config/security.xml -->
<config>
    <role id="ROLE_ADMIN">ROLE_USER</role>
    <role id="ROLE_SUPER_ADMIN">ROLE_ADMIN, ROLE_ALLOWED_TO_SWITCH</role>
</config>
```

- *PHP*

```
<?php
// app/config/security.php
$container->loadFromExtension('security', array(
    'role_hierarchy' => array(
        'ROLE_ADMIN'       => 'ROLE_USER',
        'ROLE_SUPER_ADMIN' => array('ROLE_ADMIN', 'ROLE_ALLOWED_TO_SWITCH'),
    ),
));
```

В примере выше, пользователь с ролью `ROLE_ADMIN` будет также иметь роль `ROLE_USER`. Роль `ROLE_SUPER_ADMIN` включает в себя `ROLE_ADMIN`, `ROLE_ALLOWED_TO_SWITCH`, и `ROLE_USER` (унаследовав её от `ROLE_ADMIN`).

2.12.7 Выход из системы

Как правило, вы также захотите дать вашим пользователям возможность выйти из системы. К счастью, брандмауэр позволяет обрабатывать выход автоматически, если вы активируете параметр `logout` в конфигурации:

- *YAML*

```
# app/config/config.yml
security:
    firewalls:
        secured_area:
            # ...
            logout:
                path:   /logout
                target: /
    # ...
```

- *XML*

```
<!-- app/config/config.xml -->
<config>
    <firewall name="secured_area" pattern="^/">
        <!-- ... -->
        <logout path="/logout" target="/" />
    </firewall>
    <!-- ... -->
</config>
```

- *PHP*

```
<?php
// app/config/config.php
$container->loadFromExtension('security', array(
    'firewalls' => array(
        'secured_area' => array(
            // ...
            'logout' => array('path' => 'logout', 'target' => '/'),
        ),
    ),
    // ...
));
```

Будучи настроенной для вашего брандмауэра, эта конфигурация при направлении пользователя на `/logout` (или любой другой путь, который вы укажете в параметре `path`) будет де-аутентифицировать его. Этот пользователь будет перенаправлен на главную страницу сайта (также может быть настроено при помощи параметра `target`). Оба эти параметра - `path` и `target` имеют параметры по умолчанию, такие же, как указаны в примере выше. Другими словами, вы можете их не указывать, что упростит настройку:

- *YAML*

```
logout: ~
```

- *XML*

```
<logout />
```

- *PHP*

```
'logout' => array(),
```

Отметим также, что вам *не* нужно создавать контроллер для URL `/logout`, так как брандмауэр сам позаботится обо всём. Возможно, вы также захотите создать маршрут и использовать его для генерации URL:

- *YAML*

```
# app/config/routing.yml
logout:
    pattern:   /logout
```

- *XML*

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

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

    <route id="logout" pattern="/logout" />

</routes>
```

- *PHP*

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

$collection = new RouteCollection();
$collection->add('logout', new Route('/logout', array()));

return $collection;
```

После того как пользователь выходит из системы, он будет перенаправлен по пути, указанному в параметре `target` (например `homepage`). Подробнее о конфигурации logout читайте в `Справочнике по настройке системы безопасности`.

2.12.8 Контроль доступа в шаблонах

Если вы хотите проверить, имеет ли пользователь некоторую роль внутри шаблона, воспользуйтесь встроенным хелпером:

- *Twig*

```
{% if is_granted('ROLE_ADMIN') %}
    <a href="...">Delete</a>
{% endif %}
```

- *PHP*

```
<?php if ($view['security']->isGranted('ROLE_ADMIN')): ?>
    <a href="...">Delete</a>
<?php endif; ?>
```

Примечание: Если вы используете эту функцию на странице, URL которой *не* обрабатывается брандмауэром, будет брошено исключение. Напомним ещё раз, что в большинстве случаев хорошей практикой является наличие главного брандмауэра, который контролирует все URL (как было показано в этой главе).

2.12.9 Контроль доступа в контроллерах

Если вы хотите проверить, имеет ли текущий пользователь ту или иную роль в вашем контроллере, используйте метод `isGranted` контекста безопасности:

```
<?php
public function indexAction()
{
    // show different content to admin users
    if ($this->get('security.context')->isGranted('ADMIN')) {
        // Загружаем админ-контент
    }
    // Загружаем прочий контент
}
```

Примечание: Брандмауэр должен быть активен, или же будет брошено исключение при вызове метода `isGranted`. Посмотрите также замечание для шаблонов чуть выше.

2.12.10 Подмена пользователя

Иногда необходимо иметь возможность переключения с одного пользователя на другого без выполнения выхода/входа (например, при отладке или при попытке воспроизвести баг, который пользователь видит, а вы нет). Это можно выполнить при помощи листенера `switch_user` в брандмауэре:

- *YAML*

```
# app/config/security.yml
security:
    firewalls:
        main:
            # ...
            switch_user: true
```

- *XML*

```
<!-- app/config/security.xml -->
<config>
    <firewall>
        <!-- ... -->
        <switch-user />
    </firewall>
</config>
```

- *PHP*

```
<?php
// app/config/security.php
$container->loadFromExtension('security', array(
    'firewalls' => array(
        'main'=> array(
            // ...
            'switch_user' => true
        ),
    ),
));
```

Для переключения на другого пользователя просто добавьте в строку запроса текущего URL параметр `_switch_user` и имя пользователя:

http://example.com/somewhere?_switch_user=thomas

Для того, чтобы переключиться обратно, используйте специальное имя `_exit`:

http://example.com/somewhere?_switch_user=_exit

Естественно, такая возможность должна быть доступна небольшой группе пользователей. По умолчанию, эта функция доступна пользователям с ролью `ROLE_ALLOWED_TO_SWITCH`. Наименование этой роли можно изменить при помощи опции `role`. Для большей безопасности вы также можете изменить наименования параметра для строки запроса при помощи опции `parameter`:

- *YAML*

```
# app/config/security.yml
security:
    firewalls:
        main:
            // ...
            switch_user: { role: ROLE_ADMIN, parameter: _want_to_be_this_user }
```

- *XML*

```
<!-- app/config/security.xml -->
<config>
    <firewall>
        <!-- ... -->
        <switch-user role="ROLE_ADMIN" parameter="_want_to_be_this_user" />
    </firewall>
</config>
```

- *PHP*

```
<?php
// app/config/security.php
$container->loadFromExtension('security', array(
    'firewalls' => array(
        'main'=> array(
            // ...
            'switch_user' => array('role' => 'ROLE_ADMIN', 'parameter' => '_want_to_be_this
        ),
    ),
));
```

2.12.11 Аутентификация без сохранения состояния (stateless)

По умолчанию, Symfony2 использует куки (сессию) для хранения контекста безопасности пользователя. Но, если вы используете, к примеру, сертификаты или HTTP аутентификацию, сохранение не требуется, так как авторизационные данные доступны для каждого запроса. В этом случае, и если вы не хотите сохранять что-либо между запросами, вы можете активировать *stateless аутентификацию* (без сохранения состояния, т.е. Symfony2 не будет создавать куки):

- *YAML*

```
# app/config/security.yml
security:
    firewalls:
        main:
            http_basic: ~
            stateless:  true
```

- *XML*

```
<!-- app/config/security.xml -->
<config>
    <firewall stateless="true">
        <http-basic />
    </firewall>
</config>
```

- *PHP*

```
<?php
// app/config/security.php
$container->loadFromExtension('security', array(
    'firewalls' => array(
        'main' => array('http_basic' => array(), 'stateless' => true),
    ),
));
```

Примечание: Если вы используете форму логина, Symfony2 будет создавать куки всегда, даже если `stateless` имеет значение `true`.

2.12.12 Заключение

Безопасность может быть весьма сложным вопросом для решения его в вашем приложении. К счастью, компонент безопасности Symfony следует хорошо зарекомендовавшей себя модели, основанной на *аутентификации* и *авторизации*. Аутентификация, которая всегда идёт первой, обрабатывается брандмауэром, чья работа заключается установить "личность" пользователя при помощи любого из доступных методов (HTTP аутентификация, форма логина и т.д.). В книге рецептов вы также найдёте примеры других методов аутентификации, включая то, как реализовать функцию "запомнить меня" при помощи куки.

После того как пользователь аутентифицирован, авторизация может определить имеет ли он доступ к тому или иному ресурсу. Как правило, к URL, классам или методам ставятся в соответствие некоторые *роли* и если пользователь не имеет требуемой роли, доступ ему будет запрещён. Тем не менее, авторизация это более сложный механизм, следующий системе "голосования", благодаря которой множество разных участников могут определить имеет ли пользователь права доступа к ресурсу или же нет.

2.12.13 Читайте также в книге рецептов

- `Форсирование HTTP/HTTPS`
- `Блэклистинг пользователя по IP при помощи custom voter`
- `Списки контроля доступа (ACLs)`
- `/cookbook/security/remember_me`

2.13 HTTP Кэширование

Природа насыщенных (богатых) веб-приложений подразумевает, что они динамические. Вне зависимости от того, насколько эффективно ваше приложение, каждый запрос будет содержать работы больше чем отдача простого статического файла.

И для большинства веб-приложений это вполне нормально. Symfony2 очень быстр и, если вы не делаете чего-то действительно тяжеловесного, каждый запрос будет обрабатываться быстро и не создавая стрессовых ситуаций на сервере.

Но, по мере роста вашего сайта, рост нагрузки может стать проблемой. Работа, которая обычно выполняется для каждого запроса, теперь должна быть выполнена только единожды. И это именно то, чего позволяет добиться кэширование.

2.13.1 Кэширование на плечах гигантов

Наиболее эффективным способом увеличить быстродействие приложения является кэширование страницы целиком и затем, в обход приложения, отдавать кэшированные данные для каждого запроса. Конечно же, это не всегда возможно применить, особенно для очень динамично меняющихся сайтов... или всё же возможно? В этой главе вы увидите, как работает система кэширования Symfony2 и почему мы считаем это наилучшим решением из возможных.

Система кэширования Symfony2 отличается от других, так как она полагается на простоту и мощь HTTP кэширования, как это определено в спецификации HTTP (см. *Спецификация протокола HTTP*). Вместо того чтобы изобретать кэширование заново, Symfony2 пользуется стандартом, который определяет базовые коммуникации в Web. Как только вы поймёте основополагающие модели HTTP валидации и истечения срока для кэша, вы будете готовы к управлению системой кэширования Symfony2.

С целью изучения того, как кэшировать в Symfony2, мы пройдём четыре шага:

- **Шаг 1**: *кэширующий шлюз*, или обратный прокси-сервер (reverse proxy), это независимый слой, который располагается перед вашим приложением. Обратный прокси кэширует ответы по мере их поступления от приложения и отвечает на запросы при помощи кэшированных ответов, не подключая приложение. Symfony2 содержит свой собственный обратный прокси, но вы также можете использовать любой обратный прокси на ваш выбор.
- **Шаг 2**: заголовки *HTTP кэша* используются для коммуникации кэширующего шлюза и любого другого кэшера, который может находиться между вашим приложением и клиентом. Symfony2 содержит типовую конфигурацию по умолчанию и мощный интерфейс для работы с заголовками кэша.
- **Шаг 3**: *окончание срока действия и валидация HTTP кэша* - это две модели, используемые для определения является ли кэшированный контент *свежим* (и

может повторно браться из кэша) или же *просроченным* (и его необходимо пересоздать при помощи приложения).

- **Шаг 4**: *Edge Side Includes* (ESI) позволяют использовать HTTP кэш для независимого кэширования фрагментов страниц (даже вложенных фрагментов). При помощи ESI вы можете кэшировать всю страницу на 60 минут, но встроенную боковую панель лишь на 5 минут.

Так как HTTP кэширование не является достоянием лишь Symfony, существует множество статей по данной теме. Если вы новичок в HTTP кэшировании, мы *настоятельно* рекомендуем вам прочитать статью *Ryan Tomayko*: Things Caches Do. Другим исчерпывающим руководством является Cache Tutorial от *Mark Nottingham*.

2.13.2 Кэширование при помощи кэширующего шлюза

При кэшировании при помощи HTTP, *кэш* полностью отделён от вашего приложения и располагается между вашим приложением и клиентом, выполнившем запрос.

Работа кэша заключается в приёме запроса от клиента и передаче его вашему приложению. Кэш также будет получать ответ от вашего приложения и перенаправлять его далее к клиенту. Кэш является посредником в клиент-серверных коммуникациях между клиентом и вашим приложением.

По пути, кэш будет сохранять каждый ответ, который полагает "кэшируемым" (см. *Введение в HTTP кэширование*). Если этот же ресурс будет запрошен ещё раз, кэш отправит сохранённый (кэшированный) ответ клиенту, игнорируя ваше приложение.

Этот тип кэширования известен под именем "кэширующего HTTP шлюза". Существует много кэшеров такого типа, например: Varnish, Squid в режиме обратного прокси, а также обратный прокси Symfony2.

Типы кэширования

Но кэширующим шлюзом типы кэшеров не исчерпываются. Фактически, заголовки HTTP кэша, отправляемые вашим приложением, могут быть получены и использованы тремя различными типами кэшеров:

- *Кэш браузера*: Каждый браузер имеет свой собственный локальный кэш, который в основном используется, когда вы нажимаете кнопку "back", а также кэш картинок и прочих ресурсов. Кэш браузера - это *личный* кэш, который не используется никем более.
- *Кэширующие прокси*: Прокси - это кэш *общего доступа*, так как за одним таким прокси может находиться много клиентов. Такие прокси как правило устанавливаются большими компаниями и Интернет-провайдером для уменьшения времени доступа к ресурсам и снижению сетевого трафика.

- *Кэширующие шлюзы*: Как и прокси, они также представляют собой кэш *общего доступа*, но на стороне сервера. Устанавливаемые администраторами, они делают сайты более масштабируемыми, надёжными и быстрыми.

Совет: Кэширующие шлюзы иногда называют кэширующими обратными прокси, суррогатными кэшерами и даже HTTP акселераторами.

Примечание: Значимость *личного* кэша по сравнению с кэшем *общего доступа* становится более заметной, если мы говорим о кэшировании ответов, содержащих контент, относящийся к конкретному пользователю (например, информация о счёте).

Каждый ответ от вашего приложения будет проходить через первый тип кэша или же через оба - первый и второй. Эти кэши находятся вне вашего контроля, но следуют указаниям для HTTP кэша, которые есть в ответе.

Обратный прокси Symfony2

Symfony2 содержит обратный прокси (также называемый кэширующим шлюзом), написанный на PHP. Активируйте его и кэшируемые ответы вашего приложения начнут кэшироваться надлежащим образом. Его установка очень проста. Каждое новое приложение Symfony2 содержит уже настроенное кэширующее ядро (`AppCache`), которое служит оболочкой для ядра по умолчанию (`AppKernel`). Кэширующее ядро и есть *том самый* обратный прокси.

Для того чтобы активировать кэширование, модифицируйте код фронт-контроллера таким образом, чтобы он использовал кэширующее ядро:

```
<?php
// web/app.php

require_once __DIR__.'/../app/bootstrap.php.cache';
require_once __DIR__.'/../app/AppKernel.php';
require_once __DIR__.'/../app/AppCache.php';

use Symfony\Component\HttpFoundation\Request;

$kernel = new AppKernel('prod', false);
$kernel->loadClassCache();
// wrap the default AppKernel with the AppCache one
$kernel = new AppCache($kernel);
$kernel->handle(Request::createFromGlobals())->send();
```

Кэширующее ядро немедленно начнёт действовать в качестве обратного прокси - будет кэшировать ответы вашего приложения и отправлять их клиенту.

Совет: Кэширующее ядро имеет особый метод `getLog()`, который возвращает строковое представление того, что происходит на кэширующем уровне. В dev окружении вы можете использовать его для отладки и проверки вашей стратегии кэширования:

```
error_log($kernel->getLog());
```

Объект `AppCache` имеет конфигурацию по умолчанию, но вы можете конфигурировать и настраивать его опции посредством переопределения метода `getOptions()`:

```
<?php
// 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,
        );
    }
}
```

Совет: Для изменения опции `debug` переопределять `getOptions()` не обязательно, так как она автоматически принимает значение параметра debug от `AppKernel`.

Ниже представлен список основных опций:

- `default_ttl`: Время (в секундах), в течение которого кэшированный элемент считается свежим, если ответ не содержит точных данных о его "свежести". Явно указанные заголовки `Cache-Control` или `Expires` перезаписывают это значение (по умолчанию `0`);
- `private_headers`: Набор заголовков запроса, которые активируют "приватный" `Cache-Control` для ответов, которые явно не указывают поведение "приватный" или "публичный" посредством директивы `Cache-Control` (по умолчанию `Authorization` и `Cookie`);
- `allow_reload`: Определяет, может ли клиент форсировать обновление кэша при помощи директивы `Cache-Control` "no-cache" в запросе. Установите её в `true` для следования спецификации RFC 2616 (по умолчанию `false`);

- `allow_revalidate`: Определяет, может ли клиент форсировать перепроверку кэша при помощи директивы `Cache-Control` "max-age=0" в запросе. Установите её в `true` для следования спецификации RFC 2616 (по умолчанию `false`);
- `stale_while_revalidate`: Определяет число секунд по умолчанию (квантификация времени производится в секундах, так как TTL (time to live) ответа измеряется в секундах) во время которого кэш будет немедленно возвращать просроченный ответ, пока производится его фоновая перепроверка (по умолчанию `2`); эта опция переопределяется расширением HTTP `Cache-Control - stale-while-revalidate` (см. RFC 5861);
- `stale_if_error`: Определяет число секунд по умолчанию (квантификация времени производится в секундах, так как TTL (time to live) ответа измеряется в секундах), во время которого кэш может обслуживать просроченный ответ, если возникает ошибка (по умолчанию `60`). Эта опция переопределяется расширением HTTP `Cache-Control - stale-if-error` (см. RFC 5861)

Если `debug` имеет значение `true`, Symfony2 автоматически добавляет в ответ заголовок `X-Symfony-Cache`, содержащий полезную информацию о числе срабатываний кэша и о числе не найденных ответов в кэше.

Использование другого обратного прокси

Обратный прокси Symfony2 это отличный инструмент для использования во время разработки или же при выгрузке вашего сайта на виртуальный (шаред) хостинг, где вы не можете установить ничего, кроме PHP кода. Но, прокси на PHP никогда не будет быстрее прокси на Си. Вот почему мы настоятельно рекомендуем вам использовать Varnish или Squid на ваших продуктовых серверах, если это возможно. Хорошей новостью для вас будет то, что переключение с одного прокси сервера на другой выполняется просто и прозрачно и не требует модификации кода вашего приложения. Просто начните работу с обратным прокси Symfony2 и замените его на Varnish, когда трафик возрастёт.
Больше об использовании Varnish с Symfony2 читайте в книге рецептов: *Как использовать Varnish*.

Примечание: Быстродействие обратного прокси Symfony2 не зависит от сложности приложения. Это достигается за счёт того, что ядро приложения загружается лишь в том случае, когда к нему требуется перенаправить входящий запрос.

2.13.3 Введение в HTTP кэширование

Для того, чтобы получить пользу от кэширования, ваше приложение должно иметь возможность сообщить, какие ответы могут быть кэшированы, а также правила, ко-

торые будут указывать когда и как истекает срок действия этого кэша. Этого можно достичь при помощи HTTP заголовков для кэширования ответов.

Совет: Имейте в виду, что "HTTP" это не более чем язык (простой текстовый язык), который веб клиенты (например, браузеры) и веб серверы используют для коммуникаций между собой. Когда мы говорим об HTTP кэшировании, мы говорим о части этого языка, которая позволяется клиентам и серверам обмениваться информацией, относящейся к кэшированию.

Спецификация HTTP содержит четыре заголовка, относящихся к кэшированию:

- `Cache-Control`
- `Expires`
- `ETag`
- `Last-Modified`

Наиболее важным и многосторонним является заголовок `Cache-Control`, который на самом деле является коллекцией разнообразной информации о кэшировании.

Примечание: Каждый из заголовков будет детально рассмотрен в секции *Модели кэширования в HTTP: expiration и validation*.

Заголовок Cache-Control

Заголовок `Cache-Control` уникален за счёт того, что он содержит не одно конкретное значение, а много различных данных о кэшируемости ответа. Каждая новая порция данных отделяется запятой:

```
Cache-Control: private, max-age=0, must-revalidate
```

```
Cache-Control: max-age=3600, must-revalidate
```

Symfony предоставляет методы для более удобного управления заголовком `Cache-Control`:

```
<?php
//...

$response = new Response();

// пометить ответ как public или private
$response->setPublic();
$response->setPrivate();
```

```
// установить max age для private и shared ответов
$response->setMaxAge(600);
$response->setSharedMaxAge(600);

// установить специальную директиву Cache-Control
$response->headers->addCacheControlDirective('must-revalidate', true);
```

Публичные (public) vs Частные (private) ответы

Кэширующие шлюзы и прокси рассматривают "общие" кэши как кэшированный контент, который используется более чем одним пользователем. Если будет случайно сохранён ответ, специфичный для отдельного пользователя, впоследствии он может быть отправлен множеству различных пользователей. Представьте, что информация о вашем счёте была кэширована и будет отправлена любому пользователю, который запросит свою собственную страницу со счётом!

Для того чтобы корректно обработать эту ситуацию, каждый ответ может быть объявлен публичным или же частным:

- *public*: Публичный ответ может кэшироваться как частным, так и публичным кэшами;
- *private*: Частный ответ подразумевает что он целиком или же его часть предназначена для одного единственного пользователя и не должен кэшироваться публичными кэшерами.

Symfony действует консервативно и помечает каждый ответ как частный. Для того чтобы получить преимущества от использования публичных кэшеров (в том числе и обратного прокси Symfony2), ответ должен быть помечен как публичный (public).

Безопасные методы

HTTP кэширование работает лишь для "безопасных" HTTP методов (таких как GET и HEAD). Под безопасностью этих методов понимается, что вы никогда не измените состояние приложения при обработке таких запросов (при этом вы, конечно, можете логгировать информацию, кэшировать данные и т.д.). Это ограничение имеет два следствия:

- Вы *никогда* не должны изменять состояние вашего приложения, отвечая на GET или HEAD запрос. Даже если вы не используете кэширующий шлюз, наличие прокси-кэша означает, что любой GET или HEAD запрос может как попасть в ваше приложение, так и не попасть (прокси вернёт кэшированные данные, не затрагивая приложение).
- Ни в коем случае не кэшируйте PUT, POST и DELETE методы. Эти методы предназначены для изменения состояния приложения (например, удаления записи из

блога). Если их кэшировать, то часть запросов на изменение состояния приложения не будут достигать его.

Правила кэширования и значения по умолчанию

HTTP 1.1 по умолчанию разрешает кэширование, если явно не указан заголовок `Cache-Control`. На практике, большинство кэшеров ничего не делают, если запросы имеют куки, авторизационный заголовок, используют небезопасные методы (т.е. PUT, POST, DELETE), или когда ответ имеет перенаправляющий статус-код (например, 301 или 302).

Symfony2 автоматически устанавливает разумно-консервативный заголовок `Cache-Control`, если разработчик не задал правила кэширования явно. Эти умолчания следуют следующим правилам:

- Если не определены заголовки кэширования (`Cache-Control`, `Expires`, `ETag` или `Last-Modified`), `Cache-Control` устанавливается в значение `no-cache`, то есть ответ кэшироваться не будет;
- Если `Cache-Control` пустой (но присутствует любой другой кэширующий заголовок), его значение устанавливается в `private, must-revalidate`;
- Если присутствует хотя бы одна директива `Cache-Control` и явно не указаны директивы `public` или `private`, Symfony2 добавляет директиву `private` автоматически (за исключением случая, когда установлен `s-maxage`).

2.13.4 Модели кэширования в HTTP: expiration и validation

Спецификация HTTP определяет две модели кэширования:

- Первая - модель "окончания срока действия" (expiration), вы просто указываете как долго ответ будет "свежим", включая заголовки `Cache-Control` и/или `Expires`. Кэшеры, которые поддерживают эту модель, не будут выполнять некоторый запрос до тех пор, пока его кэшированная версия не достигнет окончания срока действия (`expiration`) и не станет "просроченной".
- Когда страницы очень быстро меняются, часто бывает необходимо использовать модель валидации (validation). При использовании этой модели кэшер сохраняет ответ, но при каждом последующем запросе он запрашивает сервер - является ли кэшированный ответ валидным или нет. Приложение использует некоторый уникальный идентификатор ответа (заголовок `Etag`) и/или временную метку (заголовок `Last-Modified`) для проверки изменилась ли страница с момента её кэширования.

Целью обоих этих моделей является следующая: не генерировать один и тот же ответ дважды, если в кэше уже есть "свежий" ответ, сохранённый там ранее.

Читаем спецификацию HTTP

Спецификация HTTP определяет простой, но мощный язык, при помощи которого осуществляются клиент-серверные коммуникации в сети. Модель запрос-ответ определяет всю вашу работу, как веб-разработчика. К несчастью, оригинальную спецификацию - RFC 2616 - читать весьма непросто.
В настоящее время существует инициатива (HTTP Bis) по переписыванию RFC 2616. Она не ставит целью написание новой версии HTTP, а в основном сосредоточена на разъяснении оригинальной спецификации HTTP. Структура спецификации также подверглась улучшению - она разбита на семь частей; всё что относится к HTTP кэшированию - расположено в двух независимых частях (P4 - Conditional Requests и P6 - Caching: Browser and intermediary caches).
Вам, как веб разработчику, мы настоятельно рекомендуем прочитать эту спецификацию. Её простота и сила - даже спустя десять лет после её написания - бесценны. И не бойтесь внешнего вида спецификации - её содержание много лучше, чем её обложка.

HTTP Expiration - окончание строка действия

Модель окончания срока действия более эффективная и простая из двух поддерживаемых моделей кэширования и должна использоваться везде, где это возможно. Когда ответ кэшируется со сроком окончания действия, кэш будет хранить ответ и возвращать его на клиент напрямую, не затрагивая приложение, пока срок действия не окончится.

Модель окончания срока действия может быть задействована с использованием двух похожих HTTP заголовков: `Expires` или `Cache-Control`.

Окончание срока действия при помощи заголовка `Expires`

Следуя спецификации HTTP, "заголовок `Expires` содержит дату/время, после которого этот ответ будет считаться просроченным". Заголовок `Expires` может быть установлен при помощи метода `setExpires()` класса `Response`. Он принимает экземпляр `DateTime` в качестве аргумента:

```
<?php
//...
$date = new DateTime();
$date->modify('+600 seconds');

$response->setExpires($date);
```

Результирующий заголовок будет выглядеть следующим образом:

```
Expires: Thu, 01 Mar 2011 16:00:00 GMT
```

Примечание: Метод `setExpires()` автоматически конвертирует дату в зону GMT, как того требует спецификация.

Заголовок `Expires` имеет 2 ограничения. Первое, часы на веб-сервере и и часы кэшера (например, браузера) должны быть синхронизированными. Второе, следует из спецификации и гласит, что "HTTP/1.1 серверы никогда не должны устанавливать дату `Expires` более чем на один год вперёд".

Окончание срока действия при помощи заголовка `Cache-Control`

Поскольку заголовок `Expires` имеет ограничения, вы должны использовать заголовок `Cache-Control`. Вспоминайте, что заголовок `Cache-Control` используется для указания различных директив, относящихся к кэшированию. Для окончания срока действия имеются две директивы, `max-age` и `s-maxage`. Первая используется всеми кэшерами, в то время как вторая используется лишь "общими" (shared) кэшами:

```
<?php
//...
// Устанавливаем число секунд, после которого ответ более не будет считаться свежим
$response->setMaxAge(600);

// Тоже что и выше, но только для общих кэшей
$response->setSharedMaxAge(600);
```

Заголовок `Cache-Control` будет иметь следующий формат (также там могут быть и другие директивы):

```
Cache-Control: max-age=600, s-maxage=600
```

Валидация

Когда некоторый ресурс должен быть обновлён, в связи с тем, что произошли изменения в данных, лежащих в его основе, модель окончания срока действия становится несостоятельной. При подходе, используемом в модели окончания срока действия, кэш не обратится к приложению для обновления ответа пока данные не становятся просроченными (т.е. когда истечёт срок действия кэшированного ответа).

Модель валидации решает эту проблему. С её помощью кэш также продолжает сохранять ответы. Различие заключается в том, что для каждого запроса, кэш запрашивает приложение изменился или нет запрашиваемый ресурс. Если кэш *ещё* валиден, ваше приложение должно вернуть статус код 304 и не возвращать контент. Это означает, что кэш ещё валиден и можно возвращать кэшированный ответ.

С этой моделью вы, прежде всего, сохраняете пропускную способность вашего интернет-канала, так как страница целиком не отсылается дважды тому же клиенту (вместо этого будет отправлен ответ со статус кодом 304). Но, если вы аккуратно проектируете ваше приложение, мы можете получить необходимый минимум данных, необходимых для того чтобы отправить статус код 304 и сохранить также ресурсы CPU и/или оперативной памяти (см. ниже реализацию этого варианта).

Совет: Статус 304 означает "Not Modified". Это важный статус, так как вместе с ним не отправляется запрошенный контент. Вместо этого, ответ состоит из небольшого набора указаний, которые сообщают кэшу, что можно использовать сохранённую ранее версию.

Как и в случае с моделью окончания срока действия, есть два HTTP заголовка, которые могут быть использованы для реализации модели валидации: `ETag` и `Last-Modified`.

Валидация при помощи заголовка `ETag`

Заголовок `ETag` - это строковый заголовок (называемый "entity-tag"), который единственным образом идентифицирует представление целевого ресурса. Он генерируется и устанавливается всецело внутри вашего приложения, так что вы можете понять, к примеру, соответствует ли кэшированный ресурс `/about` тому, который ваше приложение собирается вернуть. Заголовок `ETag` похож на отпечатки пальцев и используется для быстрого определения эквивалентны ли две версии ресурса. Как и отпечатки пальцев, каждый `ETag` должен быть уникальным для любого представления одного и того же ресурса.

Давайте взглянем на простую реализацию, которая генерирует ETag в виде md5 хэша от контента:

```
<?php
//...
public function indexAction()
{
    $response = $this->render('MyBundle:Main:index.html.twig');
    $response->setETag(md5($response->getContent()));
    $response->isNotModified($this->getRequest());

    return $response;
}
```

Метод `Response::isNotModified()` сравнивает `ETag`, отправленный в запросе (`Request`) с этим же тагом в ответе (`Response`). Если они совпадают, этот метод автоматически устанавливает для `Response` статус код 304.

Этот алгоритм достаточно простой и вполне типичный, но вам нужно создать экземпляр `Response` целиком, перед тем как вы получите возможность сравнить ETag'и, а

это весьма расточительно. Другими словами, этот подход сохраняет пропускную способность, но не ресурсы CPU.

В секции *Оптимизация вашего кода при помощи метода валидации* мы покажем как можно использовать валидацию более интеллигентно и определять валидность кэша без излишних затрат ресурсов сервера.

Совет: Symfony2 также поддерживает "слабые" ETag'и - для этого надо передать `true` в качестве второго аргумента в метод **:method:'Symfony\\Component\\HttpFoundation\\Response::setETag'**.

Валидация при помощи заголовка `Last-Modified`

Заголовок `Last-Modified` - это второй возможный способ валидации. Следуя спецификации HTTP, "Заголовок `Last-Modified` содержит дату и время, когда представление ресурса было изменено в последний раз, по версии исходного сервера". Другими словами, приложение принимает решение о том, должен ли быть обновлён кэшированный контент, основываясь на том, изменялся ли он со времени кэширования.

Например, вы можете использовать дату последнего обновления для всех объектов, необходимых для создания представления ресурса в качестве значения заголовка `Last-Modified`:

```
<?php
//...
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;
}
```

Метод `Response::isNotModified()` сравнивает заголовок `If-Modified-Since`, отправленный в запросе с заголовком `Last-Modified`, установленном в ответе. Если они идентичны, `Response` будет установлен статус код 304.

Примечание: Заголовок запроса `If-Modified-Since` соответствует заголовку

`Last-Modified` последнего ответа, отправленного клиенту для некоторого ресурса. Таким образом, клиент и сервер общаются друг с другом и определяют был ли ресурс обновлён с момента его кэширования.

Оптимизация вашего кода при помощи метода валидации

Основная цель любой стратегии кэширования - понизить нагрузку на приложение. Иными словами, чем меньше делает ваше приложение для того, чтобы вернуть ответ 304, тем лучше. Метод `Response::isNotModified()` именно этим и занимается при использовании простого и эффективного шаблона:

```
<?php
//...
public function showAction($articleSlug)
{
    // Получаем минимум информации для вычисления
    // значений для заголовков ETag или Last-Modified
    // (основываясь на запросе Request, данных, получаемых из базы данных
    // или же из хранилища ключ-значение)
    $article = // ...

    // Создаём ответ Response с заголовком ETag и/или Last-Modified
    $response = new Response();
    $response->setETag($article->computeETag());
    $response->setLastModified($article->getPublishedAt());

    // Проверяем, что ответ не модифицировался для этого запроса
    if ($response->isNotModified($this->getRequest())) {
        // возвращаем ответ 304
        return $response;
    } else {
        // делаем дополнительные действия, например, получаем дополнительные данные
        $comments = // ...

        // или отображаем шаблон при помощи $response, который был создан ранее
        return $this->render(
            'MyBundle:MyController:article.html.twig',
            array('article' => $article, 'comments' => $comments),
            $response
        );
    }
}
```

Если ответ `Response` не модифицировался, метод `isNotModified()` автоматически устанавливает статус код ответа в `304`, удаляет контент и удаляет некоторые заголовки, которые не должны присутствовать в ответе 304 (см. метод

:method:‘Symfony\\Component\\HttpFoundation\\Response::setNotModified‘).

Вариации ответа

Ранее вы узнали, что каждый URI имеет единственное представление целевого ресурса. По умолчанию, HTTP кэширование выполняется с использованием URI ресурса в качестве ключа к значению кэша. Если два человека запросят один и тот же URI для кэшируемого ресурса, второй клиент получит уже кэшированную версию.

Иногда этого не достаточно и требуется кэшировать различные версии одного и того же URI, основываясь на значении одного или нескольких заголовков. Например, если вы сжимаете страницы для клинентов, которые поддерживают сжатие, любой URI будет иметь два представления: одно для клиентов, поддерживающих сжатие, и одно для тех кто не поддерживает. Это определяется на основе значения заголовка запроса `Accept-Encoding`.

В этом случае, вам необходимо хранить обе версии ответа для некоторого ресурса - сжатую и не сжатую и возвращать ее, основываясь на значении заголовка запроса `Accept-Encoding`. Этого можно достичь при помощи заголовка ответа `Vary`, который является списком (разделители - запятые) различных заголовков, чьи значения переключают различные представления запрошенного ресурса:

```
Vary: Accept-Encoding, User-Agent
```

Совет: Заголовок `Vary` из примера выше позволяет кэшировать различные версии для каждого ресурса, основываясь на URI и значении заголовков запроса `Accept-Encoding` и `User-Agent`.

Объект `Response` предоставляет простой интерфейс для управления заголовком `Vary`:

```
<?php
//...
// устанавливаем один заголовок vary
$response->setVary('Accept-Encoding');

// устанавливаем несколько заголовков vary
$response->setVary(array('Accept-Encoding', 'User-Agent'));
```

Метод `setVary()` принимает в качестве параметра имя заголовка или же массив наименований заголовков, на основании значений которых необходимо варьировать ответ.

Окончание срока действия и валидация

Вы можете использовать окончание срока действия совместно с валидацией в одном и том же экземпляре `Response`. Если окончание срока действия работает раньше валида-

ции, вы сможете получить лучшие преимущества от обеих моделей. Другими словами, используя совместно модели окончание срока действия и валидации вы можете проинструктировать кэш хранить контент пока с некоторым интервалом осуществляется (окончание срока действия) проверка, что контент всё ещё валиден.

Другие методы класса Response

Класс Response содержит также другие методы для работы с кэшем. Пример ниже иллюстрирует самые часто употребляемые из них:

```
<?php
// пометить ответ как "просроченный"
$response->expire();

// Форсировать возврат ответа 304 без контента
$response->setNotModified();
```

В дополнение к этому, все основные HTTP относящиеся к кэшу, могут быть установлены при помощи одного метода `setCache()`:

```
<?php
// Установить заголовки для кэширования одним вызовом
$response->setCache(array(
    'etag'          => $etag,
    'last_modified' => $date,
    'max_age'       => 10,
    's_maxage'      => 10,
    'public'        => true,
    // 'private'    => true,
));
```

2.13.5 Использование ESI (Edge Side Includes)

Кэширующие шлюзы - это отличный способ сделать ваш сайт более производительным. Но они также имеют и одно ограничение: они могут кэшировать лишь страницы целиком. Если вы по каким-то причинам не можете кэшировать страницы целиком или в случае когда страница имеет несколько динамических частей, вы вышли из зоны удачи. К счастью, Symfony2 предоставляет решение для этих случаев, основанное на технологии ESI, или Edge Side Includes. Компания Akamaï создала эту спецификацию почти 10 лет назад, и она позволяет иметь для отдельных частей страницы различные стратегии кэширования.

Спецификация ESI описывает таги, которые вы можете добавить в ваши страницы для общения с кэширующим шлюзом. В Symfony2 реализован лишь один таг - `include`, так как это наиболее полезный таг вне контекста Akamaï:

```
<html>
    <body>
        Some content

        <!-- Подключаем контент другой страницы -->
        <esi:include src="http://..." />

        More content
    </body>
</html>
```

Примечание: Обратите внимание, в примере выше, что для ESI тага указан полный URL. ESI таг представляет собой фрагмент страницы, который можно получить по этому URL.

При обработке запроса, кэширующий шлюз получает страницу целиком из своего кэша или же запрашивает его у приложения. Если ответ содержит один или более ESI тагов, они обрабатываются тем же образом. Другими словами, кэширущий шлюз получает включённые фрагменты страниц из своего кэша, либо запрашивает эти фрагменты у приложения. Когда все ESI таги обработаны, шлюз включает все фрагменты в основную страницу и отправляет итоговый контент клиенту.

Всё это происходит незаметно на уровне кэширующего шлюза (т.е. вне вашего приложения). Как вы увидите далее, если вы захотите использовать преимущества, которые предоставляют ESI таги, Symfony2 позволит вам подключать их не прилагая особых усилий.

Использование ESI в Symfony2

Во-первых, перед использованием ESI, убедитесь, что вы активировали их в настройках приложения:

- *YAML*

  ```
  # app/config/config.yml
  framework:
      # ...
      esi: { enabled: true }
  ```

- *XML*

  ```
  <!-- app/config/config.xml -->
  <framework:config ...>
      <!-- ... -->
      <framework:esi enabled="true" />
  </framework:config>
  ```

- *PHP*

```
<?php
// app/config/config.php
$container->loadFromExtension('framework', array(
    // ...
    'esi'    => array('enabled' => true),
));
```

Теперь, предположим, что у вас есть страница, которая по большей части статическая, за исключением новостей, расположенных под контентом. При помощи ESI вы можете кэшировать новости независимо от остальной страницы.

```
<?php
// ...
public function indexAction()
{
    $response = $this->render('MyBundle:MyController:index.html.twig');
    $response->setSharedMaxAge(600);

    return $response;
}
```

В этом примере вы устанавливаете для всей страницы время жизни кэша в 10 минут. Затем, подключите новости в шаблон при помощи встраивания действия. Это можно сделать при помощи хелпера `render` (см. *Внедрение контроллеров*).

Так как встроенный контент поступает из другой страницы (или контроллера в данном случае), Symfony2 использует стандартный хэлпер `render` для конфигурирования ESI тага:

- *Twig*

```
{% render '...:news' with {}, {'standalone': true} %}
```

- *PHP*

```
<?php echo $view['actions']->render('...:news', array(), array('standalone' => true)) ?>
```

Указав параметр `standalone` равный `true`, вы говорите Symfony2, что действие должно отображаться как ESI таг. Вы возможно удивлены - зачем использовать хелпер, вместо того, чтобы написать ESI таг самостоятельно. Это необходимо для того, чтобы ваше приложение работало даже если не установлен никакой кэширующий шлюз. Давайте разберём, как работает эта конструкция.

Когда опция standalone имеет значение `false` (по умолчанию), Symfony2 объединяет контент подключённой страницы с контентом основной перед отправкой ответа на

клиент. Но когда standalone имеет значение `true`, *и* если Symfony2 определяет, что кэширующий шлюз, через который работает приложение, поддерживает ESI, генерится ESI таг. Но если шлюз не обнаружен или же он не поддерживает ESI, Symfony2 будет объединять контент подключённой страницы с контентом основной также, как это было бы выполнено при значении standalone равном `false`.

Примечание: Symfony2 определяет, поддерживает ли шлюз ESI, при помощи другой спецификации Akamaï, которая поддерживается обратным прокси Symfony2 "из коробки".

Теперь для встроенного действия вы можете указать собственные правила кэширования, независимо от главной страницы:

```
<?php
public function newsAction()
{
  // ...

  $response->setSharedMaxAge(60);
}
```

При помощи ESI кэш страницы будет валидным в течение 600 секунд, но компонент новостей будет кэшироваться только на 60 секунд.

Требованием, при использовании ESI, является следующее: встроенное действие должно быть доступно через некоторый URL, чтобы кэширующий шлюз мог получить его контент независимо от остальной страницы. Конечно, действие не может быть доступным без маршрута, который указывает на него. Symfony2 заботится и об этом при помощи базового маршрута и контроллера. Чтобы ESI таг `include` работал, вы должны определить маршрут `_internal`:

- *YAML*

  ```
  # app/config/routing.yml
  _internal:
      resource: "@FrameworkBundle/Resources/config/routing/internal.xml"
      prefix:   /_internal
  ```

- *XML*

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

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

```
    <import resource="@FrameworkBundle/Resources/config/routing/internal.xml" prefix="/_int
</routes>
```

- *PHP*

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

$collection->addCollection($loader->import('@FrameworkBundle/Resources/config/routing/inter

return $collection;
```

Совет: Так как маршрут позволяет получить доступ к вашему действию при помощи URL, вы возможно захотите защитить его при помощи брандмауэра Symfony2 (разрешив доступ по IP вашего обратного прокси). См. секцию *Защита по IP* главы *Безопасность*.

Самое большое преимущество этой стратегии кэширования заключается в том, что вы можете делать ваше приложение настолько динамическим, насколько это вам нужно, при этом обращаясь к приложению лишь тогда, когда это необходимо.

Примечание: При использовании ESI, помните, что вам всегда необходимо использовать директиву `s-maxage` вместо `max-age`. Это необходимо, так как браузер получает агрегированный ресурс, следовательно, он не заботится о вложенных компонентах и будет подчиняться директиве `max-age` и кэшировать страницу целиком, чего вы точно не захотите.

Хелпер `render` поддерживает две важных опции:

- `alt`: используется в качестве атрибута `alt` тэга ESI, который позволяет указать альтернативный URL, который будет использован, если `src` не будет найден;
- `ignore_errors`: при значении true, атрибут `onerror` будет добавлен к ESI тагу. Его значение будет равно `continue`, что будет означать удаление ESI тага в случае ошибки на уровне кэширующего шлюза.

2.13.6 Очистка (аннулирование) кэша

> "В науке о компьютерах есть лишь две сложные вещи: аннулирование кэша и вопросы именования." –Phil Karlton

Вы не должны заботиться об аннулировании кэша, так как аннулирование уже заложено в модели кэширования HTTP. Если вы используете модель валидации, вам не нужно ничего аннулировать по определению; если вы используете окончание срока действия и требуется аннулировать ресурс, это означает, что ранее вы для этого ресурса установили срок окончания далеко в будущее.

Примечание: Так как аннулирование кэша - это тема, специфичная для каждого конкретного обратного прокси, если вы специально не побеспокоились об этом - то с лёгкостью сможете переключаться между различными прокси ничего не меняя в коде вашего приложения.

На самом же деле, любой обратный прокси предоставляет способ для очистки кэша, но вы должны стараться избегать этого, насколько возможно. Наиболее типичный путь для очистки кэша для некоторого URL - запросить чего при помощи специального HTTP метода `PURGE`.

Ниже вы увидите как настроить обратный прокси Symfony2 для поддержки HTTP метода `PURGE`:

```
<?php
// 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;
    }
}
```

Осторожно: Вы должны защитить метод `PURGE` каким-либо образом, чтобы не допускать возможности очистки кэша случайными людьми.

2.13.7 Summary

Symfony2 создан таким образом, чтобы следовать проверенным правилам "движения" по дорогам HTTP. Кэширование - не исключение. Настройка системы кэширования Symfony2 подразумевает близкое знакомство с моделью кэширования HTTP и её эффективное использование. Это означает, что вместо того, чтобы полагаться только на документацию Symfony2 и примеры кода, вы получаете доступ к целому миру знаний, относящихся к кэшированию в HTTP и кэширующим шлюзам, таким как Varnish.

2.13.8 Дополнительная информация в книге рецептов:

- *Как использовать Varnish для ускорения работы сайта*

2.14 Переводы

Термин "интернационализация" отсылает нас к процессу извлечения строк текста и прочих специфичных для конкретной локали объектов из вашего приложения и перемещения их на некоторый уровень абстракции, где эти элементы могут быть переведены и конвертированы на основании локали пользователя (т.е. в зависимости от языка и страны). Для текста это означает, что его надо передавать в специальную функцию, способную переводить текст (или некое "сообщение") на язык пользователя:

```
// этот текст *всегда* будет отображаться на английском
echo 'Hello World';

// текст может быть переведён на язык конечного пользователя или же останется на английском
echo $translator->trans('Hello World');
```

Примечание: Термин *локаль* можно грубо определить как совокупность языка и страны пользователя. Это может быть любая строка, которую ваше приложение сможет использовать для управления переводами и прочими различиями в форматах (например, формат даты или валюты). Рекомендуется использовать стандарт ISO639-1 для языковых кодов, подчерк (_) и затем стандарт ISO3166 для кодов стран (например, получится `fr_FR` для French/France).

В этой главе вы узнаете, как подготовить приложение к поддержке нескольких локалей и как создать переводы для них. В общих чертах процесс имеет несколько стандартных шагов:

1. Подключить и настроить компонент Symfony - `Translation`;
2. Завернуть строки (т.н. "сообщения") в вызовы `Translator`'а;

3. Создать ресурсы перевода для каждой поддерживаемой локали и после перевести все сообщения в приложении;
4. Определить, установить и управлять локалью пользователя при помощи сессии.

2.14.1 Настройка

Переводы обрабатываются сервисом (*service*) `Translator`, который использует локаль пользователя для поиска и отображения переведённого сообщения. Перед тем как его использовать, подключите `Translator` в файле конфигурации:

- *YAML*

```yaml
# app/config/config.yml
framework:
    translator: { fallback: en }
```

- *XML*

```xml
<!-- app/config/config.xml -->
<framework:config>
    <framework:translator fallback="en" />
</framework:config>
```

- *PHP*

```php
<?php
// app/config/config.php
$container->loadFromExtension('framework', array(
    'translator' => array('fallback' => 'en'),
));
```

Опция `fallback` определяет локаль для отката, когда перевод не существует для локали пользователя.

Совет: Когда перевод для локали не существует, переводчик пытается сначала найти перевод для языка (`fr` если локаль `fr_FR`, например). Если это также не удаётся, он ищет перевод, используя локаль отката.

Локаль используемая при переводе хранится в сессии пользователя.

2.14.2 Основы переводов

Перевод текста осуществляется сервисом `translator` (`Symfony\Component\Translation\Translator`). Для перевода текстового блока (называемого "сообщением") используйте метод

:method:‘Symfony\\Component\\Translation\\Translator::trans‘. Предположим, например, что вы переводите простое сообщение внутри контроллера:

```
<?php
// ...
public function indexAction()
{
    $t = $this->get('translator')->trans('Symfony2 is great');

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

При выполнении этого кода, Symfony2 попытается перевести сообщение “Symfony2 is great”, основываясь на локали пользователя. Для этого необходимо указать Symfony2 как необходимо перевести это сообщение при помощи “ресурса для перевода”, который представляет собой набор переведённых сообщений для нужной локали. Этот “словарь” переводов может быть создан в нескольких различных форматах, рекомендуемым же является XLIFF формат:

- *XML*

  ```
  <!-- messages.fr.xliff -->
  <?xml version="1.0"?>
  <xliff version="1.2" xmlns="urn:oasis:names:tc:xliff:document:1.2">
      <file source-language="en" datatype="plaintext" original="file.ext">
          <body>
              <trans-unit id="1">
                  <source>Symfony2 is great</source>
                  <target>J'aime Symfony2</target>
              </trans-unit>
          </body>
      </file>
  </xliff>
  ```

- *PHP*

  ```
  <?php
  // messages.fr.php
  return array(
      'Symfony2 is great' => 'J\'aime Symfony2',
  );
  ```

- *YAML*

  ```
  # messages.fr.yml
  Symfony2 is great: J'aime Symfony2
  ```

Теперь, если локалью пользователя будет Французская (например, `fr_FR` или `fr_BE`), это сообщение будет переведено как `J'aime Symfony2`.

Процесс перевода

Для того чтобы перевести сообщение, Symfony2 использует простой процесс:

- Определяется `локаль` текущего пользователя, которая хранится в сессии;
- Загружается каталог переводов сообщений из соответствующего ресурса, определяемого локалью (например, `fr_FR`), сообщения, соответствующие локали отката (fallback), также загружаются и добавляются к каталогу, если он ещё не загружен. В конечном итоге получается большой "словарь" с переводами. См. также Каталоги сообщений.
- Если сообщение есть в каталоге, возвращается его перевод. Если же нет, переводчик возвращает оригинал сообщения.

При использовании метода `trans()` Symfony2 ищет строку целиком в подходящем каталоге и возвращает его (если есть что возвращать).

Заполнители в сообщениях

Иногда, сообщение, которое нужно перевести, содержит переменную:

```
<?php
// ...
public function indexAction($name)
{
    $t = $this->get('translator')->trans('Hello '.$name);

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

Тем не менее, создание перевода для этой строки невозможно, так как переводчик будет искать строку целиком, включая переменную (например, "Hello Ryan" или "Hello Fabien"). Вместо того, чтобы писать переводы для каждого возможного значения переменной `$name`, мы можем заменить переменную "заполнителем" (aka "placeholder"):

```
<?php
// ...
public function indexAction($name)
{
    $t = $this->get('translator')->trans('Hello %name%', array('%name%' => $name));

    new Response($t);
}
```

Symfony2 теперь будет искать перевод оригинала с заполнителем (`Hello %name%`) и *лишь затем* заменять заполнитель его реальным значением. Создание перевода не будет от того, что вы делали ранее:

- *XML*

```
<!-- messages.fr.xliff -->
<?xml version="1.0"?>
<xliff version="1.2" xmlns="urn:oasis:names:tc:xliff:document:1.2">
    <file source-language="en" datatype="plaintext" original="file.ext">
        <body>
            <trans-unit id="1">
                <source>Hello %name%</source>
                <target>Bonjour %name%</target>
            </trans-unit>
        </body>
    </file>
</xliff>
```

- *PHP*

```
<?php
// messages.fr.php
return array(
    'Hello %name%' => 'Bonjour %name%',
);
```

- *YAML*

```
# messages.fr.yml
'Hello %name%': Hello %name%
```

Примечание: Заполнители могут иметь любую форму, так как полное сообщение восстанавливается с использованием PHP-функции strtr function. Тем не менее, нотация `%var%` необходима для использовании шаблонов Twig и, в конечном итоге, более читабельна.

Как вы могли видеть, процесс создания перевода состоит из двух шагов:

1. Извлечение сообщения, которое нужно перевести, передав его в `Translator`.
2. Создание перевода сообщения для каждой локали, которую вы собираетесь поддерживать.

Второй шаг выполняется посредством создания каталогов сообщений, которые содержат переводы для любого количества локалей.

2.14.3 Каталоги сообщений

Когда сообщение переводится, Symfony2 собирает каталог сообщений для локали пользователя и ищет в нём его перевод. Каталог сообщений схож со словарём переводов

для некоторой локали. Например, каталог для локали `fr_FR` может содержать такой перевод:

Symfony2 is Great => J'aime Symfony2

Обязанностью разработчика (или переводчика) интернационализированного приложения является создание таких переводов. Переводы хранятся в файловой системе и обнаруживаются Symfony благодаря некоторым соглашениям.

Совет: Каждый раз, когда вы создаёте *новый* ресурс переводов (или устанавливаете пакет, который включает переводы), убедитесь, что вы очистили кэш, чтобы Symfony смог найти новые ресурсы для перевода:

```
php app/console cache:clear
```

Переводы: расположение в проекте и соглашения по именованию

Symfony2 ищет файлы сообщений (т.е. переводы) в двух местах:

- Для сообщений внутри пакета, файлы сообщений должны быть распложены в директории `Resources/translations/`;
- Для переопределений переводов любого пакета, разместите файлы в директории `app/Resources/translations`.

Наименование файлов переводов также важно, так как Symfony2 использует соглашение по определению деталей перевода. Каждый файл сообщений должен быть назван в соответствии со следующим шаблоном: `domain.locale.loader`:

- **domain**: Не обязательный путь для структурирования сообщений в группы (например, `admin`, `navigation` или же по умолчанию `messages`) - см. Использование доменов сообщений
- **locale**: Локаль, которой соответствует перевод (например, `en_GB`, `en`, и т.д.);
- **loader**: Как Symfony2 должен загрузить и парсить файл (например, `xliff`, `php` или `yml`).

Loader может быть наименованием любого зарегистрированного загрузчика. По умолчанию в Symfony представлены следующие загрузчики:

- `xliff`: XLIFF файл;
- `php`: PHP файл;
- `yml`: YAML файл.

Выбор загрузчика, который будет использован, зависит целиком от вас и по сути это вопрос вкуса.

Примечание: Вы также можете хранить переводы в базе данных, или любом другом хранилище при помощи вашего собственного класса, реализующего интерфейс `Symfony\Component\Translation\Loader\LoaderInterface`. См. статью в книге рецептов: `Пользовательские загрузчики переводов`.

Создание переводов

Каждый файл содержит набор пар "id-translation" для заданного домена и локали. Id - это идентификатор единичного перевода и может быть как сообщением на языка базовой локали (например, "Symfony is great") или же некоторым уникальным идентификатором (например, "symfony2.great" - ниже мы ещё скажем об этом пару слов):

- *XML*

```
<!-- src/Acme/DemoBundle/Resources/translations/messages.fr.xliff -->
<?xml version="1.0"?>
<xliff version="1.2" xmlns="urn:oasis:names:tc:xliff:document:1.2">
    <file source-language="en" datatype="plaintext" original="file.ext">
        <body>
            <trans-unit id="1">
                <source>Symfony2 is great</source>
                <target>J'aime Symfony2</target>
            </trans-unit>
            <trans-unit id="2">
                <source>symfony2.great</source>
                <target>J'aime Symfony2</target>
            </trans-unit>
        </body>
    </file>
</xliff>
```

- *PHP*

```
<?php
// src/Acme/DemoBundle/Resources/translations/messages.fr.php
return array(
    'Symfony2 is great' => 'J\'aime Symfony2',
    'symfony2.great'    => 'J\'aime Symfony2',
);
```

- *YAML*

```
# src/Acme/DemoBundle/Resources/translations/messages.fr.yml
Symfony2 is great: J'aime Symfony2
symfony2.great:    J'aime Symfony2
```

Symfony2 будет находить эти файлы и использовать их при переводе как “Symfony2 is great”, так и “symfony2.great” при использовании французской локали (`fr_FR` или `fr_BE`).

Обычные фразы VS ключевые слова в файлах сообщений

Этот пример иллюстрирует два различных подхода по созданию переводимых сообщений:

```
$t = $translator->trans('Symfony2 is great');

$t = $translator->trans('symfony2.great');
```

При использовании первого метода, сообщение пишется на языке локали по умолчанию (в данном случае это английский). Это же сообщение затем используется как "id" при создании переводов.
При использовании второго метода, сообщение это некое "ключевое слово", которое передаёт идею сообщения. Ключевое слово для сообщения затем используется в качестве "id" в любом переводе. В этом случае, перевод необходимо также сделать и для локали по умолчанию (т.е. перевести `symfony2.great` в `Symfony2 is great`). Второй метод более удобен, так как ключ сообщения не нужно изменять в каждом файле перевода, если вы решите, что сообщение для локали по умолчанию должно выглядеть следующим образом: "Symfony2 is really great".
Выбор того или иного метода - также целиком зависит от вас, но мы бы рекомендовали использовать второй подход с ключевыми словами.
В дополнение к этому, `php` и `yaml` форматы поддерживают вложенные id для того чтобы исключить постоянное повторение при использовании ключевых слов:

- *YAML*

```
symfony2:
    is:
        great: Symfony2 is great
        amazing: Symfony2 is amazing
    has:
        bundles: Symfony2 has bundles
user:
    login: Login
```

- *PHP*

```
<?php
return array(
    'symfony2' => array(
        'is' => array(
            'great' => 'Symfony2 is great',
            'amazing' => 'Symfony2 is amazing',
        ),
        'has' => array(
            'bundles' => 'Symfony2 has bundles',
        ),
    ),
    'user' => array(
        'login' => 'Login',
    ),
);
```


Множественные уровни разделяются при помощи точки (.) между ними, таким образом предыдущий пример соответствует также такому написанию:

- *YAML*

2.14.4 Использование доменов сообщений

Как вы уже видели, файлы сообщений структурированы по различным локалям, которым соответствуют их переводы. Файлы сообщений могут быть также структурированы по “доменам”. При создании файлов сообщений, домен - это первая часть имени файла. Домен по умолчанию - `messages`. Например, предположим, что для лучшей организации файлов переводов они были разделены на три различные домена: `messages`, `admin` и `navigation`. Для французского перевода были созданы следующие файлы сообщений:

- `messages.fr.xliff`
- `admin.fr.xliff`
- `navigation.fr.xliff`

Когда переводится строка не из домена по умолчанию (`messages`), вы явно должны указать домен третьим аргументом функции `trans()`:

```
$this->get('translator')->trans('Symfony2 is great', array(), 'admin');
```

Symfony2 будет теперь искать сообщение в домене `admin`, соответствующем локали пользователя.

2.14.5 Работа с локалью пользователя

Локаль текущего пользователя хранится в сессии и доступна при помощи сервиса `session`:

```
$locale = $this->get('request')->getLocale();

$this->get('request')->setLocale('en_US');
```

Также возможно хранить локаль в сессии:

```
$this->get('session')->set('_locale', 'en_US');
```

Локаль по умолчанию и Локаль для отката

Если локаль в сессии явно не указана, `Translator` будет использовать параметр `fallback_locale`. По умолчанию этот параметр установлен в `en` (см. Настройка).

В качестве альтернативы, вы можете гарантировать, что локаль будет установлена в сессии, если определите параметр `default_locale` для сервиса сессии:

- *YAML*

```yaml
# app/config/config.yml
framework:
    default_locale: en
```

- *XML*

```xml
<!-- app/config/config.xml -->
<framework:config>
    <framework:default-locale>en</framework:default-locale>
</framework:config>
```

- *PHP*

```php
<?php
// app/config/config.php
$container->loadFromExtension('framework', array(
    'default_locale' => 'en',
));
```

Добавлено в версии 2.1.

Локаль и URL

Так как локаль пользователя хранится в сессии, возможно вам захочется использовать один и тот же URL для отображения ресурса на любых других языках, основываясь на локали пользователя. Например, `http://www.example.com/contact` будет отображать контент на английском для одного пользователя, на французском для другого пользователя. К сожалению, это нарушает основополагающее правило Web: каждый URL должен возвращать один и тот же ресурс вне зависимости от пользователя. Для того чтобы усугубить проблему, задумайтесь - какую версию контента должна будет индексироваться поисковиками?

Наилучшим решением является включение локали в URL. Этот метод полностью поддерживается системой маршрутизации при помощи специального параметра `_locale`:

- *YAML*

```yaml
contact:
    pattern:   /{_locale}/contact
    defaults:  { _controller: AcmeDemoBundle:Contact:index, _locale: en }
    requirements:
        _locale: en|fr|de
```

- *XML*

```xml
<route id="contact" pattern="/{_locale}/contact">
    <default key="_controller">AcmeDemoBundle:Contact:index</default>
    <default key="_locale">en</default>
```

```
    <requirement key="_locale">en|fr|de</requirement>
</route>
```

- *PHP*

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

$collection = new RouteCollection();
$collection->add('contact', new Route('/{_locale}/contact', array(
    '_controller' => 'AcmeDemoBundle:Contact:index',
    '_locale'     => 'en',
), array(
    '_locale'     => 'en|fr|de'
)));

return $collection;
```

При использовании в маршруте параметра *_locale*, соответствующая локаль будет *автоматически* установлена в пользовательской сессии. Другими словами, если пользователь посещает URI `/fr/contact`, локаль `fr` будет автоматически установлена для пользователя в сессии.

Теперь вы можете использовать локаль при создании маршрутов к другим переведённым страницам вашего приложения.

2.14.6 Множественное число для сообщений

Множественное число для сообщений - это сложный вопрос, так как правила могут быть очень сложными. Например, ниже представлено математическое представление для множественного числа в русском языке:

```
(($number % 10 == 1) && ($number % 100 != 11)) ? 0 : ((($number % 10 >= 2) && ($number % 10 <= 4
```

Как вы можете видеть, в русском языке имеется три различных формы множественного числа. Для каждой формы множественное число будет другим и поэтому перевод также сложен.

Когда перевод имеет различные формы из-за множественного числа, вы можете предоставить все формы в качестве строки, разделённой вертикальной чертой (`|`):

```
'There is one apple|There are %count% apples'
```

Для того чтобы переводить сообщения с учётом множественного числа, используйте метод:method:*Symfony|Component|Translation|Translator::transChoice*:

```
<?php
// ...
$t = $this->get('translator')->transChoice(
    'There is one apple|There are %count% apples',
    10,
    array('%count%' => 10)
);
```

Второй аргумент (10 в данном примере), это *число* объектов, которое будет использоваться для определения какой именно перевод будет использован, а также будет замещать `%count%`.

Основываясь на этом числе, переводчик выберет правильную форму множественного числа. В английском языке, слова в основном имеют форму единственного числа, когда имеется один объект и форму множественного числа для любого другого числа (0, 1, 2...). Итак, если `count` будет `1`, переводчик будет использовать первую строку (`There is one apple`) в качестве перевода. В противном случае, он будет использовать `There are %count% apples`.

Французский перевод будет таким:

```
'Il y a %count% pomme|Il y a %count% pommes'
```

Даже если эти строки выглядят похожим образом (состоят из двух подстрок, разделённых вертикальной чертой), французское правило отличается: первая форма (единственное число) используется если `count` равен `0` или `1`. Таким образом, переводчик автоматически будет использовать первую строку (`Il y a %count% pomme`), когда `count` будет равен `0` или `1`.

Каждая локаль имеет свой собственный набор правил, некоторые из которых имеют целых шесть различных форм множественного числа со сложными правилами, лежащими в их основе. Правила просты для английского и французского, но в русском языке вы вероятно захотите знать, какое правило соответствует какой строке. Для того чтобы помочь переводчику, вы можете дополнительно добавить таг для каждой строки:

```
'one: There is one apple|some: There are %count% apples'

'none_or_one: Il y a %count% pomme|some: Il y a %count% pommes'
```

Таги являются лишь подсказками для переводчика и не влияют на логику, используемой для определения нужной формы множественного числа. Тагом может быть любая описательная строка, оканчивающаяся двоеточием (`:`). Таги также могут быть различными в оригинальном сообщении и в переводе.

Подробнее о множественности (интервальный метод)

Наиболее простой путь создания множественного числа для сообщения в Symfony2 - использовать встроенную логику для выбора строки на основе данного номера. Иногда вам может потребоваться более полный контроль над переводом множественных чисел или же в особых случаях требуется не стандартный перевод (для числа `0` или же для отрицательных чисел, к примеру). Для таких случаев вы можете использовать интервалы:

```
'{0} There are no apples|{1} There is one apple|]1,19] There are %count% apples|[20,Inf] There a
```

Эти интервалы следуют нотации ISO 31-11. Строка выше определяет четыре различных интервала: точно `0`, точно `1`, `2-19`, а также `20` и более.

Вы также можете комбинировать явные правила и стандартные правила. В этом случае, если число не соответствует указанным интервалам, будет использовано стандартное правило:

```
'{0} There are no apples|[20,Inf] There are many apples|There is one apple|a_few: There are %cou
```

Например, для одного яблока будет использовано стандартное правило `There is one apple`. Для `2-19` - будет использовано второе стандартное правило `There are %count% apples`.

Класс `Symfony\Component\Translation\Interval` может представлять конечный набор чисел:

```
{1,2,3,4}
```

Или же число в интервале между двумя числами:

```
[1, +Inf[
]-1,2[
```

Левая часть разделителя может быть `[` (включая) или `]` (исключая). Правая часть может быть `[` (исключая) or `]` (включая). Для бесконечности вы можете использовать `-Inf` и `+Inf`.

2.14.7 Переводы в шаблонах

Основную часть времени, переводы появляются в шаблонах. Symfony2 предоставляет поддержку переводов как для Twig так и для PHP шаблонов.

Twig шаблоны

Symfony2 предоставляет специализированные таги для Twig (`trans` и `transchoice`) для того чтобы помочь с переводом *статических блоков текста*:

```
{% trans %}Hello %name%{% endtrans %}

{% transchoice count %}
    {0} There are no apples|{1} There is one apple|]1,Inf] There are %count% apples
{% endtranschoice %}
```

Таг `transchoice` автоматически получает переменную `%count%` из контекста и передаёт её переводчику. Этот механизм работает лишь когда вы используете заполнитель в стиле `%var%`.

Совет: Если вам нужно использовать символ процента (`%`) в строке, экранируйте его при помощи дублирования: `{% trans %}Percent: %percent%%%{% endtrans %}`

Вы также можете указать домен для сообщений и передать некоторые дополнительные переменные:

```
{% 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 are no apples|{1} There is one apple|]1,Inf] There are %count% apples
{% endtranschoice %}
```

Фильтры `trans` и `transchoice` могут быть использованы для перевода *текста переменных* и сложных выражений:

```
{{ message | trans }}

{{ message | transchoice(5) }}

{{ message | trans({'%name%': 'Fabien'}, "app") }}

{{ message | transchoice(5, {'%name%': 'Fabien'}, 'app') }}
```

Совет: Использование тагов или фильтров для перевода имеет один и тот же эффект, но с одним небольшим отличием: автоматическое экранирование вывода применяется только к переменным, переведённым при помощи фильтра. Другими словами, если вам нужно быть уверенными, что ваша переменная *не* экранирована, вы должны применять фильтр `raw` после фильтра `trans`:

```
{# текст между тагами никогда не будет экранирован #}
{% trans %}
    <h3>foo</h3>
{% endtrans %}
```

```
{% set message = '<h3>foo</h3>' %}

{# переменная переведённая при помощи фильтра экранирована по умолчанию #}
{{ message | trans | raw }}

{# но статическая строка никогда не экранируется #}
{{ '<h3>foo</h3>' | trans }}
```

PHP Шаблоны

Сервис-переводчик доступен в PHP шаблонах при помощи хелпера `translator`:

```
<?php echo $view['translator']->trans('Symfony2 is great') ?>

<?php echo $view['translator']->transChoice(
    '{0} There are no apples|{1} There is one apple|]1,Inf[ There are %count% apples',
    10,
    array('%count%' => 10)
) ?>
```

2.14.8 Форсирование локали переводчика

Когда переводится сообщение, Symfony2 использует локаль из сессии пользователя или же `fallback` локаль, если требуется. Вы также можете вручную указать локаль для перевода:

```
$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',
);
```

2.14.9 Перевод контента из базы данных

Перевод контента из базы данных должен обрабатываться Doctrine при помощи Translatable Extension. Информацию об этой библиотеке вы можете найти в её документации.

2.14.10 Заключение

При помощи компонента Translation, создание интернациональных приложений больше не требует болезненного процесса и может быть достигнуто при помощи следующих шагов:

- Извлеките сообщения вашего приложения, завернув каждое в методы **:method:‘Symfony\\Component\\Translation\\Translator::trans‘** или **:method:‘Symfony\\Component\\Translation\\Translator::transChoice‘**;
- Переведите каждое сообщение для различных локалей, создав файлы переводов. Symfony2 найдёт и обработает каждый файл так как их имена следуют специфическим соглашениям;
- Управляйте локалью пользователя, которая хранится в сессии.

2.15 Контейнер служб

В современном PHP приложении множество объектов. Один объект может облегчать отправку email'ов, другой - сохранять информацию в базе данных. В вашем приложении вы можете создать объект, который ведёт учёт товаров, или же объект, который обрабатывает данные от сторонних API. Здесь важно понимание того, что современное приложение выполняет множество функций и состоит из множества объектов, реализующих эти функции.

В этой главе мы поговорим об особом объекте в Symfony2, который позволяет вам создавать экземпляры, систематизировать и получать различные объекты вашего приложения. Этот объект, называемый контейнером служб, позволит вам стандартизировать и централизовать способ создания объектов в вашем приложении. Контейнер делает вашу жизнь проще, быстрее и делает особый акцент на архитектуре, которая предоставляет независимый, готовый к повторному использованию код. Так как все классы ядра Symfony2 используют контейнер, вы узнаете, как расширять, настраивать и использовать любой объект Symfony2. Контейнер служб в значительной степени определяет скорость и расширяемость Symfony2.

И, в конце концов, настройка и использование контейнера служб весьма проста. В конце этой главы вы будете с лёгкостью создавать ваши собственные объекты при помощи

контейнера и настраивать объекты из сторонних пакетов. Вы будете писать более тестируемый и менее запутанный код, который можно будет использовать повторно лишь потому, что контейнер служб делает написание хорошего кода простым.

2.15.1 Что такое служба?

Если вкратце, *Служба* - это некий PHP объект, который выполняет какую-либо "глобальную" задачу. Это наименование используется в компьютерной науке для описания объекта, который создан с некоторой целью (например, отправлять email'ы). Каждая служба используется в любом модуле приложения, где бы вам ни понадобился функционал, который предоставляет служба. Вам не требуется делать ничего особенного, для того чтобы создать службу: просто создайте PHP класс, решающий некую конкретную задачу. Поздравляем, Вы только что создали службу!

Примечание: Как правило, PHP объект является службой, если он используется в вашем приложении глобально. Единственная служба `Mailer` используется для отправки email сообщений, в то время как множество объектов `Message`, которые содержат эти сообщения, службами *не* являются. Точно так же, объект `Product` не является службой, но объект, который сохраняет `Product` в базу данных - *является* службой.

Так где же выгода? Мыслить в терминах "служб" полезно, когда вы начинаете думать о распределении каждого кусочка функционала в вашем приложении по ряду служб. Так как каждая служба выполняет единственную функцию, вы можете легко получить доступ к любой службе и использовать её возможности там, где это требуется. Каждая служба легко тестируется и настраивается, так как она не зависит от прочего функционала. Такое разделение на службы называется Сервис-ориентированная архитектура и она не уникальна ни в рамках Symfony2, ни даже в масштабе всего PHP. Структурирование вашего приложения в виде набора независимых служб - это хорошо известная, а также хорошо зарекомендовавшая себя практика. Знание этой архитектуры будет полезно любому хорошему разработчику вне зависимости от языка, на котором он программирует.

2.15.2 Что такое контейнер служб?

Контейнер служб (или же *контейнер внедрения зависимости*) - это также PHP объект, который управляет созданием служб (т.е. объектов). Например, положим у вас есть простой PHP класс, который отправляет email сообщения. Не имея контейнера служб, вы будете вынуждены вручную создавать этот объект там где вам это потребуется:

```
<?php
use Acme\HelloBundle\Mailer;
```

```
$mailer = new Mailer('sendmail');
$mailer->send('ryan@foobar.net', ... );
```

Выглядит не сложно. Ваш воображаемый класс `Mailer` позволяет указать метод, используемый для отправки сообщений (например, `sendmail`, `smtp` и т.д.). Но что будет, если потребуется использовать эту службу где-то ещё? Естественно вам не захочется конфигурировать объект `Mailer` *каждый раз*, когда вам потребуется его использовать. Что, если вам потребуется изменить транспорт с `sendmail` на `smtp` во всём вашем приложении? Потребовалось бы искать все места, где создаётся `Mailer` и изменять их.

2.15.3 Создание/настройка служб в контейнере

Наилучшим решением на практике - разрешить контейнеру служб создать объект `Mailer` для вас. Для этого контейнер необходимо *обучить* - как создавать объект `Mailer`. Это выполняется при помощи конфигурации, которую можно выполнить в форматах YAML, XML или PHP:

- *YAML*

```
# app/config/config.yml
services:
    my_mailer:
        class:        Acme\HelloBundle\Mailer
        arguments:    [sendmail]
```

- *XML*

```
<!-- app/config/config.xml -->
<services>
    <service id="my_mailer" class="Acme\HelloBundle\Mailer">
        <argument>sendmail</argument>
    </service>
</services>
```

- *PHP*

```
<?php
// app/config/config.php
use Symfony\Component\DependencyInjection\Definition;

$container->setDefinition('my_mailer', new Definition(
    'Acme\HelloBundle\Mailer',
    array('sendmail')
));
```

Примечание: При инициализации Symfony2 он создаёт контейнер служб, используя конфигурацию приложения (по умолчанию `app/config/config.yml`). Файл, который будет загружен, определяется методом `AppKernel::registerContainerConfiguration()`, который загружает файл, относящийся к конкретному окружению (например, `config_dev.yml` для `dev` или же `config_prod.yml` для `prod`).

Экземпляр объекта `Acme\HelloBundle\Mailer` теперь можно получить через контейнер служб. А сам контейнер доступен в любом традиционном контроллере Symfony2 при помощи вспомогательного метода `get()`:

```
<?php
class HelloController extends Controller
{
    // ...

    public function sendEmailAction()
    {
        // ...
        $mailer = $this->get('my_mailer');
        $mailer->send('ryan@foobar.net', ... );
    }
}
```

Когда запрашивается служба `my_mailer`, контейнер создаёт её объект и возвращает её. Это ещё одно преимущество от использования контейнера служб. А именно, служба не создаётся вплоть до того момента, когда она будет нужна вам. Если вы определите службу, но нигде её не используете - она никогда не будет создана. Это экономит память и делает ваше приложение быстрее. Это также означает, что вы можете определять сколько угодно служб без ущерба быстродействию приложения - службы, которые не используются - не будут и созданы.

В качестве приятного бонуса, служба `Mailer` будет создана лишь однажды и один и тот же её экземпляр будет возвращаться всякий раз, когда вы запрашиваете данную службу. Как правило, вы именно этого и ожидаете (такой подход более гибок и удобен), однако ниже вы узнаете как настроить службу таким образом, чтобы она могла иметь несколько экземпляров одновременно.

2.15.4 Параметры службы

Создание новой службы (т.е. объекта) при помощи контейнера происходит очень просто. Параметры делают службы более гибкими:

- *YAML*

```
# app/config/config.yml
parameters:
    my_mailer.class:      Acme\HelloBundle\Mailer
    my_mailer.transport:  sendmail

services:
    my_mailer:
        class:        %my_mailer.class%
        arguments:    [%my_mailer.transport%]
```

- *XML*

```
<!-- app/config/config.xml -->
<parameters>
    <parameter key="my_mailer.class">Acme\HelloBundle\Mailer</parameter>
    <parameter key="my_mailer.transport">sendmail</parameter>
</parameters>

<services>
    <service id="my_mailer" class="%my_mailer.class%">
        <argument>%my_mailer.transport%</argument>
    </service>
</services>
```

- *PHP*

```
<?php
// app/config/config.php
use Symfony\Component\DependencyInjection\Definition;

$container->setParameter('my_mailer.class', 'Acme\HelloBundle\Mailer');
$container->setParameter('my_mailer.transport', 'sendmail');

$container->setDefinition('my_mailer', new Definition(
    '%my_mailer.class%',
    array('%my_mailer.transport%')
));
```

Конечный результат такой же, как и раньше - отличие лишь в том, *как* определена служба. Заключив строки `my_mailer.class` и `my_mailer.transport` между символами процента (%), вы указали контейнеру искать параметры с этими именами. Когда контейнер создан, он ищет значение для каждого параметра и использует их при создании служб.

Назначение параметров - передача информации внутрь служб. Конечно же, нет ничего зазорного в создании служб без параметров, тем не менее параметры дают некоторые преимущества:

- разделение и структурирование всех опций службы;
- значения параметров могут быть использованы для множественного определения служб;
- при создании службы в пакете (об этом будет чуть ниже), использование параметров позволяет легко настроить службу в вашем приложении.

Выбор - использовать или не использовать параметры - целиком лежит на вас. Качественные пакеты от сторонних разработчиков *всегда* используют параметры, так как они делают службы более конфигурируемыми. Для служб в вашем приложении, тем не менее, вам может и не потребоваться та гибкость, которую даёт использование параметров.

Массивы параметров

Параметры - это не обязательно строки, это также могут быть и массивы. В формате XML вы должны использовать атрибут type=”collection” для параметров-массивов.

- *YAML*

```yaml
# app/config/config.yml
parameters:
    my_mailer.gateways:
        - mail1
        - mail2
        - mail3
    my_multilang.language_fallback:
        en:
            - en
            - fr
        fr:
            - fr
            - en
```

- *XML*

```xml
<!-- app/config/config.xml -->
<parameters>
    <parameter key="my_mailer.gateways" type="collection">
        <parameter>mail1</parameter>
        <parameter>mail2</parameter>
        <parameter>mail3</parameter>
    </parameter>
    <parameter key="my_multilang.language_fallback" type="collection">
        <parameter key="en" type="collection">
            <parameter>en</parameter>
            <parameter>fr</parameter>
```

```
            </parameter>
            <parameter key="fr" type="collection">
                <parameter>fr</parameter>
                <parameter>en</parameter>
            </parameter>
        </parameter>
    </parameters>
```

- *PHP*

```
<?php
// app/config/config.php
use Symfony\Component\DependencyInjection\Definition;

$container->setParameter('my_mailer.gateways', array('mail1', 'mail2', 'mail3'));
$container->setParameter('my_multilang.language_fallback',
                         array('en' => array('en', 'fr'),
                               'fr' => array('fr', 'en'),
                        ));
```

2.15.5 Импорт конфигураций контейнера

Совет: В этом разделе мы будем ссылаться на файлы конфигурации служб, как на некоторые *ресурсы*. Это подчёркивает тот факт, что, хотя большинство ресурсов конфигурации будут представлены в виде файлов (например, YAML, XML, PHP), Symfony2 настолько гибок, что конфигурация может быть загружена практически отовсюду (например, из базы данных или даже через внешний веб-сервис).

Контейнер служб создаётся с использованием одного конфигурационного ресурса (по умолчанию `app/config/config.yml`). Все прочие ресурсы для служб (включая ядро Symfony2 и настройки сторонних пакетов) должны импортироваться тем или иным способом. Это даёт вам абсолютную гибкость в настройке служб в вашем приложении.

Настройки служб могут быть импортированы двумя различными способами. Во-первых, мы поговорим о методе, который вы будете в основном использовать в вашем приложении: директива `imports`. В следующей секции мы рассмотрим другой, более гибкий метод, наиболее предпочтительный для импорта настроек служб из сторонних приложений.

Импорт конфигурации при помощи директивы `imports`

Ранее вы разместили определение службы `my_mailer` напрямую в файле конфигурации приложения (`app/config/config.yml`). Поскольку класс `Mailer` располагается в пакете

AcmeHelloBundle, имеет смысл разместить определение контейнера my_mailer внутри этого пакета.

Во-первых, переместите определение my_mailer в новый файл внутри AcmeHelloBundle. Если директории Resources или Resources/config отсутствуют - создайте их.

- *YAML*

```
# src/Acme/HelloBundle/Resources/config/services.yml
parameters:
    my_mailer.class:      Acme\HelloBundle\Mailer
    my_mailer.transport:  sendmail

services:
    my_mailer:
        class:        %my_mailer.class%
        arguments:    [%my_mailer.transport%]
```

- *XML*

```
<!-- src/Acme/HelloBundle/Resources/config/services.xml -->
<parameters>
    <parameter key="my_mailer.class">Acme\HelloBundle\Mailer</parameter>
    <parameter key="my_mailer.transport">sendmail</parameter>
</parameters>

<services>
    <service id="my_mailer" class="%my_mailer.class%">
        <argument>%my_mailer.transport%</argument>
    </service>
</services>
```

- *PHP*

```
<?php
// src/Acme/HelloBundle/Resources/config/services.php
use Symfony\Component\DependencyInjection\Definition;

$container->setParameter('my_mailer.class', 'Acme\HelloBundle\Mailer');
$container->setParameter('my_mailer.transport', 'sendmail');

$container->setDefinition('my_mailer', new Definition(
    '%my_mailer.class%',
    array('%my_mailer.transport%')
));
```

Само определение службы осталось без изменений, изменилось только расположение файла конфигурации. Конечно же, контейнер пока ничего не знает о новом ресурсе. К счастью, вы можете легко импортировать файл ресурса при помощи ключевого слова

`imports` в конфигурации приложения:

- *YAML*

```
# app/config/config.yml
imports:
    hello_bundle:
        resource: @AcmeHelloBundle/Resources/config/services.yml
```

- *XML*

```
<!-- app/config/config.xml -->
<imports>
    <import resource="@AcmeHelloBundle/Resources/config/services.xml"/>
</imports>
```

- *PHP*

```
<?php
// app/config/config.php
$this->import('@AcmeHelloBundle/Resources/config/services.php');
```

Директива `imports` позволяет приложению подключать ресурсы конфигурации контейнера служб из различных мест (как правило, из пакетов). Расположение `resource` для файлов - это абсолютный путь к этому файлу. Специальный синтаксис `@AcmeHello` соответствует пути к директории пакета `AcmeHelloBundle`. Это помогает указывать путь к ресурсу не заботясь о том, что пакет `AcmeHelloBundle` может быть в будущем перемещён в другое место.

Импорт конфигурации при помощи расширений контейнера

При разработке с использованием Symfony2 чаще всего вы будете использовать директиву `imports` для импорта конфигурации контейнера из пакетов, которые вы создали специально для вашего приложения. Конфигурация контейнера для сторонних пакетов, включая службы ядра Symfony2, как правило, загружается при помощи другого метода, более гибкого и просто для настройки в вашем приложении.

Вот как работает этот метод. Внутри каждого пакета его службы определяются очень похожим образом, как вы делали это ранее. А именно, пакет использует один или более файлов конфигурации (как правило, XML) для указания параметров и служб этого пакета. Тем не менее, вместо того, чтобы импортировать каждый ресурс непосредственно в файл конфигурации вашего приложения при помощи директивы `imports`, вы можете вызвать *расширение контейнера служб* внутри пакета, которое выполнит эту работу за вас. Расширение контейнера служб - это PHP класс, созданный автором пакета для выполнения следующих функций:

- Импорта всех ресурсов контейнера служб, необходимых для конфигурации всех служб пакетов;

- Предоставления простой и понятной конфигурации, при помощи которой пакет можно настроить, не взаимодействуя напрямую с конфигурацией контейнера служб пакета.

Другими словами, расширение контейнера служб настраивает службы пакета от вашего имени. И, как вы скоро увидите, расширение предоставляет удобный высокоуровневый интерфейс для настройки пакета.

Возьмём в качестве примера `FrameworkBundle` - основу Symfony2. Наличие следующего кода в конфигурации вашего приложения вызывает расширение контейнера служб внутри `FrameworkBundle`:

- *YAML*

```yaml
# app/config/config.yml
framework:
    secret:          xxxxxxxxxx
    charset:         UTF-8
    form:            true
    csrf_protection: true
    router:        { resource: "%kernel.root_dir%/config/routing.yml" }
    # ...
```

- *XML*

```xml
<!-- app/config/config.xml -->
<framework:config charset="UTF-8" secret="xxxxxxxxxx">
    <framework:form />
    <framework:csrf-protection />
    <framework:router resource="%kernel.root_dir%/config/routing.xml" />
    <!-- ... -->
</framework>
```

- *PHP*

```php
<?php
// app/config/config.php
$container->loadFromExtension('framework', array(
    'secret'          => 'xxxxxxxxxx',
    'charset'         => 'UTF-8',
    'form'            => array(),
    'csrf-protection' => array(),
    'router'          => array('resource' => '%kernel.root_dir%/config/routing.php'),
    // ...
));
```

Когда парсится конфигурационный файл, контейнер ищет расширение, которое может обработать директиву `framework`. Вызывается расширение, которое находится в `FrameworkBundle` и загружается конфигурация служб для `FrameworkBundle`. Если вы удалите ключ `framework` из конфигурации приложения - основные сервисы ядра

Symfony2 не будут загружены. Суть же заключается в том, что всё находится под вашим контролем: Symfony2 не содержит никакой магии и не делает ничего такого, чего бы вы не контролировали.

Конечно же, вы можете делать много больше, чем просто активировать расширение контейнера служб для `FrameworkBundle`. Каждое расширение позволяет вам легко настроить пакет, не заботясь о том, как именно определены его службы.

В нашем случае, расширение позволяет настроить `charset`, `error_handler`, `csrf_protection`, `router` и многое другое. Внутри `FrameworkBundle` использует опции, указанные в настройках приложения для определения и конфигурирования своих служб. Пакет позаботится о создании всех необходимых настроек `parameters` и `services` для контейнера служб, при этом также сохраняя гибкость настройки. В качестве бонуса большинство расширений контейнера служб также выполняют валидацию - уведомляют вас об отсутствующих или же имеющих неправильный тип параметрах.

Когда вы устанавливаете или настраиваете пакет - смотрите его документацию, чтобы узнать как установить и настроить его службы. Опции, доступные для основных пакетов ядра вы можете посмотреть в *справочнике*.

Примечание: Контейнер служб распознаёт лишь директивы `parameters`, `services`, и `imports`. Все остальные директивы обрабатываются расширениями.

2.15.6 Использование одних служб внутри других (Внедрение служб)

Рассмотренная выше служба `my_mailer` проста: она принимает лишь один аргумент конструктора, который легко настраивается. Как вы увидите, свою силу контейнер показывает, когда вам нужно создать службу, которая зависит от одной или нескольких служб контейнера.

Давайте начнём с примера. Предположим у вас есть новая служба `NewsletterManager`, которая помогает подготавливать и рассылать email-сообщения на некоторый набор адресов. Службу `my_mailer` было бы неплохо использовать для отправки сообщений внутри службы `NewsletterManager`. Таким образом, класс может выглядеть примерно так:

```
<?php
namespace Acme\HelloBundle\Newsletter;

use Acme\HelloBundle\Mailer;

class NewsletterManager
{
    protected $mailer;
```

```
    public function __construct(Mailer $mailer)
    {
        $this->mailer = $mailer;
    }

    // ...
}
```

Не используя контейнер служб, мы может создать `NewsletterManager` внутри контроллера:

```
<?php
public function sendNewsletterAction()
{
    $mailer = $this->get('my_mailer');
    $newsletter = new Acme\HelloBundle\Newsletter\NewsletterManager($mailer);
    // ...
}
```

Такой подход, в общем-то, не плох, но что, если потребуется добавить к конструктору класса `NewsletterManager` второй или даже третий аргумент? Что, если вы решите выполнить рефакторинг и переименуете класс? В обоих случаях вам потребовалось бы найти все места, где создаются экземпляры `NewsletterManager` и изменить их. И тут контейнер служб предоставляет вам удобное решение:

- *YAML*

  ```
  # src/Acme/HelloBundle/Resources/config/services.yml
  parameters:
      # ...
      newsletter_manager.class: Acme\HelloBundle\Newsletter\NewsletterManager

  services:
      my_mailer:
          # ...
      newsletter_manager:
          class:     %newsletter_manager.class%
          arguments: [@my_mailer]
  ```

- *XML*

  ```
  <!-- src/Acme/HelloBundle/Resources/config/services.xml -->
  <parameters>
      <!-- ... -->
      <parameter key="newsletter_manager.class">Acme\HelloBundle\Newsletter\NewsletterManager
  </parameters>

  <services>
  ```

```
        <service id="my_mailer" ... >
          <!-- ... -->
        </service>
        <service id="newsletter_manager" class="%newsletter_manager.class%">
            <argument type="service" id="my_mailer"/>
        </service>
    </services>
```

- *PHP*

```
<?php
// src/Acme/HelloBundle/Resources/config/services.php
use Symfony\Component\DependencyInjection\Definition;
use Symfony\Component\DependencyInjection\Reference;

// ...
$container->setParameter('newsletter_manager.class', 'Acme\HelloBundle\Newsletter\Newslette

$container->setDefinition('my_mailer', ... );
$container->setDefinition('newsletter_manager', new Definition(
    '%newsletter_manager.class%',
    array(new Reference('my_mailer'))
));
```

В YAML, специальный синтаксис `@my_mailer` сообщает контейнеру, что нужно искать службу `my_mailer` и передать этот объект в конструктор `NewsletterManager`. В этом случае служба `my_mailer` должна существовать. Если её определение не будет найдено, будет вызвано исключение. Вы также можете пометить зависимости опциональными - это будет обсуждаться в следующей секции.

Использование ссылок на службы (внедрение служб) - это мощный инструмент, который позволяет создавать независимые классы служб с чётко определёнными зависимостями. В этом примере, службе `newsletter_manager` для функционирования необходима служба `my_mailer`. Когда вы определите эту зависимость в контейнере служб, он позаботится о создании всех необходимых объектов.

Опциональные зависимости

Внедрение зависимостей в конструктор - это прекрасный способ удостовериться, что зависимость доступна для использования. Если у вас есть необязательные зависимости для класса, то лучшим выбором будет использование “setter injection”. Это означает, что внедрение зависимости производится при помощи некоторого метода, а не в конструкторе. Класс будет выглядеть следующим образом:

```
<?php
namespace Acme\HelloBundle\Newsletter;
```

```
use Acme\HelloBundle\Mailer;

class NewsletterManager
{
    protected $mailer;

    public function setMailer(Mailer $mailer)
    {
        $this->mailer = $mailer;
    }

    // ...
}
```

Внедрение зависимости при помощи метода требует также изменения синтаксиса:

- *YAML*

```
# src/Acme/HelloBundle/Resources/config/services.yml
parameters:
    # ...
    newsletter_manager.class: Acme\HelloBundle\Newsletter\NewsletterManager

services:
    my_mailer:
        # ...
    newsletter_manager:
        class:     %newsletter_manager.class%
        calls:
            - [ setMailer, [ @my_mailer ] ]
```

- *XML*

```
<!-- src/Acme/HelloBundle/Resources/config/services.xml -->
<parameters>
    <!-- ... -->
    <parameter key="newsletter_manager.class">Acme\HelloBundle\Newsletter\NewsletterManager
</parameters>

<services>
    <service id="my_mailer" ... >
      <!-- ... -->
    </service>
    <service id="newsletter_manager" class="%newsletter_manager.class%">
        <call method="setMailer">
             <argument type="service" id="my_mailer" />
        </call>
    </service>
```

```
</services>
```

- *PHP*

```
<?php
// src/Acme/HelloBundle/Resources/config/services.php
use Symfony\Component\DependencyInjection\Definition;
use Symfony\Component\DependencyInjection\Reference;

// ...
$container->setParameter('newsletter_manager.class', 'Acme\HelloBundle\Newsletter\Newslette

$container->setDefinition('my_mailer', ... );
$container->setDefinition('newsletter_manager', new Definition(
    '%newsletter_manager.class%'
))->addMethodCall('setMailer', array(
    new Reference('my_mailer')
));
```

Примечание: Подходы к внедрению служб, представленные в этой секции называются "constructor injection" и "setter injection". Контейнер служб Symfony2 также поддерживает "property injection".

2.15.7 Делаем ссылки на службы опциональными

Иногда, службы могут иметь опциональные зависимости, т.е. зависимость не требуется, для того, чтобы служба правильно работала. В примере выше служба `my_mailer` должна существовать, в противном случае будет сгенерирована ошибка (исключение). Изменив определение службы `newsletter_manager` вы можете сделать зависимость необязательной. Контейнер будет внедрять её лишь когда эта зависимость существует, в случае же если она не существует, никаких действий производиться не будет:

- *YAML*

```
# src/Acme/HelloBundle/Resources/config/services.yml
parameters:
    # ...

services:
    newsletter_manager:
        class:     %newsletter_manager.class%
        arguments: [@?my_mailer]
```

- *XML*

```
<!-- src/Acme/HelloBundle/Resources/config/services.xml -->

<services>
    <service id="my_mailer" ... >
      <!-- ... -->
    </service>
    <service id="newsletter_manager" class="%newsletter_manager.class%">
        <argument type="service" id="my_mailer" on-invalid="ignore" />
    </service>
</services>
```

- *PHP*

```
<?php
// src/Acme/HelloBundle/Resources/config/services.php
use Symfony\Component\DependencyInjection\Definition;
use Symfony\Component\DependencyInjection\Reference;
use Symfony\Component\DependencyInjection\ContainerInterface;

// ...
$container->setParameter('newsletter_manager.class', 'Acme\HelloBundle\Newsletter\Newslette

$container->setDefinition('my_mailer', ... );
$container->setDefinition('newsletter_manager', new Definition(
    '%newsletter_manager.class%',
    array(new Reference('my_mailer', ContainerInterface::IGNORE_ON_INVALID_REFERENCE))
));
```

В YAML, специальный синтаксис `@?` сообщает контейнеру служб, что зависимость не обязательная. И, конечно же, конструктор `NewsletterManager` должен быть переписан, чтобы поддерживать опциональную зависимость:

```
<?php
// ...
public function __construct(Mailer $mailer = null)
{
    // ...
}
```

2.15.8 Основные службы Symfony и службы от сторонних разработчиков

Так как Symfony2 и все сторонние пакеты настраивают и получают свои службы при помощи контейнера, вы можете легко получить доступ к ним или даже использовать их в своих собственных службах. Для простоты Symfony2 по умолчанию не требует, чтобы контроллеры были бы определены как службы. Кроме того, Symfony2 внедряет

в ваш контроллер контейнер служб целиком. Например, для работы с пользовательской сессией, Symfony2 предоставляет службу `session`, которая позволяет получить доступ к сессии внутри обычного контроллера:

```
<?php
public function indexAction($bar)
{
    $session = $this->get('session');
    $session->set('foo', $bar);

    // ...
}
```

В Symfony2 вы постоянно будете пользоваться службами, предоставляемыми ядром Symfony или же прочими пакетами от сторонних разработчиков, для выполнения таких задач как отображение шаблонов (`templating`), отправку майлов (`mailer`) или доступ к переменным запроса (`request`).

Вы можете пойти ещё дальше и использовать эти службы внутри ваших служб, созданных для вашего приложения. Дайте изменим класс `NewsletterManager`, чтобы он использовал стандартный `mailer` Symfony2 (вместо `my_mailer`). Давайте также внедрим службу шаблонизатора в `NewsletterManager`, чтобы можно было создавать контент электронных писем из шаблонов:

```
<?php
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;
    }

    // ...
}
```

Настройку контейнера выполнить не сложно:

- *YAML*

```
services:
    newsletter_manager:
        class:     %newsletter_manager.class%
        arguments: [@mailer, @templating]
```

- *XML*

```
<service id="newsletter_manager" class="%newsletter_manager.class%">
    <argument type="service" id="mailer"/>
    <argument type="service" id="templating"/>
</service>
```

- *PHP*

```
<?php
$container->setDefinition('newsletter_manager', new Definition(
    '%newsletter_manager.class%',
    array(
        new Reference('mailer'),
        new Reference('templating')
    )
));
```

Служба `newsletter_manager` теперь имеет доступ к службам `mailer` и `templating`. Это типичный путь по созданию служб, специфичных для вашего приложения, который позволяет использовать всю мощь различных служб Фреймворка.

Совет: Удостоверьтесь, что в конфигурации вашего приложения присутствует раздел `swiftmailer`. Как указано в секции *Импорт конфигурации при помощи расширений контейнера*, ключ `swiftmailer` внедряет расширение из `SwiftmailerBundle`, которое регистрирует службу `mailer`.

2.15.9 Продвинутая конфигурация контейнера

Как вы могли видеть ранее, определение служб в контейнере выполняется легко, в основном с применением ключа конфигурации `service` и нескольких параметров. Тем не менее, контейнер имеет ещё несколько дополнительных инструментов, с помощью которых можно получить дополнительную функциональность, создавать более сложные службы и выполнять операции после создания контейнера.

Публичные и приватные службы

Как правило, при определении служб, вы рассчитываете получить доступ к ним в вашем приложении. Такие службы называются *публичными* (`public`). Например, служба

`doctrine` из состава DoctrineBundle является публичной и вы можете получить к ней доступ следующим образом:

```
$doctrine = $container->get('doctrine');
```

Тем не менее, имеются случаи, когда вы не захотите, чтобы ваши службы были публичными. Как правило, это случается, когда служба определена лишь для того, чтобы быть использованной в качестве аргумента для другой службы.

Примечание: Если вы используете приватные службы в качестве аргумента более чем для одной службы, в результате будут созданы два различных экземпляра этой приватной службы (т.е. `new PrivateFooBar()`).

Служба должна быть приватной, когда вы не хотите, чтобы она была доступна напрямую из кода.

Например:

- *YAML*

  ```
  services:
     foo:
       class: Acme\HelloBundle\Foo
       public: false
  ```

- *XML*

  ```
  <service id="foo" class="Acme\HelloBundle\Foo" public="false" />
  ```

- *PHP*

  ```
  <?php
  $definition = new Definition('Acme\HelloBundle\Foo');
  $definition->setPublic(false);
  $container->setDefinition('foo', $definition);
  ```

Теперь служба определена как приватная и в *не можете* получить к ней доступ напрямую:

```
$container->get('foo');
```

Тем не менее, даже если служба обозначена как приватная, вы ещё можете использовать её псевдоним (alias, см. ниже) для доступа к ней (при помощи этого псевдонима).

Примечание: По умолчанию все службы - публичные.

Псевдонимы

При использовании основных или же сторонних пакетов в вашем приложении, вы возможно захотите использовать ярлычки для доступа к некоторым их службам. Вы можете сделать это при помощи псевдонимов и даже можете создавать псевдонимы для приватных служб.

- *YAML*

```
services:
   foo:
     class: Acme\HelloBundle\Foo
   bar:
     alias: foo
```

- *XML*

```
<service id="foo" class="Acme\HelloBundle\Foo"/>

<service id="bar" alias="foo" />
```

- *PHP*

```
<?php
$definition = new Definition('Acme\HelloBundle\Foo');
$container->setDefinition('foo', $definition);

$containerBuilder->setAlias('bar', 'foo');
```

Это означает, что при использовании контейнера, вы можете получить доступ к службе `foo` запрашивая службу `bar`:

```
$container->get('bar'); // В итоге получите службу foo
```

Подключение файлов

Также возможны случаи, когда вам будет необходимо подключить некоторый файл прямо перед загрузкой службы. Для этого вы можете воспользоваться директивой `file`:

- *YAML*

```
services:
   foo:
     class: Acme\HelloBundle\Foo\Bar
     file: %kernel.root_dir%/src/path/to/file/foo.php
```

- *XML*

```
<service id="foo" class="Acme\HelloBundle\Foo\Bar">
    <file>%kernel.root_dir%/src/path/to/file/foo.php</file>
</service>
```

- *PHP*

```
$definition = new Definition('Acme\HelloBundle\Foo\Bar');
$definition->setFile('%kernel.root_dir%/src/path/to/file/foo.php');
$container->setDefinition('foo', $definition);
```

Имейте в виду, что symfony будет подключать файл при помощи PHP функции `require_once`, поэтому файл будет подключаться один единственный раз в рамках каждого запроса.

Таги (tags)

Точно также как ваша запись в блоге может быть снабжена тагами, так и любая служба может иметь свои таги. В контейнере служб таг означает, что служба используется для некоторых специфических функций. Давайте рассмотрим пример:

- *YAML*

```
services:
    foo.twig.extension:
        class: Acme\HelloBundle\Extension\FooExtension
        tags:
            -  { name: twig.extension }
```

- *XML*

```
<service id="foo.twig.extension" class="Acme\HelloBundle\Extension\FooExtension">
    <tag name="twig.extension" />
</service>
```

- *PHP*

```
<?php
$definition = new Definition('Acme\HelloBundle\Extension\FooExtension');
$definition->addTag('twig.extension');
$container->setDefinition('foo.twig.extension', $definition);
```

Таг `twig.extension` - это специализированный таг, который `TwigBundle` использует во время конфигурирования. Присваивая службе таг `twig.extension`, `TwigBundle` будет знать, что служба `foo.twig.extension` должна быть зарегистрирована в качестве расширения Twig. Другими словами, Twig таким образом ищет все службы с тагом `twig.extension` и автоматически регистрирует их как расширения Twig.

Таги, таким образом, являются способом сообщить Symfony2 или другим сторонним пакетам, что ваша служба должна быть зарегистрирована или использована некоторым особым способом внутри целевого пакета.

Ниже представлен список тагов, доступных в ядре Symfony2. Каждый из этих тагов имеет свой собственный эффект на вашу службу и многие таги требуют наличия дополнительных параметров (помимо параметра `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

2.15.10 Дополнительно читайте в книге рецептов:

- `/cookbook/service_container/factories`
- `/cookbook/service_container/parentservices`
- *Как определять Контроллеры в качестве сервисов*

2.16 Составные части

Похоже, что вы хотите понять, как работает Symfony2 и как его расширить. Это радует! Этот раздел подробно объясняет внутренности Symfony2.

Примечание: Чтение этого раздела необходимо, только если вы хотите понять, как работает Symfony2 за кулисами или если хотите расширять Symfony2.

2.16.1 Обзор

Код Symfony2 сделан из нескольких независимых слоёв. Каждый следующий слой надстраивается на предыдущем.

Совет: Автозагрузка не управляется непосредственно фреймворком; она выполняется независимо с помощью класса `Symfony\Component\ClassLoader\UniversalClassLoader` и файла `src/autoload.php`. За дополнительной информацией обращайтесь к *разделу*, посвящённому этой теме.

Компонент `HttpFoundation`

На самом глубоком уровне находится компонент **:namespace:'Symfony\\Component\\HttpFound** HttpFoundation предоставляет основные объекты, необходимые для работы с HTTP. Это объектно-ориентированная абстракция некоторых встроенных PHP функций и переменных:

- Класс `Symfony\Component\HttpFoundation\Request` абстрагирует основные глобальные переменные в PHP, такие как `$_GET`, `$_POST`, `$_COOKIE`, `$_FILES` и `$_SERVER`;
- Класс `Symfony\Component\HttpFoundation\Response` абстрагирует некоторые PHP функции типа `header()`, `setcookie()` и `echo`;
- Класс `Symfony\Component\HttpFoundation\Session` и `Symfony\Component\HttpFoundation\SessionStorage\SessionStorageInterface` абстрагируют функции `session_*()` для управления сессией.

Компонент `HttpKernel`

Поверх HttpFoundation располагается компонент **:namespace:'Symfony\\Component\\HttpKern** HttpKernel управляет динамической частью HTTP; это тонкая обёртка поверх классов Request и Response, которая приводит способы обработки запросов к стандарту. Компонент также предоставляет точки для расширений и инструменты, делающие его идеальной стартовой площадкой для создания Web фреймворка без лишних проблем.

Также, дополнительно, он добавляет настраиваемость и расширяемость благодаря компоненту Dependency Injection и мощной системе пакетов (Bundles).

См.также:

Узнайте больше о компоненте `HttpKernel`. Узнайте больше о *Dependency Injection* и Пакетах.

Пакет `FrameworkBundle`

:namespace:'Symfony\\Bundle\\FrameworkBundle' это пакет, связывающий основные компоненты и библиотеки вместе, что создаёт лёгкий и быстрый MVC фреймворк. Он поставляется с правильной первоначальной конфигурацией и соглашениями для облегчения изучения.

2.16.2 Ядро (Kernel)

Класс `Symfony\Component\HttpKernel\HttpKernel` - это центральный класс в Symfony2 и он в ответе за обработку клиентских запросов. Его главная цель - "превратить" объект `Symfony\Component\HttpFoundation\Request` в объект `Symfony\Component\HttpFoundation\Response`.

Каждый Symfony2 Kernel наследует `Symfony\Component\HttpKernel\HttpKernelInterface`:

```
function handle(Request $request, $type = self::MASTER_REQUEST, $catch = true)
```

Контроллеры (Controllers)

При преобразования запроса в ответ, Kernel полагается на "Controller". Контроллер может быть любой валидной PHP-сущностью, которую можно вызвать тем или иным образом.

Ядро делегирует право выбора запустить тот или иной контроллер классу, реализующему интерфейс `Symfony\Component\HttpKernel\Controller\ControllerResolverInterface`:

```
public function getController(Request $request);

public function getArguments(Request $request, $controller);
```

Метод **:method:'Symfony\\Component\\HttpKernel\\Controller\\ControllerResolverInterfa** возвращает контроллер (PHP callable - функцию, метод, замыкание...), ассоциированный с данным запросом. Каноническая реализация (`Symfony\Component\HttpKernel\Controller\ControllerResolver`) ищет атрибут запроса `_controller`, который хранит наименование контроллера (строку "class::method", например `Bundle\BlogBundle\PostController:indexAction`).

Совет: Реализация по умолчанию использует `Symfony\Bundle\FrameworkBundle\EventListener\Rou` для определения атрибута `_controller` из запроса (see *Событие kernel.request*).

Метод **:method:'Symfony\\Component\\HttpKernel\\Controller\\ControllerResolverInterfa** возвращает массив аргументов для передачи их в контроллер. Реализация по умолчанию автоматически определяет аргументы, основываясь на атрибутах запроса.

Сопоставление аргументов метода контроллера по атрибутам запроса

Для каждого аргумента метода Symfony2 пытается получить из запроса значение атрибута с таким же именем. Если он не определён, используется значение по умолчанию (если оно также определено):

```
// Symfony2 будет искать обязательный атрибут 'id'
// и опциональный атрибут 'admin'
public function showAction($id, $admin = true)
{
    // ...
}
```

Обработка запросов

Метод `handle()` принимает `Request` и *всегда* возвращает `Response`. При конвертации объекта `Request`, `handle()` полагается на Resolver и упорядоченную цепь нотификаций о событиях (Event notifications, см. следующую секцию для более подробной информации о каждом событии из этой цепи):

1. Перед тем как что-либо делать, срабатывает нотификация о событии `kernel.request` – если один из слушателей (listeners) возвращает объект `Response`, процесс сразу переходит к шагу 8;
2. Вызывается Resolver для определения Контроллера, который необходимо выполнить;
3. Слушатели события `kernel.controller` теперь могут манипулировать методом Контроллера (изменить, обернуть...);
4. Kernel проверяет, что Контроллер представляет собой валидный PHP callable;
5. Для определения аргументов Контроллера вызывается Resolver;
6. Kernel выполняет Контроллер;
7. Если Контроллер не возвращает объект `Response`, слушатели события `kernel.view` могут конвертировать данные, которые вернул Контроллер в объект `Response`;
8. Слушатели события `kernel.response` могут манипулировать объектом `Response` (контент и заголовки);
9. Возвращается Ответ.

Если во время этого процесса возникает исключительная ситуация, срабатывает событие `kernel.exception` и его слушатели получают возможность конвертировать исключение (Exception) в Ответ. Если это удаётся, событие уведомляется, если нет, исключение вызывается повторно.

Если вы не хотите, чтобы возникали исключения (для вложенных запросов, к примеру), отключите событие `kernel.exception` передав `false` в качестве третьего аргумента метода `handle()`.

Внутренние Запросы

В любой момент во время обработки запроса (назовём его 'мастер'), может быть обработан подзапрос. Вы можете передать тип запроса в метод `handle()` его вторым параметром:

- `HttpKernelInterface::MASTER_REQUEST`;
- `HttpKernelInterface::SUB_REQUEST`.

Тип также передаётся во все события, и их слушатели могут действовать в соответствии с переданным типом (некоторые действия могут соответствовать только мастер-запросу).

События

Каждое событие, создаваемое в Kernel, это дочерний класс `Symfony\Component\HttpKernel\Event\KernelEvent`. Это означает, что каждое событие имеет доступ к одной и той же базовой информации:

- `getRequestType()` - возвращает *тип* запроса (`HttpKernelInterface::MASTER_REQUEST` или `HttpKernelInterface::SUB_REQUEST`);
- `getKernel()` - возвращает экземпляр Kernel, обрабатывающий этот запрос;
- `getRequest()` - возвращает объект `Request`, соответствующий обрабатываемому запросу;

`getRequestType()`

Метод `getRequestType()` позволяет слушателям узнавать тип запроса. Например, если слушатель должен быть активен только для мастер-запроса, добавьте следующий код в начало вашего "слушающего" метода:

```
<?php
use Symfony\Component\HttpKernel\HttpKernelInterface;
```

```
if (HttpKernelInterface::MASTER_REQUEST !== $event->getRequestType()) {
    // немедленно возвращаемся
    return;
}
```

Совет: Если вы ещё не знакомы с Диспетчером Событий Symfony2 (Event Dispatcher), прочитайте сначала секцию *События*.

Событие `kernel.request`

Класс события: `Symfony\Component\HttpKernel\Event\GetResponseEvent`

Цель этого события - либо незамедлительно вернуть объект `Response`, или же подготовить переменные, чтобы можно было вызвать контроллер после события. Любой слушатель может вернуть объект `Response` при помощи метода события `setResponse()`. В этом случае, все остальные слушатели не будут вызываться.

Это событие используется в `FrameworkBundle` для заполнения атрибута `_controller` в объекте `Request` при помощи класса `Symfony\Bundle\FrameworkBundle\EventListener\RouterListener`. RequestListener использует объект, реализующий интерфейс `Symfony\Component\Routing\RouterInterface` для согласования объекта `Request` и определения наименования Контроллера (которое хранится в атрибуте `_controller` объекта `Request`).

Событие `kernel.controller`

Класс события: `Symfony\Component\HttpKernel\Event\FilterControllerEvent`

Это событие не используется в `FrameworkBundle`, но оно может быть точкой входа, используемой для модификации исполняемого контроллера:

```
<?php
use Symfony\Component\HttpKernel\Event\FilterControllerEvent;

public function onKernelController(FilterControllerEvent $event)
{
    $controller = $event->getController();
    // ...

    // the controller can be changed to any PHP callable
    $event->setController($controller);
}
```

Событие kernel.view

Класс события: Symfony\Component\HttpKernel\Event\GetResponseForControllerResultEvent

Это событие не используется в FrameworkBundle, но оно может быть использовано для реализации подсистемы view. Это событие вызывается *только* если Контроллер *не* возвращает объект Response. Назначение этого события - разрешить конвертацию возвращаемых значений в объект Response.

Значение, возвращаемое Контроллером доступно при помощи метода getControllerResult:

```
<?php
use Symfony\Component\HttpKernel\Event\GetResponseForControllerResultEvent;
use Symfony\Component\HttpFoundation\Response;

public function onKernelView(GetResponseForControllerResultEvent $event)
{
    $val = $event->getReturnValue();
    $response = new Response();
    // код получения объекта Response из полученного значения

    $event->setResponse($response);
}
```

Событие kernel.response

Класс события: Symfony\Component\HttpKernel\Event\FilterResponseEvent

Назначение этого события - позволить другим системам модифицировать или заменять объект Response после его создания:

```
<?php
public function onKernelResponse(FilterResponseEvent $event)
{
    $response = $event->getResponse();
    // .. modify the response object
}
```

FrameworkBundle регистрирует несколько слушателей:

- Symfony\Component\HttpKernel\EventListener\ProfilerListener: собирает данные для текущего запроса;
- Symfony\Bundle\WebProfilerBundle\EventListener\WebDebugToolbarListener: внедряет Web Debug Toolbar;

- `Symfony\Component\HttpKernel\EventListener\ResponseListener`: устанавливает `Content-Type` ответа, основываясь на формате запроса;
- `Symfony\Component\HttpKernel\EventListener\EsiListener`: добавляет заголовок `Surrogate-Control`, в случае если ответ необходимо парсить на предмет наличия ESI тагов.

Событие `kernel.exception`

Класс события: `Symfony\Component\HttpKernel\Event\GetResponseForExceptionEvent`

`FrameworkBundle` регистрирует `Symfony\Component\HttpKernel\EventListener\ExceptionListener`, который перенаправляет `Request` в указанные Контроллер (определяется значением параметра `exception_listener.controller`, указывается в нотации `class::method`).

Слушатель этого события может создавать объект `Response`, создавать новый объект `Exception` или же ничего не делать:

```
<?php
use Symfony\Component\HttpKernel\Event\GetResponseForExceptionEvent;
use Symfony\Component\HttpFoundation\Response;

public function onKernelException(GetResponseForExceptionEvent $event)
{
    $exception = $event->getException();
    $response = new Response();
    // Настраиваем объект Response, основываясь на перехваченном исключении
    $event->setResponse($response);

    // как вариант - вы можете создать новое исключение
    // $exception = new \Exception('Some special exception');
    // $event->setException($exception);
}
```

2.16.3 Диспетчер событий (Event Dispatcher)

Объектно-ориентированный код прошёл длинный путь по обеспечению расширяемости кода. Путём создания узкоспециализированных классов, ваш код становится более гибким и разработчик может расширять его при помощи дочерних классов, чтобы изменять их поведение. Но что, если требуется использовать его изменения совместно с другими разработчиками, которые также создают свои дочерние классы? Здесь использование наследования уже не столь удобно.

Рассмотрим реальный пример, в котором вам нужно создать систему плагинов для вашего проекта. Плагин должен иметь возможность добавлять методы или же делать

что-то до или после выполнения некоторого метода, не пересекаясь с прочими плагинами. Эту задачу непросто решить при помощи одиночного наследования, да и множественное наследование (если бы оно было возможно в PHP) имеет свои недостатки.

Диспетчер событий Symfony2 реализует шаблон проектирования Observer простым и эффективным способом, позволяя создавать, например, что-то вроде системы плагинов, которую упоминали выше, и делая ваш проект действительно расширяемым.

Рассмотрим ещё один простой пример из Symfony2 HttpKernel component. Когда создаётся объект `Response`, было бы здорово позволить другим системам проекта модифицировать его (например, добавить заголовки для кэширования) перед последующим использованием. Для того, чтобы достичь этого, ядро Symfony2 создаёт событие - `kernel.response`. Вот как это работает:

- *Слушатель* (listener, PHP объект) сообщает центральному *диспетчеру*, что он собирается слушать (ожидать) событие `kernel.response`;
- В какой-то момент ядро Symfony2 просит объект *диспетчера* отправить событие `kernel.response`, и вместе с ним - объект `Response`;
- Диспетчер уведомляет (фактически вызывает метод) всех слушателей события `kernel.response`, позволяя каждому из них выполнить модификацию объекта `Response`.

События

Когда сообщение отправлено, оно идентифицируется по уникальному имени (например, `kernel.response`), которое могут ожидать некоторое число слушателей. Также создаётся экземпляр класса `Symfony\Component\EventDispatcher\Event`, который затем передаётся всем слушателям. Как вы увидите чуть позже, объект `Event` часто содержит данные о направляемом событии.

Соглашения по именованию

Уникальным именем для события может быть любая строка, но желательно следование нескольким простым правилам:

- Допустимые символы: буквы в нижнем регистре, цифры, точка (`.`), подчерк (`_`);
- Добавляйте префикс пространства имён с точкой на конце (например, `kernel.`);
- Оканчивайте имя глаголом, который обозначает действие (например, `request`).

Вот пара примеров хороших имён для событий:

- `kernel.response`
- `form.pre_set_data`

Объекты событий

Когда диспетчер уведомляет слушателей, он передаёт им объект `Event`. Базовый класс `Event` очень прост: он содержит метод для прекращения воспроизведения (*event propagation*) и ничего более.

Зачастую, необходимо передавать в объекте `Event` также данные о событии, чтобы слушатели могли их обработать тем или иным образом. В случае события `kernel.response`, объект `Event`, передаваемый каждому слушателю, фактически имеет тип `Symfony\Component\HttpKernel\Event\FilterResponseEvent`, дочерний по отношению к `Event` класс. Этот класс содержит методы, такие как `getResponse` и `setResponse`, позволяющие слушателям получать и даже заменять объект `Response`.

Мораль этой истории в следующем: при создании слушателя некоторого события, объект `Event`, который будет передан этому слушателю, может быть специализированным дочерним классом и иметь дополнительные методы для получения данных события и их обработки.

Диспетчер

Диспетчер - это центральный объект системы обработки событий. Как правило, создаётся единственный диспетчер, который обслуживает реестр слушателей. Когда событие поступает к диспетчеру - он уведомляет всех слушателей, подписанных на это событие.

```
<?php
use Symfony\Component\EventDispatcher\EventDispatcher;

$dispatcher = new EventDispatcher();
```

Подключаем Слушателей

Для того, чтобы отреагировать на некое существующее событие, вам необходимо подключить слушателя к диспетчеру, чтобы последний имел возможность сообщить о появлении нужного события. Вызов метода диспетчера `addListener()` ассоциирует любую исполнимую функцию/метод с событием:

```
<?php
$listener = new AcmeListener();
$dispatcher->addListener('foo.action', array($listener, 'onFooAction'));
```

Метод `addListener()` получает три аргумента:

- Наименование события, которое слушатель будет ожидать;
- Некий объект (функцию, в общем же случае PHP callable), который будет вызван при наступлении события;

- Опциональный приоритет (чем больше - тем более важный), который определяет очерёдность вызова слушателей (по умолчанию 0). Если два слушателя имеют одинаковый приоритет, они выполняются в порядке их добавления.

Примечание: PHP callable - это переменная, которая может быть использована в функции `call_user_func()` и возвращает `true` при проверке с помощью функции `is_callable()`. Это может быть, в том числе, и экземпляр замыкания (`\Closure`), строка с именем функции или массив, представляющий собой метод объекта или же метод класса.

Ранее вы уже видели как PHP объект может быть зарегистрирован в качестве слушателя. Вы также можете регистрировать Замыкания (Closures) в качестве слушателей:

```
<?php
use Symfony\Component\EventDispatcher\Event;

$dispatcher->addListener('foo.action', function (Event $event) {
    // этот код будет вызван при обработке события foo.action
});
```

Когда слушатель зарегистрирован диспетчером, он ожидает наступления события. В примере выше, когда появляется событие `foo.action`, диспетчер вызывает метод `AcmeListener::onFooAction` и передаёт объекту `Event` один аргумент:

```
<?php
use Symfony\Component\EventDispatcher\Event;

class AcmeListener
{
    // ...

    public function onFooAction(Event $event)
    {
        // do something
    }
}
```

Совет: Если вы используете Symfony2 MVC framework, слушатели могут быть зарегистрированы при помощи *конфигурации*. В качестве бонуса, объект слушателя будет создан лишь когда будет нужен.

Во многих случаях, слушателю передаётся специализированный дочерний класс `Event`. Это даёт слушателю доступ к информации о событии. Сверяйтесь с документацией или реализацией каждого конкретного события для определения какой именно экземпляр `Symfony\Component\EventDispatcher\Event` бу-

дет передан. Например, событие `kernel.event` передаёт экземпляр класса `Symfony\Component\HttpKernel\Event\FilterResponseEvent`:

```
<?php
use Symfony\Component\HttpKernel\Event\FilterResponseEvent

public function onKernelResponse(FilterResponseEvent $event)
{
    $response = $event->getResponse();
    $request = $event->getRequest();

    // ...
}
```

Создание и обработка события

В дополнение к регистрации слушателей для уже существующих событий, вы можете создавать и вызывать свои собственные события. Это может быть удобно при создании сторонних библиотек и если вы хотите чтобы различные компоненты вашей системы были гибкими и независимыми.

Статический класс `Events`

Предположим, вы хотите создать новое событие - `store.order` - которое создаётся всякий раз, когда в вашем приложении создаётся заказ. Для того, чтобы поддерживать порядок в приложении, начнём с создания класса `StoreEvents`, который будет определять ваше событие:

```
<?php
namespace Acme\StoreBundle;

final class StoreEvents
{
    /**
     * Событие store.order создаётся всякий раз, когда в системе создаётся заказ.
     *
     * Слушатель получит экземпляр Acme\StoreBundle\Event\FilterOrderEvent
     *
     * @var string
     */
    const onStoreOrder = 'store.order';
}
```

Отметим также, что этот класс по сути свой ничего *не делает*. Назначение класса `StoreEvents` - централизация данных о событии. Слушателям этого события будет передаваться специализированный класс `FilterOrderEvent`.

Создание объекта события

Позднее, когда вы будете отправлять это событие, вы создадите экземпляр класса `Event` и передадите этот экземпляр всем слушателям события. Если вы не хотите передавать никакой дополнительной информации слушателям, вы можете использовать класс `Symfony\Component\EventDispatcher\Event`. В большинстве же случаев, вы *будете* передавать информацию о событии слушателям. Для этого необходимо создать новый класс, который будет наследоваться от класса `Symfony\Component\EventDispatcher\Event`.

В этом примере, каждый слушатель будет должен получить доступ к некоторому объекту `Order`. Создадим класс `Event`, который реализует такое поведение:

```
<?php
namespace Acme\StoreBundle\Event;

use Symfony\Component\EventDispatcher\Event;
use Acme\StoreBundle\Order;

class FilterOrderEvent extends Event
{
    protected $order;

    public function __construct(Order $order)
    {
        $this->order = $order;
    }

    public function getOrder()
    {
        return $this->order;
    }
}
```

Каждый слушатель теперь имеет доступ к объекту `Order` при помощи метода `getOrder`.

Отправка события

Метод **:method:‘Symfony\\Component\\EventDispatcher\\EventDispatcher::dispatch‘** уведомляет всех слушателей о событии. Он принимает два аргумента: наименование события для отправки и экземпляр `Event` для передачи каждому слушателю этого события:

```
<?php
use Acme\StoreBundle\StoreEvents;
use Acme\StoreBundle\Order;
```

```
use Acme\StoreBundle\Event\FilterOrderEvent;

// заказ как-то создаётся или получается
$order = new Order();
// ...

// создаём FilterOrderEvent и его отправка
$event = new FilterOrderEvent($order);
$dispatcher->dispatch(StoreEvents::onStoreOrder, $event);
```

Объект `FilterOrderEvent` создаётся и передаётся в метод `dispatch`. Теперь, любой слушатель события `store.order` будет получать `FilterOrderEvent` и соответственно иметь доступ к объекту `Order` при помощи метода `getOrder`:

```
<?php
// какой-то слушатель, подписанный на событие store.order методом onStoreOrder
use Acme\StoreBundle\Event\FilterOrderEvent;

public function onStoreOrder(FilterOrderEvent $event)
{
    $order = $event->getOrder();
    // далее выполняются какие-то действия с заказом
}
```

Внутри объекта Диспетчера событий

Если вы взглянете на класс `EventDispatcher`, вы увидите, что этот класс работает не как Singleton (нет статического метода `getInstance()`). Это сделано преднамеренно, так как вам, возможно, потребуется иметь несколько конкурирующих диспетчеров в рамках одного запроса. Но это также означает, что вам нужен способ для передачи диспетчеру объектов, которые нужно подключить или которые надо уведомить о событии.

Общепринятой практикой является внедрение объекта диспетчера в ваши объекты, т.е. внедрение зависимости.

Вы можете использовать внедрение в конструктор:

```
class Foo
{
    protected $dispatcher = null;

    public function __construct(EventDispatcher $dispatcher)
    {
        $this->dispatcher = $dispatcher;
    }
}
```

Или же внедрение через метод (setter injection):

```
class Foo
{
    protected $dispatcher = null;

    public function setEventDispatcher(EventDispatcher $dispatcher)
    {
        $this->dispatcher = $dispatcher;
    }
}
```

Выбор того или иного метода - это дело вкуса. Многие предпочитают метод с конструктором, так как объекты полностью инициализируются во время создания. Но когда у вас имеется длинный список зависимостей, использовать метод-сеттер это тоже вариант, особенно для опциональных зависимостей.

Совет: Если вы используете внедрение зависимости как мы делали в двух примерах выше, вы можете использовать Symfony2 Dependency Injection component для того чтобы управлять внедрением службы `event_dispatcher` для этих объектов.

```
# src/Acme/HelloBundle/Resources/config/services.yml
services:
    foo_service:
        class: Acme/HelloBundle/Foo/FooService
        arguments: [@event_dispatcher]
```

Подписка на события

Типичный способ ожидать возникновение события - зарегистрировать *слушателя события* при помощи диспетчера. Этот слушатель может слушать одно или несколько событий и уведомляется каждый раз при отправке нужного события.

Альтернативным способом для ожидания событий - использование *подписчика события*. Подписчик - это PHP класс, который имеет возможность сообщить диспетчеру на какие события он подписывается. Подписчик должен реализовывать интерфейс `Symfony\Component\EventDispatcher\EventSubscriberInterface`, который требует наличие одного статического метода `getSubscribedEvents`. Рассмотрим пример подписчика, который подписывается на события `kernel.response` и `store.order`:

```
<?php
namespace Acme\StoreBundle\Event;

use Symfony\Component\EventDispatcher\EventSubscriberInterface;
use Symfony\Component\HttpKernel\Event\FilterResponseEvent;
```

```
class StoreSubscriber implements EventSubscriberInterface
{
    static public function getSubscribedEvents()
    {
        return array(
            'kernel.response' => 'onKernelResponse',
            'store.order'     => 'onStoreOrder',
        );
    }

    public function onKernelResponse(FilterResponseEvent $event)
    {
        // ...
    }

    public function onStoreOrder(FilterOrderEvent $event)
    {
        // ...
    }
}
```

Этот класс похож на класс слушателя, за исключением того, что он сам может сообщить диспетчеру, на какие именно события он подписывается (будет слушать). Для регистрации подписчика в диспетчере необходимо использовать метод **:method:'Symfony\\Component\\EventDispatcher\\EventDispatcher::addSubscriber'**:

```
<?php
use Acme\StoreBundle\Event\StoreSubscriber;

$subscriber = new StoreSubscriber();
$dispatcher->addSubscriber($subscriber);
```

Диспетчер автоматически зарегистрирует подписчика для каждого события, возвращаемого методом `getSubscribedEvents`. Этот метод возвращает массив, индексами которого служат наименования событий, а значениями служат либо наименования методов, которые будут вызваны, либо массивы с именем метода и его приоритетом при обработке события.

Совет: Если вы используете Symfony2 MVC framework, подписчики можно регистрировать при помощи *конфигурации*. В качестве приятного бонуса, экземпляр подписчика будет создан лишь когда будет нужен.

Прекращение обработки событий

В некоторых случаях, один из слушателей может затребовать прекращение обработки события другими слушателями. Другими словами, слушатель должен иметь возможность сообщить диспетчеру, что он должен остановить обработку события всеми оставшимися слушателями (не уведомлять их о событии). Этого можно достигнуть внутри слушателя при помощи метода **:method:‘Symfony\\Component\\EventDispatcher\\Event::stopPropagation‘**:

```
<?php
use Acme\StoreBundle\Event\FilterOrderEvent;

public function onStoreOrder(FilterOrderEvent $event)
{
    // ...

    $event->stopPropagation();
}
```

Теперь, все слушатели `store.order`, которые ещё не были уведомлены о событии, уведомляться уже *не* будут.

2.16.4 Профайлер

Профайлер Symfony2, если он активирован, собирает полезную информацию о каждом запросе, выполненном к вашему приложение и сохраняет его для последующего анализа. Использование профайлера в девелоперском окружении поможет вам в отладке кода и увеличении быстродействия; используйте его в продуктовой среде для обнаружения проблем “по факту”.

Вам вряд ли придётся часто взаимодействовать с профайлером непосредственно, так как Symfony2 предоставляет визуализатор по типу Web Debug Toolbar и Web Profiler. Если вы используете Symfony2 Standard Edition, профайлер, дебаг-панель и веб-профайлер уже настроены и подключены.

Примечание: Профайлер собирает информацию обо всех запросах (простые запросы, перенаправления, исключения, Ajax запросы, ESI запросы; а также о всех HTTP методах и обо всех форматах). Это означает, что для одного URL вы можете иметь много профилированных данных (по одному на каждую пару запрос/ответ).

Визуализация данных профайлера

Использование Web Debug Toolbar

В dev окружении web debug toolbar расположен в низу каждой страницы. Он отображает обобщённые данные профайлера и предоставляет доступ к полезной информации, когда что-либо работает не так как ожидалось.

Если обобщённых данных не хватает, вы можете кликнуть на ссылку с токеном (строка из 13 случайных символов) и перейти на страницу Web Profiler.

Примечание: Если токен не кликается, это означает, что маршруты профайлера не зарегистрированы (см. ниже информацию о конфигурировании).

Анализ данных в Web Profiler

Web Profiler - это инструмент визуализации данных профилирования, который вы можете использовать в разработке для отладки вашего кода и увеличения его быстродействия; но его также можно использовать для отслеживания проблем в продуктовой среде. Он предоставляет всю информацию, собранную профайлером, в своём веб-интерфейсе.

Доступ к данным профайлера

Вам не обязательно использовать визуализатор для доступа к данным профайлера. Как же вам получить доступ к информации профайлера для некоторого запроса по факту его выполнения? Когда профайлер сохраняет данные о запросе, он также ассоциирует с ними некоторый токен; этот токен доступен в заголовке ответа `X-Debug-Token`:

```
$profile = $container->get('profiler')->loadProfileFromResponse($response);

$profile = $container->get('profiler')->loadProfile($token);
```

Совет: Когда профайлер активирован, но нет web debug toolbar, или же когда вы хотите получить токен для Ajax запроса, используйте, например, Firebug для того, чтобы получить заголовок `X-Debug-Token`.

Используйте метод `find()`, для получения доступа к токенам по какому-либо критерию:

```
// получить 10 последних токенов
$tokens = $container->get('profiler')->find('', '', 10);
```

```
// получить последние 10 токенов для всех URL, содержащих /admin/
$tokens = $container->get('profiler')->find('', '/admin/', 10);

// получить последние 10 токенов для локальных запросов
$tokens = $container->get('profiler')->find('127.0.0.1', '', 10);
```

Если вы хотите манипулировать данными профайлера на другой машине, используйте методы `export()` и `import()`:

```
// в prod окружении
$profile = $container->get('profiler')->loadProfile($token);
$data = $profiler->export($profile);

// в dev окружении
$profiler->import($data);
```

Конфигурирование

Конфигурация по умолчанию содержит разумные настройки профайлера, дебаг-панели (web debug toolbar) и веб-профайлера (web profiler). Ниже приведён пример конфигурации для dev окружения:

- *YAML*

```
# загрузка профайлера
framework:
    profiler: { only_exceptions: false }

# активация веб-профайлера
web_profiler:
    toolbar: true
    intercept_redirects: true
    verbose: true
```

- *XML*

```
<!-- xmlns:webprofiler="http://symfony.com/schema/dic/webprofiler" -->
<!-- xsi:schemaLocation="http://symfony.com/schema/dic/webprofiler http://symfony.com/sche

<!-- загрузка профайлера -->
<framework:config>
    <framework:profiler only-exceptions="false" />
</framework:config>

<!-- активация веб-профайлера -->
<webprofiler:config
    toolbar="true"
```

```
    intercept-redirects="true"
    verbose="true"
/>
```

- *PHP*

```
<?php
// загрузка профайлера
$container->loadFromExtension('framework', array(
    'profiler' => array('only-exceptions' => false),
));

// активация веб-профайлера
$container->loadFromExtension('web_profiler', array(
    'toolbar' => true,
    'intercept-redirects' => true,
    'verbose' => true,
));
```

Если `only-exceptions` имеет значение `true`, профайлер собирает данные только при возникновении исключений.

Если `intercept-redirects` имеет значение `true`, профайлер перехватывает перенаправления и предоставляет вам возможность наблюдать собранные данные перед перенаправлением.

Если `verbose` имеет значение `true`, Web Debug Toolbar отображает большое количество данных. Если присвоить `verbose` значение `false`, вторичная информация не будет отображаться.

Если вы активировали web profiler, вам также необходимо подключить его маршруты:

- *YAML*

```
_profiler:
    resource: @WebProfilerBundle/Resources/config/routing/profiler.xml
    prefix:   /_profiler
```

- *XML*

```
<import resource="@WebProfilerBundle/Resources/config/routing/profiler.xml" prefix="/_profi
```

- *PHP*

```
$collection->addCollection($loader->import("@WebProfilerBundle/Resources/config/routing/pro
```

Так как профайлер выполняет дополнительную работу для каждого запроса, вы, возможно, захотите активировать его в продуктовой среде лишь в некоторых случаях. Опция `only-exceptions` устанавливает лимит профилирования в 500 страниц, но что,

если вы захотите получить информацию, когда IP клиента имеет некоторое определённое значение или если запрашивается строго определённая часть сайта? Вы можете использовать request matcher:

- *YAML*

```yaml
# активирует профайлер для запросов из подсети 192.168.0.0/24
framework:
    profiler:
        matcher: { ip: 192.168.0.0/24 }

# активирует профайлер только для URL /admin
framework:
    profiler:
        matcher: { path: "^/admin/" }

# комбинирование правил
framework:
    profiler:
        matcher: { ip: 192.168.0.0/24, path: "^/admin/" }

# использование пользовательской службы matcher
framework:
    profiler:
        matcher: { service: custom_matcher }
```

- *XML*

```xml
<!-- активирует профайлер для запросов из подсети 192.168.0.0/24 -->
<framework:config>
    <framework:profiler>
        <framework:matcher ip="192.168.0.0/24" />
    </framework:profiler>
</framework:config>

<!-- активирует профайлер только для URL /admin -->
<framework:config>
    <framework:profiler>
        <framework:matcher path="^/admin/" />
    </framework:profiler>
</framework:config>

<!-- комбинирование правил -->
<framework:config>
    <framework:profiler>
        <framework:matcher ip="192.168.0.0/24" path="^/admin/" />
    </framework:profiler>
</framework:config>
```

```
<!-- использование пользовательской службы matcher -->
<framework:config>
    <framework:profiler>
        <framework:matcher service="custom_matcher" />
    </framework:profiler>
</framework:config>
```

- *PHP*

```
<?php
// активирует профайлер для запросов из подсети 192.168.0.0/24
$container->loadFromExtension('framework', array(
    'profiler' => array(
        'matcher' => array('ip' => '192.168.0.0/24'),
    ),
));

// активирует профайлер только для URL /admin
$container->loadFromExtension('framework', array(
    'profiler' => array(
        'matcher' => array('path' => '^/admin/'),
    ),
));

// комбинирование правил
$container->loadFromExtension('framework', array(
    'profiler' => array(
        'matcher' => array('ip' => '192.168.0.0/24', 'path' => '^/admin/'),
    ),
));

# использование пользовательской службы matcher
$container->loadFromExtension('framework', array(
    'profiler' => array(
        'matcher' => array('service' => 'custom_matcher'),
    ),
));
```

2.16.5 Читайте в книге рецептов

- *Как использовать профилировщик в Функциональном тесте*
- /cookbook/profiler/data_collector
- /cookbook/event_dispatcher/class_extension
- /cookbook/event_dispatcher/method_behavior

2.17 Стабильный API Symfony2

Стабильный API Symfony2 это подмножество всех опубликованных методов Symfony2 (как компонентов, так и пакетов из состава ядра), которые объединены по следующим признакам:

- Пространство имён и имя класса не будет изменяться;
- Наименование метода не будет изменяться;
- Сигнатура метода (аргументы и тип возвращаемого значения) не будет изменяться;
- Семантика того, что метод делает не будет изменяться;

Хотя, реализация метода может меняться со временем. Единственный случай, оправдывающий изменение стабильного API - исправление дыр в безопасности.

Стабильный API основывается на списке whitelist, тагированном *@api*. По этой причине всё, что не имеет этого тага - не является частью стабильного API.

Совет: Любой сторонний пакет может также публиковать свой собственный стабильный API.

Начиная с Symfony 2.0, следующие компоненты имеют публичный API:

- BrowserKit
- ClassLoader
- Console
- CssSelector
- DependencyInjection
- DomCrawler
- EventDispatcher
- Finder
- HttpFoundation
- HttpKernel
- Locale
- Process
- Routing
- Templating

- Translation
- Validator
- Yaml
- *Symfony2 и основы HTTP*
- *Symfony2 против чистого PHP*
- *Установка и настройка Symfony2*
- *Создание страниц в Symfony2*
- *Контроллер*
- *Маршрутизация*
- *Создание и использование Шаблонов*
- *Базы данных и Doctrine ("Модель")*
- *Тестирование*
- *Валидация*
- *Формы*
- *Безопасность*
- *HTTP Кэширование*
- *Переводы*
- *Контейнер служб*
- `/book/performance`
- *Составные части*
- *Стабильный API Symfony2*
- *Symfony2 и основы HTTP*
- *Symfony2 против чистого PHP*
- *Установка и настройка Symfony2*
- *Создание страниц в Symfony2*
- *Контроллер*
- *Маршрутизация*
- *Создание и использование Шаблонов*
- *Базы данных и Doctrine ("Модель")*
- *Тестирование*

- *Валидация*
- *Формы*
- *Безопасность*
- *HTTP Кэширование*
- *Переводы*
- *Контейнер служб*
- `/book/performance`
- *Составные части*
- *Стабильный API Symfony2*

Часть III

Рецепты

Глава 3

Cookbook

3.1 Как создать и разместить Проект на Symfony2 в git-репозитории

Совет: Несмотря на то, что эта статья посвещяна git, основные принципы, описанные тут, актуальны и для Subversion.

Если Вы уже прочитали статью *Создание страниц в Symfony2* и немного познакомились с Symfony, смело можно создавать свой собственный проект. Этот рецепт познакомит Вас с лучшим способом создания проекта на Symfony2 с использованием системы контроля версий git.

3.1.1 Предварительная настройка проекта

Для начала Вам нужно скачать Symfony и инициализировать Ваш локальный git репозиторий:

1. Скачайте Symfony2 Standard Edition без сторонних библиотек (without vendors).
2. Распакуйте дестрибутив. Будет создана директория Symfony с базовой структурой проекта, файлами конфигурации и т.д. Переименуйте ее, как сочтете нужным.
3. Создайте новый файл с именем `.gitignore` в корне Вашего проекта и вставьте в него следующий код. Все файлы, совпадающие с перечисленными шаблонами, git будет игнорировать:

```
/web/bundles/
/app/bootstrap*
/app/cache/*
```

```
/app/logs/*
/vendor/
/app/config/parameters.yml
```

4. Скопируйте `app/config/parameters.yml` в `app/config/parameters.yml.dist`. Файл `parameters.yml` игнорируется git'ом (смотрите выше), поэтому такие машино-зависимые настройки, как например пароль от базы данных, не будут отправляться в репозиторий. Имея файл `app/config/parameters.yml.dist` в репозитории, новые разработчики смогут быстро выгрузуть проект, скопировать этот файл в `parameters.yml`, настроить его и начать разработку.

5. Инициализируйте Ваш git репозиторий:

```
$ git init
```

6. Добавьте все начальные файлы в git:

```
$ git add .
```

7. Создайте первый коммит Вашего проекта:

```
$ git commit -m "Initial commit"
```

8. Наконец, скачайте все сторонние библиотеки:

```
$ php bin/vendors install
```

На этом моменте Вы имеете полностью функционирующий проект на Symfony2, который правильно размещен в git. Вы можете начинать программировать и отправлять изменения в Ваш репозиторий.

Сейчас Вы можете переключиться на статью *Создание страниц в Symfony2* для изучения вопросов конфигурации и разработки Вашего приложения.

Совет: Стандартная версия Symfony2 содержит в себе некоторый функционал для демонстрации. Чтобы убрать демонстрационный код, следуйте инструкциям Standard Edition Readme.

3.1.2 Управление внешними библиотеками с помощью bin/vendors и deps

Каждый проект на Symfony использует большую группу сторонних библиотек.

По умолчанию, эти библиотеки скачиваются при запуске команды `php bin/vendors install`. Этот скрипт читает из файла `deps` и скачивает полученные библиотеки в

папку `vendor/`. Он также читает файл `deps.lock` и связывает каждую упомянутую библиотеку с указанным git хешем в репозитории.

В нашем проекте сторонние библиотеки не являются частью git репозитория. Они также не являются дочерними модулями (submodules). Вместо этого мы полагаемся на файлы `deps` и `deps.lock`, а также на скрипт `bin/vendors`, который всем управляет. Эти файлы лежат в репозитории, и каждая версия проекта использует необходимые версии сторонних библиотек. Получается, Вы можете использовать скрипт vendors, чтобы Ваш проект был в актуальном состоянии.

Всякий раз, когда разработчик копирует проект, он должен выполнить скрипт `php bin/vendors install` чтобы убедиться, что все необходимые сторонние библиотеки установлены.

Обновление Symfony

Поскольку Symfony это группа сторонних библиотек и эти сторонние библиотеки полностью контролируются через `deps` и `deps.lock`, обновление Symfony - это простое обновление этих файлов до состояния их в последней версии Symfony Standard Edition.
Конечно же, если Вы добавили дополнительные источники в `deps` или `deps.lock`, убедитесь, что обновляете только оригинальные источники (т.е. не трогайте Ваши дополнительные источники).

Осторожно: Также существует команда `php bin/vendors update`, но она не предназначена для обновления Вашего проекта и Вам не надо ее использовать в работе. Эта команда служит для фиксации версий всех Ваших сторонних библиотек при обновлений их до версии, указанной в `deps`, и записывает их хеши в файл `deps.lock`. Кроме того, если Вам захотелось обновить файл `deps.lock` в соответствии с установленными сторонними библиотеками, то просто запустите команду `php bin/vendors lock` чтобы сохранить соответствующие git SHA идентификаторы в файле deps.lock.

Сторонние библиотеки и Дочерние модули

Вместо того, чтобы использовать `deps` и скрипт `bin/vendors` для управления Вашими сторонними библиотеками, Вы можете использовать родные git submodules. Нет ничего плохого в этом выборе, но система `deps` - это официальное решение этой проблемы и дочерние модули git'а могут внести дополнительные сложности в работу.

3.1.3 Хранение Вашего проекта на Удаленном Сервере

Сейчас у Вас имеется полностью функционирующий проект на Symfony2, сохраненный в git. Тем не менее, во многих случаях Вам понадобится хранить проект на удаленном

сервере. Например для хранения резервной копии проекта или чтобы другие разработчики также имели доступ к проекту для совместной работы.

Самый простой способ хранить Ваш проект на удаленном сервере - это GitHub. На нем публичный репозитории бесплатны. За закрытые репозитории Вам нужно будет платить ежемесячную плату.

С другой стороны, Вы можете хранить Ваш git репозиторий на любом сервере. Достаточно лишь создать barebones repository и загрузить данные на него. Библиотека Gitolite может помочь Вам в этом процессе.

3.2 Как создать собственные страницы ошибок

Когда происходит какое-либо исключение в Symfony2, оно перехватывается внутри класса `Kernel` и в конечном счете перенаправляется специальному контроллеру, `TwigBundle:Exception:show` для обработки. Данный контроллер, который расположен в ядре пакета `TwigBundle`, определяет какой из шаблонов ошибок показать, и какой установить код ошибки данному исключению.

Совет: Способов настройки перехвата исключений гораздо больше, чем описано здесь. Обработка внутреннего события `kernel.exception`, которое возникает при возникновении исключений позволяет полностью получить контроль над обработкой исключений. Для получения дополнительной информации см. *Событие kernel.exception*.

Все шаблоны ошибок размещены внутри пакета `TwigBundle`. Для переопределения шаблонов, следует использовать стандартный способ переопределения шаблонов, которые размещены внутри пакета. Для получения дополнительной информации см. *overiding-bundle-templates*.

Например, чтобы переопределить шаблон по-умолчанию, который показывается конечному пользователю, создайте шаблон расположенный здесь: `app/Resources/TwigBundle/views/Exception/error.html.twig`:

```
<!DOCTYPE html>
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <title>An Error Occurred: {{ status_text }}</title>
</head>
<body>
    <h1>Oops! An Error Occurred</h1>
    <h2>The server returned a "{{ status_code }} {{ status_text }}".</h2>
</body>
</html>
```

Совет: Если вы не знакомы с Twig'ом, не стоит переживать. Twig - простой, мощный и необязательный шаблонизатор, который интегрирован с `Symfony2`.

В добавок к обычной HTML странице ошибок, Symfony предоставляет доступ к страницам ошибок для самых распространенных форматов ответов, включая JSON (`error.json.twig`), XML, (`error.xml.twig`), и даже Javascript (`error.js.twig`). И мы назвали всего несоколько из них. Для переопределения любого из этих шаблонов, просто создайте новый файл с тем же именем в каталоге `app/Resources/TwigBundle/views/Exception`. Такой способ является стандартным, с помощью которого переопределяются любые шаблоны которые есть в пакете.

3.2.1 Настройка 404 страницы и других страниц ошибок

Также вы можете настроить отдельные шаблоны ошибок в зависимости от кода состояния HTTP. Например, создайте шаблон `app/Resources/TwigBundle/views/Exception/error404.html.twig` для отображения специальной страницы для 404 ошибки (страница не найдена).

Для определения, какой шаблон использовать, Symfony использует следующий алгоритм:

- Сперва, он ищет шаблон для текущего формата и кода состояния (например `error404.json.twig`);
- Если такого шаблона не существует, тогда он ищет шаблон для текущего формата (например `error.json.twig`);
- Если такого шаблона не существует, тогда он возвращается к HTML шаблону (например `error.html.twig`).

Совет: Полный список стандартных шаблонов ошибок находится в каталоге `Resources/views/Exception`, пакета `TwigBundle`. В стандартной поставке Symfony2, пакет `TwigBundle` находится в каталоге `vendor/symfony/src/Symfony/Bundle/TwigBundle`. Чаще всего, самым простым способом настройки страницы ошибок, является её копирование из пакета `TwigBundle` в каталог `app/Resources/TwigBundle/views/Exception` и последующее редактирование.

Примечание: Страницы-исключения, которые удобны для отладки и которые демонстрируются разработчику также могут быть настроены данным способом - создайте шаблоны `exception.html.twig` для стандартной HTML страницы ошибок или соответственно `exception.json.twig` для JSON.

3.3 Как определять Контроллеры в качестве сервисов

Из руководства, вы узнали, что работать с контроллером легче, если он расширяет базовый класс `Symfony\Bundle\FrameworkBundle\Controller\Controller`. Данный способ хорошо работает, однако контроллер можно определить в виде службы.

Чтобы сослаться на контроллер который определен в виде службы, следует использовать нотацию (обозначение) с одним знаком двоеточия (:). Например, мы определили сервис с именем `my_controller` и хотим вызвать метод `indexAction()` внутри него:

```
$this->forward('my_controller:indexAction', array('foo' => $bar));
```

Также необходимо использовать такую же запись для значений маршрута `_controller`

```
my_controller:
    pattern:   /
    defaults:  { _controller: my_controller:indexAction }
```

При таком способе использования контроллера, он должен быть определен в настройках контейнера сервисов. Для получения дополнительной информации см. главу *Service Container*

Контроллеры определенные как сервисы, скорее всего не будут наследниками базового класса `Controller`. Вместо того, чтобы использовать методы которые он предоставляет, скорее всего вы будете работать непосредственно со службами которые необходимы именно вам. К счастью, решения многих распространенных задач не сопровождается большими сложностями и базовый класс `Controller` является хорошим источником знаний,

---

Примечание: Определение контроллера в виде службы требует немного больше усилий, чем просто контроллера. Основным преимуществом сервиса является то, что весь контроллер или любая служба которая передается контроллеру, может быть изменена через настройку контейнера сервисов. При разработке пакетов с открытым исходным кодом или пакета, который будет использован во множестве разных проектов, данное преимущество представляется особенно применимым. Даже если вы не будете использовать контроллеры в качестве служб, их применение можно будет обнаружить в пакетах Symfony2 с открытым исходным кодом.

3.4 Как заставить маршрутизатор всегда использовать HTTPS или HTTP

Иногда Вам необходимо установить защищенное соединение для ресурса и Вы хотите быть уверенными, что доступ к этому ресурсу будет всегда осуществляться через протокол HTTPS. Компонент маршрутизации позволяет Вам настроить принудительное использование схемы URI с помощью параметра `_scheme`:

- *YAML*

```
secure:
    pattern:  /secure
    defaults: { _controller: AcmeDemoBundle:Main:secure }
    requirements:
        _scheme:  https
```

- *XML*

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

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

    <route id="secure" pattern="/secure">
        <default key="_controller">AcmeDemoBundle:Main:secure</default>
        <requirement key="_scheme">https</requirement>
    </route>
</routes>
```

- *PHP*

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

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

return $collection;
```

Приведенная выше конфигурация маршрута `secure` всегда будет использовать протокол HTTPS.

Когда генерируется URL `secure`, в случае, если текущая схема HTTP, то Symfony автоматически сгенерирует абсолютный URL со схемой HTTPS.

```
# Если текущая схема - HTTPS
{{ path('secure') }}
# сгенерирует /secure

# Если текущая схема - HTTP
{{ path('secure') }}
# сгенерирует https://example.com/secure
```

Правило также применяется и для входящих запросов. Если Вы попробуете получить доступ к ресурсу `/secure` через HTTP, Symfony автоматически перенаправит Вас на тот же URL, но с использованием схемы HTTPS.

Приведенные выше примеры используют протокол `https` для `_scheme`, но также Вы можете ограничить маршрут на использование только `http` протокола.

Примечание: Компонент Безопасности предлагает другой способ ограничить использование только HTTP или HTTPs протокола посредством параметра `requires_channel`. Этот альтернативный метод больше подходит для защиты "области" Вашего web-сайта (все URL в `/admin`) или когда Вы хотите защитить все URL, объявленные в стороннем пакете.

3.5 Как отправлять электронную почту

Рассылка электронной почты, является классической задачей для любого веб-приложения, и одной из тех задач, в которой имеются определенные сложности и потенциальные проблемы. Вместо изобретения колеса, одним из решений по рассылке электронной почты, является использование пакета `SwiftmailerBundle`, который использует возможности библиотеки Swiftmailer .

Примечание: Не забудьте подключить пакет в ядре, до начала его использования:

```
public function registerBundles()
{
    $bundles = array(
        // ...
        new Symfony\Bundle\SwiftmailerBundle\SwiftmailerBundle(),
    );

    // ...
}
```

3.5.1 Настройка

До момента использования компонента Swiftmailer, его необходимо настроить. Обязательным в настройке компонента является параметр `transport`:

- *YAML*

```yaml
# app/config/config.yml
swiftmailer:
    transport:  smtp
    encryption: ssl
    auth_mode:  login
    host:       smtp.gmail.com
    username:   ваш_логин
    password:   ваш_пароль
```

- *XML*

```xml
<!-- app/config/config.xml -->

<!--
xmlns:swiftmailer="http://symfony.com/schema/dic/swiftmailer"
http://symfony.com/schema/dic/swiftmailer http://symfony.com/schema/dic/swiftmailer/swiftma
-->

<swiftmailer:config
    transport="smtp"
    encryption="ssl"
    auth-mode="login"
    host="smtp.gmail.com"
    username="ваш_логин"
    password="ваш_пароль" />
```

- *PHP*

```php
// app/config/config.php
$container->loadFromExtension('swiftmailer', array(
    'transport'  => "smtp",
    'encryption' => "ssl",
    'auth_mode'  => "login",
    'host'       => "smtp.gmail.com",
    'username'   => "ваш_логин",
    'password'   => "ваш_пароль",
));
```

Большая часть настроек Swiftmailer отвечает за то каким образом сообщения должны быть доставлены.

Возможно использовать следующие параметры:

- `transport` (`smtp`, `mail`, `sendmail`, или `gmail`)
- `username`
- `password`
- `host`
- `port`
- `encryption` (`tls`, или `ssl`)
- `auth_mode` (`plain`, `login`, или `cram-md5`)
- `spool`
  - `type` (каким образом организовывать очередь сообщений, на данный момент поддерживается только способ `file`)
  - `path` (где хранить сообщения)
- `delivery_address` (адрес на который отправляют ВСЕ письма)
- `disable_delivery` (установка значения в true отключает доставку писем)

3.5.2 Рассылка электронных сообщений

Библиотека Swiftmailer работает с объектами `Swift_Message`, и занимается их созданием, конфигурированием и рассылкой. "Рассылатель" (или mailer) отвечает за доставку сообщений и доступен через сервис `mailer`. В целом отправка письма достаточно проста:

```
public function indexAction($name)
{
    // получаем 'mailer' (обязателен для инициализации Swift Mailer)
    $mailer = $this->get('mailer');

    $message = \Swift_Message::newInstance()
        ->setSubject('Hello Email')
        ->setFrom('send@example.com')
        ->setTo('recipient@example.com')
        ->setBody($this->renderView('HelloBundle:Hello:email', array('name' => $name)))
    ;
    $mailer->send($message);

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

Отметим, что "тело" письма, хранится в шаблоне и отображается с помощью метода `renderView()`.

Объект `$message` содержит множество других опций, таких как вложения, содержимое в формате HTML, и т.д. В документации к библиотеке Swiftmailer хорошо освещена глава Создание сообщений в которой можно найти информацию о том как создавать сообщения и опциях которые при этом доступны.

Совет: Рекомендуем прочитать документ "*Как использовать Gmail для отправки электронных писем*" в котором рассказано как использовать почту gmail в качестве транспорта на стадии разработки.

3.6 Как использовать Gmail для отправки электронных писем

Во время разработки, отправка писем с помощью сервиса Gmail может оказаться более легким и практичным решением, нежели использование SMTP сервера.

Совет: Вместо того, чтобы использовать свою учетную запись Gmail, лучшим решением будет создать новый аккаунт, применительно для этих целей.

В конфигурационном файле для среды разработки, измените настройку `transport` на `gmail` и задайте настройки `username` и `password` согласно значениям из Gmail:

- *YAML*

```
# app/config/config_dev.yml
swiftmailer:
    transport: gmail
    username:  ваш_gmail_логин
    password:  ваш_gmail_пароль
```

- *XML*

```
<!-- app/config/config_dev.xml -->

<!--
xmlns:swiftmailer="http://symfony.com/schema/dic/swiftmailer"
http://symfony.com/schema/dic/swiftmailer http://symfony.com/schema/dic/swiftmailer/swiftma
-->

<swiftmailer:config
    transport="gmail"
    username="ваш_gmail_логин"
    password="ваш_gmail_пароль" />
```

- *PHP*

```
// app/config/config_dev.php
$container->loadFromExtension('swiftmailer', array(
    'transport' => "gmail",
    'username'  => "ваш_gmail_логин",
    'password'  => "ваш_gmail_пароль",
));
```

На этом настройка Gmail закончена!

Примечание: Транспорт `gmail` является всего лишь готовым шаблоном, который использует транспорт `smtp` и устанавливает поля `encryption`, `auth_mode` и `host` для работы с почтой Gmail.

3.7 Как смоделировать HTTP аутентификацию в Функциональном тесте

Если вашему приложению необходима HTTP аутентификация, передайте имя пользователя и пароль в качестве переменных сервера в метод `createClient()`:

```
$client = $this->createClient(array(), array(
    'PHP_AUTH_USER' => 'username',
    'PHP_AUTH_PW'   => 'pa$$word',
));
```

Также можно делать переопределения в каждом запросе:

```
$client->request('DELETE', '/post/12', array(), array(
    'PHP_AUTH_USER' => 'username',
    'PHP_AUTH_PW'   => 'pa$$word',
));
```

3.8 Как тестировать взаимодействие с несколькими клиентами

Если требуется смоделировать взаимодействие между разными Клиентами (представьте, например, чат), то создайте несколько Клиентов:

```
$harry = static::createClient();
$sally = static::createClient();
```

```
$harry->request('POST', '/say/sally/Hello');
$sally->request('GET', '/messages');

$this->assertEquals(201, $harry->getResponse()->getStatusCode());
$this->assertRegExp('/Hello/', $sally->getResponse()->getContent());
```

Этот подход работает, за исключением тех случаев, когда ваш код обрабатывает глобальное состояния или зависит от библиотек третьих лиц, которые также его используют. Для подобных случаев можно изолировать клиентов:

```
$harry = static::createClient();
$sally = static::createClient();

$harry->insulate();
$sally->insulate();

$harry->request('POST', '/say/sally/Hello');
$sally->request('GET', '/messages');

$this->assertEquals(201, $harry->getResponse()->getStatusCode());
$this->assertRegExp('/Hello/', $sally->getResponse()->getContent());
```

Изолированные клиенты выполняют свои запросы в отдельных чистых PHP процессах, что исключает любые побочные эффекты.

Совет: Так как изолированный клиент работает медленнее, то можно одного клиента оставить выполняться в главном процессе, а остальных изолировать.

3.9 Как использовать профилировщик в Функциональном тесте

Настоятельно рекомендуется чтобы функциональный тест проверял только Response. Но если пишутся функциональные тесты, следящие за production серверами, то возможно у вас появится желание написать тесты, использующие данные профилировщика, т. к. они позволяют проверить множество параметров и обеспечить соблюдение определенных показателей.

`Профилировщик` в Symfony2 собирает множество данных по каждому запросу. Используйте их для замера количества запросов к БД, времени затраченного фреймворком и т. д. Но, прежде чем писать проверочные выражения, всегда следует проверять доступность профилировщика (по-умолчанию к нему есть доступ в `test` окружении):

```
class HelloControllerTest extends WebTestCase
{
    public function testIndex()
    {
        $client = static::createClient();
        $crawler = $client->request('GET', '/hello/Fabien');

        // Напишите выражения, относящиеся к Response
        // ...

        // Проверяет, доступен ли профилировщик
        if ($profile = $client->getProfile()) {
            // проверяет количество запросов
            $this->assertTrue($profile->get('db')->getQueryCount() < 10);

            // проверяет время, затраченное фреймворком
            $this->assertTrue( $profile->get('timer')->getTime() < 0.5);
        }
    }
}
```

Если тест провалится, основываясь на данных профилировщика (например, слишком много запросов к БД), то можно воспользоваться Веб Профилировщиком для анализа запросов после завершения тестов. Это легко сделать если встроить метку в сообщение об ошибке:

```
$this->assertTrue(
    $profile->get('db')->getQueryCount() < 30,
    sprintf('Checks that query count is less than 30 (token %s)', $profile->getToken())
);
```

Осторожно: Хранилище профилировщика может различаться в зависимости от окружения (особенно если используется хранилище SQLite, являющееся одним из сконфигурированных по-умолчанию).

Примечание: В тестах информация профилировщика доступна даже в тех случаях, когда клиент изолирован либо используется HTTP слой.

Совет: Прочитайте про API встроенных `сборщиков данных` чтобы узнать больше об их интерфейсах.

3.10 Как использовать Varnish для ускорения работы сайта

Так как кеш Symfony2 использует стандартные HTTP-заголовки кеша, *Обратный прокси Symfony2* может быть легко заменен любым другим reverse proxy. Varnish - это мощный HTTP-акселератор с открыытми исходными кодами, который позволяет быстро отдавать закешированный контент и позволяет использовать *Edge Side Includes*.

3.10.1 Настройка

Как мы видели раньше, Symfony2 может определить, используется ли reverse proxy, который понимает ESI, или нет. Это работает «из коробки», когда вы пользуетесь reverse proxy в Symfony2, но для работы с Varnish нужна специальная настройка. Благодаря стандарту, разработанному Akamaï (**'Edge Architecture'_**), советы из этой главы могут быть полезны даже если вы не используете Symfony2.

Примечание: Varnish поддерживает только атрибут `src` для тегов ESI (атрибуты `onerror` и `alt` игнорируются).

Для начала настройте Varnish таким образом, чтобы он оповещал о поддержке ESI через заголовок `Surrogate-Capability` тех запросов, которые перенаправляются backend-приложению:

```
sub vcl_recv {
    set req.http.Surrogate-Capability = "abc=ESI/1.0";
}
```

Затем, оптимизируйте Varnish таким образом, чтобы он парсил содержимое ответа, когда в нем присутствует хотя бы один тег ESI. Этого можно добиться, проверив наличие ответа `Surrogate-Control`, который добавляется автоматически Symfony2:

```
sub vcl_fetch {
    if (beresp.http.Surrogate-Control ~ "ESI/1.0") {
        unset beresp.http.Surrogate-Control;
        esi;
    }
}
```

Осторожно: Не используйте сжатие с ESI, так как Varnish не сможет пропарсить содержимое ответа. Если вы хотите использовать сжатие, установите веб-сервер перед Varnish, который бы организовывал сжатие ответа.

3.10.2 Аннулирование кеша

По идее, вам никогда не потребуется аннулирование кеша, потому что это уже учитывается в HTTP (см. *Очистка (аннулирование) кэша*).

Однако, Varnish может быть настроен так, чтобы мог принимать специальный метод HTTP - `PURGE` - который может аннулировать кеш для входящих запросов:

```
sub vcl_hit {
    if (req.request == "PURGE") {
        set obj.ttl = 0s;
        error 200 "Purged";
    }
}

sub vcl_miss {
    if (req.request == "PURGE") {
        error 404 "Not purged";
    }
}
```

Осторожно: Мы должны ограничить доступ к HTTP-методу `PURGE`, чтобы избежать его использование другими людьми;

3.11 Внедрение переменных во все шаблоны (т.н. Глобальные переменные)

Иногда вам может потребоваться, чтобы некоторая переменная была доступна во всех ваших шаблонах. Этого можно достичь при помощи вашего файла `app/config/config.yml`:

```
# app/config/config.yml
twig:
    # ...
    globals:
        ga_tracking: UA-xxxxx-x
```

Теперь, переменная `ga_tracking` будет доступна во всех шаблонах Twig:

```
<p>Our google tracking code is: {{ ga_tracking }} </p>
```

Так просто! Вы также можете получить доступ к системным параметрам "*Параметры службы*", которые позволят вам изолировать или повторно использовать значение:

```
; app/config/parameters.yml
[parameters]
    ga_tracking: UA-xxxxx-x

# app/config/config.yml
twig:
    globals:
        ga_tracking: %ga_tracking%
```

The same variable is available exactly as before.

3.11.1 Более сложные глобальные переменные

Если глобальная переменная, которую вам надо использовать, более сложная - к примеру, объект - тогда вы не сможете воспользоваться приведённым выше методом. Вместо этого вам нужно создать *Расширение Twig* и возвращать глобальную переменную среди значений метода `getGlobals`.

3.12 Как использовать PHP шаблоны вмесро Twig

Не смотря на то, что Symfony2 по умолчанию использует шаблоны Twig в частве шаблонного движка, вы можете использовать PHP шаблоны, если захотите. Оба шаблонных движка одинаково поддерживаются в Symfony2. Symfony2 также добавляет к PHP шаблонам несколько удобных фич, чтобы сделать их более мощными.

3.12.1 Отображение PHP шаблонов

Если вы хотите использовать PHP шаблоны, во-первых, убедитесь что вы их активировали в настройках приложения:

- *YAML*

  ```
  # app/config/config.yml
  framework:
      # ...
      templating:    { engines: ['twig', 'php'] }
  ```

- *XML*

  ```
  <!-- app/config/config.xml -->
  <framework:config ... >
      <!-- ... -->
      <framework:templating ... >
  ```

```
            <framework:engine id="twig" />
            <framework:engine id="php" />
        </framework:templating>
    </framework:config>
```

- *PHP*

```
$container->loadFromExtension('framework', array(
    // ...
    'templating'      => array(
        'engines' => array('twig', 'php'),
    ),
));
```

Теперь вы можете использовать PHP шаблоны взамен Twig, просто указывая расширение `.php` для ваших шаблонов, а не `.twig`. Контроллер из примера ниже отображает шаблон `index.html.php`:

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

public function indexAction($name)
{
    return $this->render('AcmeHelloBundle:Hello:index.html.php', array('name' => $name));
}
```

3.12.2 Декорирование шаблонов

Чаще всего, шаблоны используют некоторые общие элементы, например всем известные header и footer. В Symfony2 этот вопрос решается несколько иначе - шаблон может быть декорирован другим шаблоном.

Шаблон `index.html.php` будет декорироваться шаблоном `layout.html.php` благодаря вызову метода `extend()`:

```
<!-- src/Acme/HelloBundle/Resources/views/Hello/index.html.php -->
<?php $view->extend('AcmeHelloBundle::layout.html.php') ?>

Hello <?php echo $name ?>!
```

Нотация `AcmeHelloBundle::layout.html.php` выглядит знакомо, не так ли? Это такая же нотация, которая используется для ссылки на шаблон. Часть `::` означает, что элемент "контроллер", таким образом соответствующий файл распложен напрямую в директории `views/`.

Теперь давайте взглянем на файл `layout.html.php`:

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

<h1>Hello Application</h1>

<?php $view['slots']->output('_content') ?>
```

Декорирующий шаблон (layout) в свою очередь декорирован другим шаблоном (`::base.html.php`). Symfony2 поддерживает множественные уровни декорирования: layout может быть декорирован другим шаблоном более высокого уровня. Когда часть "bundle" в наименовании шаблона пуста, он ищется в директории `app/Resources/views/`. Она содержит глобальные шаблоны уровня приложения:

```
<!-- app/Resources/views/base.html.php -->
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
        <title><?php $view['slots']->output('title', 'Hello Application') ?></title>
    </head>
    <body>
        <?php $view['slots']->output('_content') ?>
    </body>
</html>
```

Для обоих шаблонов, выражение `$view['slots']->output('_content')` заменяется контентом дочернего шаблона: `index.html.php` и `layout.html.php` соответственно (о слотах подробнее поговорим в следующей секции).

Как вы можете видеть, Symfony2 предоставляет методы посредством объекта `$view`. В шаблоне переменная `$view` всегда доступна и представляет собой специальный объект, которые предоставляет пакет методов, которые собственно и крутят винтики в движке шаблонизатора.

3.12.3 Работаем со Слотами

Слот - это некоторый кусочек кода, определённый в шаблоне и который можно повторно использовать в любом декорирующем шаблоне. В шаблоне `index.html.php` определён слот `title`:

```
<!-- src/Acme/HelloBundle/Resources/views/Hello/index.html.php -->
<?php $view->extend('AcmeHelloBundle::layout.html.php') ?>

<?php $view['slots']->set('title', 'Hello World Application') ?>

Hello <?php echo $name ?>!
```

Базовый шаблон уже имеет код для вывода заголовка:

```
<!-- app/Resources/views/base.html.php -->
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <title><?php $view['slots']->output('title', 'Hello Application') ?></title>
</head>
```

Метод `output()` вставляет контент слота и опционально принимает значение по умолчанию, которое будет использовано, если слот не будет определён. `_content` - это особый слот, который содержит рендер дочернего шаблона.

Для больших слотов можно использовать расширенный синтаксис:

```
<?php $view['slots']->start('title') ?>
    Some large amount of HTML
<?php $view['slots']->stop() ?>
```

3.12.4 Включение других шаблонов

Наилучшим способом сделать доступным в других шаблонах некоторый кусочек кода - это включить его в другие шаблорны.

Создадим шаблон `hello.html.php`:

```
<!-- src/Acme/HelloBundle/Resources/views/Hello/hello.html.php -->
Hello <?php echo $name ?>!
```

И изменим шаблон `index.html.php` таким образом чтобы он его подключал:

```
<!-- src/Acme/HelloBundle/Resources/views/Hello/index.html.php -->
<?php $view->extend('AcmeHelloBundle::layout.html.php') ?>

<?php echo $view->render('AcmeHelloBundle:Hello:hello.html.php', array('name' => $name)) ?>
```

Метод `render()` вычисляет и возвращает значение другого шаблона (это такой же метод, который используется в контроллере).

3.12.5 Встраивание Контроллеров

Что, если вы захотите встроить результат выполнения другого контроллера в шаблон? Это очень удобно при работе с Ajax, или же когда встраиваемый шаблон требует некоторые переменные, не доступные в главном шаблоне.

Если вы создадите действие `fancy` и захотите включить его в шаблон `index.html.php`, просто используйте такой код:

```
<!-- src/Acme/HelloBundle/Resources/views/Hello/index.html.php -->
<?php echo $view['actions']->render('AcmeHelloBundle:Hello:fancy', array('name' => $name, 'color
```

Строка `AcmeHelloBundle:Hello:fancy` ссылается на действие `fancy` контроллера `Hello`:

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

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

        return $this->render('AcmeHelloBundle:Hello:fancy.html.php', array('name' => $name, 'obj
    }

    // ...
}
```

Но где же определён элемент массива `$view['actions']`? Как и `$view['slots']`, здесь вызывается хелпер и в следующей секции об этом будет чуть подробнее.

3.12.6 Использование хелперов в шаблонах

Система шаблонов Symfony2 может быть легко и просто при помощи хелперов. Хелперы это PHP объекты, которые предоставляют удобные фичи в контексте шаблонов. `actions` и `slots` - это два хелпера из числа встроенных в Symfony2.

Создание ссылок между страницами

Говоря о веб-приложениях, создание ссылок между страницами - это крайне необходимая функция. Вместо того, чтобы хордкодить URLы в шаблонах, нужно использовать хелпер `router`, который знает как генерировать URL, основываясь на конфигурацию маршрутизатора. Если использовать этот подход, любой URL может быть легко обновлён при помощи изменения конфигурации:

```
<a href="<?php echo $view['router']->generate('hello', array('name' => 'Thomas')) ?>">
    Greet Thomas!
</a>
```

Метод `generate()` принимает имя маршрута и массив параметров в качестве аргументов. Имя маршрута - это ключ, по которому определяется маршрут, а параметры - это значения плэйсхолдеров из шаблона маршрута:

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

Работа с ресурсами: картинки, JavaScript, CSS

Чем бы был интернет без картинок, джаваскриптов и стилей? Symfony2 предоставляет вам таг `assets` для работы с ними:

```
<link href="<?php echo $view['assets']->getUrl('css/blog.css') ?>" rel="stylesheet" type="text/c

<img src="<?php echo $view['assets']->getUrl('images/logo.png') ?>" />
```

Главной обязанностью хелпера `assets` - сделать ваше приложение более портируемым. Благодаря этому хелперу вы можете перемещать корень вашего приложения внутри web root не меняя ничего у коде шаблонов.

3.12.7 Экранирование

При использовании PHP шаблонов необходимо экранировать все переменные:

```
<?php echo $view->escape($var) ?>
```

По умолчанию, метод `escape()` полагает, что переменная выводится в контексте HTML. Второй аргумент метода позволяет изменить контекст. Например, выводя что-то в JavaScript, используйте `js` контекст:

```
<?php echo $view->escape($var, 'js') ?>
```

3.13 Как использовать Monolog для журналирования

Библиотека Monolog предназначена для ведения журналов в PHP 5.3 и используется Symfony2. Её прототипом послужила библиотека LogBook в Python.

3.13.1 Использование

В Монологе, каждый элемент журналирования(логгер) определяет свой канал журналирования(logger). Каждый канал имеет стек обработчиков которые пишут журнал (лог) (причем обработчики могут быть общими).

Совет: При установке логгера в сервис, можно использовать *свой канал* и легко просматривать какая часть приложения оставила сообщение в журнале.

Простейшим обработчиком является `StreamHandler`, который пишет журнал в поток (по-умолчанию в `app/logs/prod.log` в production среде и `app/logs/dev.log` в среде разработки).

В состав Monolog также входит мощный обработчик, предназначенный для журналирования в production среде: `FingersCrossedHandler`. Он позволяет хранить сообщения в буфере и записывать их в журнал только при условии того, что оно доходит до уровня контроллера (ERROR в конфигурации стандартной редакции) перенаправляя их к другому обработчику.

Чтобы записать сообщение в журнал, просто получите доступ к сервису логгера из контейнера в вашем контроллере:

```
$logger = $this->get('logger');
$logger->info('We just go the logger');
$logger->err('An error occured');
```

Использование нескольких обработчиков

Логер использует стек обработчиков которые вызываются последовательно. Данная особенность позволяет легко записывать сообщения в журнал различными способами.

- *YAML*

```
monolog:
    handlers:
        syslog:
            type: stream
            path: /var/log/symfony.log
            level: error
        main:
            type: fingerscrossed
            action_level: warning
            handler: file
        file:
            type: stream
            level: debug
```

- *XML*

```
<container xmlns="http://symfony.com/schema/dic/services"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:monolog="http://symfony.com/schema/dic/monolog"
    xsi:schemaLocation="http://symfony.com/schema/dic/services http://symfony.com/schema/di
```

```
                    http://symfony.com/schema/dic/monolog http://symfony.com/schema/dic

    <monolog:config>
        <monolog:handler
            name="syslog"
            type="stream"
            path="/var/log/symfony.log"
            level="error"
        />
        <monolog:handler
            name="main"
            type="fingerscrossed"
            action-level="warning"
            handler="file"
        />
        <monolog:handler
            name="file"
            type="stream"
            level="debug"
        />
    </monolog:config>
</container>
```

Конфигурация выше, определяет стек обработчиков которые будут вызваны в порядке в котором они объявлены.

Совет: Обработчик “file” не будет включен в стек, так как он сам используется в качестве вложенного обработчика в production среде.

Примечание: Если у вас появиться желание изменить настройки MonologBundle в другом файле настроек, то необходимо будет полностью переопределить весь стек. Он не может быть объединен с текущими настройками, т.к. в результате объединения настроек невозможно управлять порядком вызова обработчиков.

Изменение форматирования

Обработчик использует `Formatter` для форматирования записей, перед записью их в журнал. Все обработчики Monolog по-умолчанию используют экземпляр `Monolog\Formatter\LineFormatter`, но его легко заменить своим собственным. Ваш собственный форматировщик должен использовать интерфейс `Monolog\Formatter\LineFormatterInterface`.

- *YAML*

```
services:
    my_formatter:
        class: Monolog\Formatter\JsonFormatter
monolog:
    handlers:
        file:
            type: stream
            level: debug
            formatter: my_formatter
```

- *XML*

```
<container xmlns="http://symfony.com/schema/dic/services"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:monolog="http://symfony.com/schema/dic/monolog"
    xsi:schemaLocation="http://symfony.com/schema/dic/services http://symfony.com/schema/di
                        http://symfony.com/schema/dic/monolog http://symfony.com/schema/dic

    <services>
        <service id="my_formatter" class="Monolog\Formatter\JsonFormatter" />
    </services>
    <monolog:config>
        <monolog:handler
            name="file"
            type="stream"
            level="debug"
            formatter="my_formatter"
        />
    </monolog:config>
</container>
```

3.13.2 Дополнительная информация в сообщениях журнала

Monolog позволяет добавлять дополнительные данные в сообщения перед их записью в журнал. Процессор может быть применен как ко всему стеку так и к какому-либо определенному обработчику из его состава.

Процессор - это сервис получающий запись в качестве первого аргумента и логгер или обработчик в качестве второго, в зависимости от того на каком уровне он вызывается.

- *YAML*

```
services:
    my_processor:
        class: Monolog\Processor\WebProcessor
monolog:
    handlers:
```

```
            file:
                type: stream
                level: debug
                processors:
                    - Acme\MyBundle\MyProcessor::process
        processors:
            - @my_processor
```

- *XML*

```
<container xmlns="http://symfony.com/schema/dic/services"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:monolog="http://symfony.com/schema/dic/monolog"
    xsi:schemaLocation="http://symfony.com/schema/dic/services http://symfony.com/schema/di
                        http://symfony.com/schema/dic/monolog http://symfony.com/schema/dic

    <services>
        <service id="my_processor" class="Monolog\Processor\WebProcessor" />
    </services>
    <monolog:config>
        <monolog:handler
            name="file"
            type="stream"
            level="debug"
            formatter="my_formatter"
        >
            <monolog:processor callback="Acme\MyBundle\MyProcessor::process" />
        </monolog:handler />
        <monolog:processor callback="@my_processor" />
    </monolog:config>
</container>
```

Совет: Если вашему процессору требуются зависимости, то можно объявить сервис и реализовать метод `__invoke` в классе, с тем чтобы сделать его вызываемым. После изменений процессор можно добавить в стек.

3.14 Как автоматически загружать классы

В случаях когда используются неопределенные классы, PHP использует механизм автозагрузки которому поручает загрузку файла описывающего класс. С Symfony2 поставляется универсальный автозагрузчик, который может загружать классы из файлов, которые реализуют одно из следующих соглашений:

- Технические стандарты взаимодействия для имен пространств и классов PHP 5.3;

- Соглашения именования классов в PEAR.

Если ваши классы и библиотеки 3-х лиц которыми вы пользуетесь в проекте следуют данным стандартам, автозагрузчик Symfony2 единственный автозагрузчик который вам когда-либо понадобиться.

3.14.1 Использование

Добавлено в версии 2.1: The `useIncludePath` method was added in Symfony 2.1. Регистрация класса `Symfony\Component\ClassLoader\UniversalClassLoader` автозагрузки проста:

```
<?php
require_once '/path/to/src/Symfony/Component/ClassLoader/UniversalClassLoader.php';

use Symfony\Component\ClassLoader\UniversalClassLoader;

$loader = new UniversalClassLoader();

// В качестве последней инстанции ищем в include_path.
$loader->useIncludePath(true);

$loader->register();
```

С целью улучшения быстродействия - пути к классам могут кэшироваться в памяти при помощи APC - для этого необходимо регистрировать класс `Symfony\Component\ClassLoader\ApcUniversalClassLoader`:

```
<?php
require_once '/path/to/src/Symfony/Component/ClassLoader/UniversalClassLoader.php';
require_once '/path/to/src/Symfony/Component/ClassLoader/ApcUniversalClassLoader.php';

use Symfony\Component\ClassLoader\ApcUniversalClassLoader;

$loader = new ApcUniversalClassLoader('apc.prefix.');
$loader->register();
```

Автозагрузчик полезен только при условии того, что вы добавите несколько библиотек для автозагрузки.

Примечание: Автозагрузчик автоматически регистрируется приложением на Symfony2 (см. `app/autoload.php`).

Если классы которые требуется автоматически загружать используют пространства имён, применяйте методы

:method:‘Symfony\\Component\\ClassLoader\\UniversalClassLoader::registerNamespace‘ или :method:‘Symfony\\Component\\ClassLoader\\UniversalClassLoader::registerNamespa

```
<?php
$loader->registerNamespace('Symfony', __DIR__.'/vendor/symfony/src');

$loader->registerNamespaces(array(
    'Symfony' => __DIR__.'/../vendor/symfony/src',
    'Monolog' => __DIR__.'/../vendor/monolog/src',
));

$loader->register();
```

Для классов которые используют соглашения об именовании в стиле PEAR, используйте метод :method:‘Symfony\\Component\\ClassLoader\\UniversalClassLoader::registerPrefi или :method:‘Symfony\\Component\\ClassLoader\\UniversalClassLoader::registerPrefixes‘:

```
<?php
$loader->registerPrefix('Twig_', __DIR__.'/vendor/twig/lib');

$loader->registerPrefixes(array(
    'Swift_' => __DIR__.'/vendor/swiftmailer/lib/classes',
    'Twig_'  => __DIR__.'/vendor/twig/lib',
));

$loader->register();
```

Примечание: Некоторые библиотеки требуют чтобы их корневой каталог также был включен в качестве пути для поиска PHP (`set_include_path()`).

Классы находящиеся в подпространствах имён или в суб-иерархии классов PEAR можно легко сгруппировать в подмножества, которые можно использовать в больших проектах:

```
<?php
$loader->registerNamespaces(array(
    'Doctrine\\Common'           => __DIR__.'/vendor/doctrine-common/lib',
    'Doctrine\\DBAL\\Migrations' => __DIR__.'/vendor/doctrine-migrations/lib',
    'Doctrine\\DBAL'             => __DIR__.'/vendor/doctrine-dbal/lib',
    'Doctrine'                   => __DIR__.'/vendor/doctrine/lib',
));

$loader->register();
```

В этом примере, при попытке использования класса в пространстве имен `Doctrine\Common` или его потомков, автозагрузчик в первую очередь просмотрит в поисках класса каталог `doctrine-common`, и в последнюю очередь каталог `Doctrine` (ко-

торый сконфигурирован последним). В данном случае порядок регистрации классов важен.

3.15 Как искать файлы

С помощью компонента **:namespace:‘Symfony\\Component\\Finder‘** можно легко и быстро находить необходимые файлы и каталоги.

3.15.1 Использование

Класс `Symfony\Component\Finder\Finder` производит поиск файлов и/или каталогов:

```
use Symfony\Component\Finder\Finder;

$finder = new Finder();
$finder->files()->in(__DIR__);

foreach ($finder as $file) {
    print $file->getRealpath()."\n";
}
```

Объект `$file` является экземпляром класса **:phpclass:‘SplFileInfo‘**. Код выше печатает имена всех файлов в текущем каталоге рекурсивно. Класс Finder использует свободный интерфейс, так что все методы возвращают тип данных Finder.

Совет: Экземпляр класса Finder является Iterator (итератором) PHP. Так, что вместо прохода над Finder’ом с помощью `foreach`, можно также конвертировать его в массив с помощью метода **:phpfunction:‘iterator_to_array‘**, или получить количество элементов, с помощью **:phpfunction:‘iterator_count‘**.

3.15.2 Критерии поиска

Расположение

Расположение является единственным обязательным параметром. Данный параметр указывает поисковику какую директорию использовать для поиска:

```
$finder->in(__DIR__);
```

Поиск в нескольких местах реализуется с помощью последовательных вызовов метода **:method:‘Symfony\\Component\\Finder\\Finder::in‘**:

```
$finder->files()->in(__DIR__)->in('/elsewhere');
```

Исключение каталогов из поиска осуществляется методом **:method:'Symfony\\Component\\Finder\\Finder::exclude'**

```
$finder->in(__DIR__)->exclude('ruby');
```

Т.к. Finder использует PHP итераторы, ему можно передать любой URL с поддерживаемым протоколом protocol:

```
$finder->in('ftp://example.com/pub/');
```

Также он работает с пользовательскими потоками:

```
use Symfony\Component\Finder\Finder;

$s3 = new \Zend_Service_Amazon_S3($key, $secret);
$s3->registerStreamWrapper("s3");

$finder = new Finder();
$finder->name('photos*')->size('< 100K')->date('since 1 hour ago');
foreach ($finder->in('s3://bucket-name') as $file) {
    // do something

    print $file->getFilename()."\n";
}
```

Примечание: В документации Streams можно узнать как создавать свои собственные потоки.

Файлы или каталоги

По-умолчанию, Finder возвращает файлы или каталоги; но методами **:method:'Symfony\\Component\\Finder\\Finder::files'** и **:method:'Symfony\\Component\\Finder\\Finder::directories'** можно управлять его поведением:

```
$finder->files();

$finder->directories();
```

Если хотите следовать по ссылкам, используйте метод `followLinks()`:

```
$finder->files()->followLinks();
```

По-умолчанию, итератор игнорирует популярные файлы VCS. Данное поведение может быть изменено с помощью метода `ignoreVCS()`:

```
$finder->ignoreVCS(false);
```

Сортировка

Сортировка результатов по имени или типу (каталоги первыми, файлы последними):

```
$finder->sortByName();

$finder->sortByType();
```

Примечание: Обратите внимание, что методам `sort*` требуется получить все подходящие под обработку объекты. Данная процедура при больших объемах весьма медленна.

Также можно определить свои собственные алгоритмы сортировки с помощью метода `sort()`:

```
$sort = function (\SplFileInfo $a, \SplFileInfo $b)
{
    return strcmp($a->getRealpath(), $b->getRealpath());
};

$finder->sort($sort);
```

Имена файлов

Наложить ограничения по имени файлов можно с помощью метода **:method:‘Symfony\\Component\\Finder\\Finder::name‘**:

```
$finder->files()->name('*.php');
```

Метод `name()` принимает строки, регулярные выражения или шаблоны:

```
$finder->files()->name('/\.php$/');
```

Метод `notNames()` исключает файлы по шаблону:

```
$finder->files()->notName('*.rb');
```

Размер файла

Ограничить размер файлов можно с помощью метода Restrict files by size with the **:method:‘Symfony\\Component\\Finder\\Finder::size‘**:

```
:method:‘Symfony\\Component\\Finder\\Finder::size‘ method::

    $finder->files()->size(‘< 1.5K’);
```

Ограничить размер в рамках можно с помощью связанных вызовов:

Оператор сравнения может быть любым из следующих: `>`, `>=`, `<`, ‘<=’, ‘==’.

Целевое значение может использовать приставки (`k`, `ki`) килобайты, (`m`, `mi`) мегабайты, или (`g`, `gi`) гигабайты. Те которые используют суффиксы `i` (киби/миби и т.д.) в названии являются версиями `2**n` согласно стандарту‘IEC standard‘_.

Дата файла

С помощью метода **:method:‘Symfony\\Component\\Finder\\Finder::date‘** можно наложить ограничения на файлы по дате последнего изменения:

```
$finder->date('since yesterday');
```

Оператор сравнения может быть любым из следующих:`>`, `>=`, `<`, ‘<=’, ‘==’. Также можно использовать псевдонимы `since` или `after` для оператора `>`, и `until` или `before` в качестве `<`.

Целевое значение может быть любой датой поддерживаемой функцией strtotime

Глубина каталогов

По-умолчанию Finder просматривает каталоги рекурсивно. Ограничить глубину поиска можно с помощью метода **:method:‘Symfony\\Component\\Finder\\Finder::depth‘**:

```
$finder->depth('== 0');
$finder->depth('< 3');
```

Фильтрация

Ограничить результаты поиска согласно собственным параметрам, можно используя метод **:method:‘Symfony\\Component\\Finder\\Finder::filter‘**:

```
$filter = function (\SplFileInfo $file)
{
    if (strlen($file) > 10) {
        return false;
    }
};

$finder->files()->filter($filter);
```

Метод `filter()` получает замыкание в качестве аргумента. Для каждого подходящего файла, замыкание вызывается с аргументом в виде объекта который является экземпляром класса **:phpclass:‘SplFileInfo‘**. Файл исключается из множества результатов если замыкание возвращает `false`.

- **Процесс**
  - *Как создать и разместить Проект на Symfony2 в git-репозитории*
- **Контроллеры**
  - *Как создать собственные страницы ошибок*
  - *Как определять Контроллеры в качестве сервисов*
- **Маршрутизация**
  - *Как заставить маршрутизатор всегда использовать HTTPS или HTTP*
  - `/cookbook/routing/slash_in_parameter`
- **Работа с JavaScript и CSS ресурсами**
  - `/cookbook/assetic/asset_management`
  - `/cookbook/assetic/yuicompressor`
  - `/cookbook/assetic/jpeg_optimize`
  - `/cookbook/assetic/apply_to_option`
- **Взаимодействие с СУБД (Doctrine)**
  - `/cookbook/doctrine/file_uploads`
  - `/cookbook/doctrine/common_extensions`
  - `/cookbook/doctrine/event_listeners_subscribers`
  - `/cookbook/doctrine/dbal`
  - `/cookbook/doctrine/reverse_engineering`
  - `/cookbook/doctrine/multiple_entity_managers`
  - `/cookbook/doctrine/custom_dql_functions`

- **Формы и Валидация**
  - /cookbook/form/form_customization
  - /cookbook/form/create_custom_field_type
  - /cookbook/validation/custom_constraint
  - (doctrine) /cookbook/doctrine/file_uploads
- **Конфигурация и Сервис Контейнер**
  - /cookbook/configuration/environments
  - /cookbook/configuration/external_parameters
  - /cookbook/service_container/factories
  - /cookbook/service_container/parentservices
  - /cookbook/service_container/scopes
  - /cookbook/configuration/pdo_session_storage
- **Пакеты**
  - /cookbook/bundles/best_practices
  - /cookbook/bundles/inheritance
  - /cookbook/bundles/override
  - /cookbook/bundles/extension
- **Работа с Email**
  - *Как отправлять электронную почту*
  - *Как использовать Gmail для отправки электронных писем*
  - /cookbook/email/dev_environment
  - /cookbook/email/spool
- **Тестирование**
  - *Как смоделировать HTTP аутентификацию в Функциональном тесте*
  - *Как тестировать взаимодействие с несколькими клиентами*
  - *Как использовать профилировщик в Функциональном тесте*
  - /cookbook/testing/doctrine
- **Безопасность**
  - /cookbook/security/remember_me
  - /cookbook/security/voters

  - /cookbook/security/acl
  - /cookbook/security/acl_advanced
  - /cookbook/security/force_https
  - /cookbook/security/form_login
  - /cookbook/security/securing_services
  - /cookbook/security/entity_provider
  - /cookbook/security/custom_provider
  - /cookbook/security/custom_authentication_provider
  - /cookbook/security/target_path
- **Кэширование**
  - *Как использовать Varnish для ускорения работы сайта*
- **Шаблоны**
  - *Внедрение переменных во все шаблоны (т.н. Глобальные переменные)*
  - *Как использовать PHP шаблоны вместо Twig*
- **Инструменты, Логгирование внутренние компоненты**
  - *Как автоматически загружать классы*
  - *Как искать файлы*
  - /cookbook/console
  - /cookbook/debugging
  - *Как использовать Monolog для журналирования*
- **Web Сервисы**
  - /cookbook/web_services/php_soap_extension
- **Расширение Symfony**
  - /cookbook/event_dispatcher/class_extension
  - /cookbook/event_dispatcher/method_behavior
  - /cookbook/request/mime_type
  - /cookbook/profiler/data_collector
- **Symfony2 для разработчиков symfony1**
  - /cookbook/symfony1

Прочитайте *Рецепты*.

Часть IV

Справочные документы

С ними вы быстро получите ответы:

Reference Documents

4.1 Справочник типов полей для форм

4.1.1 Тип поля collection

См. класс `Symfony\Component\Form\Extension\Core\Type\CollectionType`.

4.1.2 Поле date

Это поле позволяет пользователю изменять информацию о дате при помощи различных HTML-элементов.

Соответствующие данные для этого поля должны быть в виде объекта `\DateTime`, строки, таймштампа или массива. Если опция **'input'_** указана верно, поле будет заботиться о деталях самостоятельно.

Это поле может быть отображено как один текстбокс, три текстбокса (месяц, день, год) или же три селектбокса (см. опцию *widget_*).

Тип данных	`\DateTime`, string, timestamp, или array (см. опцию `input`)
Отображается как	один text box или три поля select
Опции	• **'widget'_** • **'input'_** • empty_value • **'years'_** • **'months'_** • **'days'_** • **'format'_** • **'pattern'_** • **'data_timezone'_** • **'user_timezone'_**
Родительский тип	`field` (если текст), иначе `form`
Класс	`Symfony\Component\Form\Extension\Core\Type\DateTy`

Примеры использования

Это поле имеет много натроект, но оно просто в использовании. Наиболее важными опциями являются `input` и `widget`.

Полагая, что у вас есть поле `publishedAt`, которое содержит дату в виде обхекта `DateTime`. Следующий код конфигурирует поле `date` для этого поля в вите трёх селектов:

```
<?php
$builder->add('publishedAt', 'date', array(
    'input'  => 'datetime',
    'widget' => 'choice',
));
```

Опция `input` *должна* всегда соответствовать типу данных, которые используются для этого поля. Например, если поле `publishedAt` использует unix timestamp, вам нужно присвоить опции `input` значение `timestamp`:

```
<?php
$builder->add('publishedAt', 'date', array(
    'input'  => 'timestamp',
    'widget' => 'choice',
));
```

Это поле также подерживает `array` и `string` в качестве валидных опций для `input`.

Опции поля

empty_value

type: `string` | `array`

Если опция widget имеет значение `choice`, тогда это поле будет представлено последовательностью селектбоксов. Опция `empty_value` может быть использована для добавления пустой опции в начале списка селектбокса:

```
<?php
$builder->add('dueDate', 'date', array(
    'empty_value' => '',
));
```

Также вы можете указать строку, которая будет значением "пустой" опции в selectbox:

```
<?php
$builder->add('dueDate', 'date', array(
    'empty_value' => array('year' => 'Year', 'month' => 'Month', 'day' => 'Day')
));
```

Форма состоит из *полей*, каждое из которых создаётся при помощи *типов* полей (например, тип `text`, тип `choice` и т.д.). Symfony2 поставляется с большим количеством типов полей, которые могут быть использованы в вашем приложении.

4.1.3 Поддерживаемые типы полей

Следующие типы полей доступны в Symfony2:

Текстовые поля

- `text`
- `textarea`
- `email`
- `integer`
- `money`
- `number`
- `password`
- `percent`
- `search`

- `url`

Поля для выбора

- `choice`
- `entity`
- `country`
- `language`
- `locale`
- `timezone`

Поля для даты и времени

- *date*
- `datetime`
- `time`
- `birthday`

Прочие поля

- `checkbox`
- `file`
- `radio`

Группы полей

- *collection*
- `repeated`

Скрытые поля

- `hidden`
- `csrf`

Базовые поля

- `field`
- `form`

4.2 Справочник функций Twig для работы с формами

Это справочное руководство содержит все функции Twig, доступные при отображении форм. Имеется несколько таких функций, каждая из которых отвечает за отображение своей части формы (метки, ошибки, виджеты и т.д.).

4.2.1 form_label(form.name, label, variables)

Отображает метку для данного поля. Опционально вы можете передать свой текст для метки в качестве второго аргумента.

```
{{ form_label(form.name) }}

{# Эти две формы записи эквивалентны #}
{{ form_label(form.name, 'Your Name', { 'attr': {'class': 'foo'} }) }}
{{ form_label(form.name, null, { 'label': 'Your name', 'attr': {'class': 'foo'} }) }}
```

4.2.2 form_errors(form.name)

Отображает ошибки для данного поля.

```
{{ form_errors(form.name) }}

{# отображает "глобальные" ошибки формы #}
{{ form_errors(form) }}
```

4.2.3 form_widget(form.name, variables)

Отображает HTML виджет для данного поля. Если вы используете этот хелпер для всей формы или набора полей, они будут полностью отображены.

```
{# отображает виджет и добавляет ему класс "foo" #}
{{ form_widget(form.name, { 'attr': {'class': 'foo'} }) }}
```

Второй аргумент `form_widget` - это массив переменных. Наиболее типичной из них является `attr`, который представляет собой мссив HTML-атрибутов, которые будут применены к HTML-виджету. В некоторых случаях, ряд типов полей также имеют другие опции, относящиеся к отображению шаблона. Эти опции описаны для каждого из таких типов полей.

4.2.4 form_row(form.name, variables)

Отображает "строку" для заданного поля, которая является комбинацией метки, ошибок и виджета.

```
{# отображаем строку, метку заменяем на "foo" #}
{{ form_row(form.name, { 'label': 'foo' }) }}
```

Второй аргумент `form_row` - это массив. Шаблоны, предоставляемые в Symfony позволяют изменять лишь метку, как, и показано в примере выше.

4.2.5 form_rest(form, variables)

Отображает все поля формы, которые ещё не были отображены. Хорошим подходом является всегда использовать этот хелпер в ваших формах для отображения скрытых полей, а также полей, которые вы могли случайно забыть.

```
{{ form_rest(form) }}
```

4.2.6 form_enctype(form)

Если поле содержит хоть одно поле для загрузки файла, этот хелпер отобразит атрибут формы `enctype="multipart/form-data"`. Хорошей практикой является его использование во всех формах:

```
<form action="{{ path('form_submit') }}" method="post" {{ form_enctype(form) }}>
```

4.3 Таги Dependency Injection

Список доступных тагов:

- `data_collector`
- `form.type`
- `form.type_extension`

- `form.type_guesser`
- `kernel.cache_warmer`
- `kernel.event_listener`
- `kernel.event_subscriber`
- `monolog.logger`
- `monolog.processor`
- `templating.helper`
- `routing.loader`
- `translation.loader`
- `twig.extension`
- `validator.initializer`

4.3.1 Подключаем пользовательские хелперы в PHP шаблоны

Для подключения пользовательского хелпера, добавьте его в виде службы в один из ваших конфигурационных файлов, пометьте его тагом `templating.helper` и определите атрибут `alias` (в шаблонах хелпер будет доступен через этот алиас):

- *YAML*

```
services:
    templating.helper.your_helper_name:
        class: Fully\Qualified\Helper\Class\Name
        tags:
            - { name: templating.helper, alias: alias_name }
```

- *XML*

```
<service id="templating.helper.your_helper_name" class="Fully\Qualified\Helper\Class\Name">
    <tag name="templating.helper" alias="alias_name" />
</service>
```

- *PHP*

```
<?php
$container
    ->register('templating.helper.your_helper_name', 'Fully\Qualified\Helper\Class\Name')
    ->addTag('templating.helper', array('alias' => 'alias_name'))
;
```

4.3.2 Подключение пользовательских расширений для Twig

Для активации расширения Twig добавьте его в виде службы в один из ваших конфигурационных файлов и пометьте эту службу тагом `twig.extension`:

- *YAML*

```
services:
    twig.extension.your_extension_name:
        class: Fully\Qualified\Extension\Class\Name
        tags:
            - { name: twig.extension }
```

- *XML*

```
<service id="twig.extension.your_extension_name" class="Fully\Qualified\Extension\Class\Nam
    <tag name="twig.extension" />
</service>
```

- *PHP*

```
<?php
$container
    ->register('twig.extension.your_extension_name', 'Fully\Qualified\Extension\Class\Name'
    ->addTag('twig.extension')
;
```

О том, как создавать расширения для Twig, читайте в документации Twig.

4.3.3 Подключение пользовательских слушателей

Для того чтобы подключить слушателя (listener), добавьте его в виде службы в один из ваших конфигурационных файлов и пометьте её тагом `kernel.event_listener`. Вы можете указать наименование события, которое будет слушать ваша служба, а также метод, который будет вызван:

- *YAML*

```
services:
    kernel.listener.your_listener_name:
        class: Fully\Qualified\Listener\Class\Name
        tags:
            - { name: kernel.event_listener, event: xxx, method: onXxx }
```

- *XML*

```
<service id="kernel.listener.your_listener_name" class="Fully\Qualified\Listener\Class\Name
    <tag name="kernel.event_listener" event="xxx" method="onXxx" />
</service>
```

- *PHP*

```
<?php
$container
    ->register('kernel.listener.your_listener_name', 'Fully\Qualified\Listener\Class\Name')
    ->addTag('kernel.event_listener', array('event' => 'xxx', 'method' => 'onXxx'))
;
```

Примечание: Вы также можете указать приоритет (положительное или отрицательное целое число) в виде атрибута тага kernel.event_listener (также как метод или событие). Это позволяет вам управлять очерёдностью вызова слушателей одного и того же события.

4.3.4 Подключение пользовательских Подписчиков

Добавлено в версии 2.1. Для активации подписчика, добавьте его в виде службы в один из конфигурационных файлов и пометьте его тагом `kernel.event_subscriber`:

- *YAML*

```
services:
    kernel.subscriber.your_subscriber_name:
        class: Fully\Qualified\Subscriber\Class\Name
        tags:
            - { name: kernel.event_subscriber }
```

- *XML*

```
<service id="kernel.subscriber.your_subscriber_name" class="Fully\Qualified\Subscriber\Clas
    <tag name="kernel.event_subscriber" />
</service>
```

- *PHP*

```
<?php
$container
    ->register('kernel.subscriber.your_subscriber_name', 'Fully\Qualified\Subscriber\Class\
    ->addTag('kernel.event_subscriber')
;
```

Примечание: Служба должна реализовывать интерфейс `SymfonyComponentEventDispatcherEventSubscriberInterface`.

Примечание: Если ваша служба создаётся фабрикой, вы **ДОЛЖНЫ** корректно указать параметр `class` для этого тага, иначе работать не будет.

4.3.5 Подключение пользовательских шаблонизаторов

Для того чтобы активировать ещё один шаблонизатор, добавьте его в виде службы в один из конфигурационных файлов и пометьте её тагом `templating.engine`:

- *YAML*

```
services:
    templating.engine.your_engine_name:
        class: Fully\Qualified\Engine\Class\Name
        tags:
            - { name: templating.engine }
```

- *XML*

```
<service id="templating.engine.your_engine_name" class="Fully\Qualified\Engine\Class\Name">
    <tag name="templating.engine" />
</service>
```

- *PHP*

```
<?php
$container
    ->register('templating.engine.your_engine_name', 'Fully\Qualified\Engine\Class\Name')
    ->addTag('templating.engine')
;
```

4.3.6 Подключение пользовательских загрузчиков для Маршрутов

Для того чтобы подключить загрузчик маршрутов, добавьте его в виде службы в один из конфигурационных файлов и пометьте её тагом `routing.loader`:

- *YAML*

```
services:
    routing.loader.your_loader_name:
        class: Fully\Qualified\Loader\Class\Name
        tags:
            - { name: routing.loader }
```

- *XML*

```
<service id="routing.loader.your_loader_name" class="Fully\Qualified\Loader\Class\Name">
    <tag name="routing.loader" />
</service>
```

4.3.7 Использование отдельного канала для логгирования в Monolog

Монолог позволяет вам делить обработчики между несколькими каналами логгирования. Служба логгера использует канал `app`, но вы можете изменять канал при инъекции логгера в службу.

- *YAML*

```
services:
    my_service:
        class: Fully\Qualified\Loader\Class\Name
        arguments: [@logger]
        tags:
            - { name: monolog.logger, channel: acme }
```

- *XML*

```
<service id="my_service" class="Fully\Qualified\Loader\Class\Name">
    <argument type="service" id="logger" />
    <tag name="monolog.logger" channel="acme" />
</service>
```

- *PHP*

```
<?php
$definition = new Definition('Fully\Qualified\Loader\Class\Name', array(new Reference('logg
$definition->addTag('monolog.logger', array('channel' => 'acme'));
$container->register('my_service', $definition);;
```

Примечание: Это работает, только когда служба логгера инжектится через аргумент конструктора, при использовании сеттера работать не будет.

4.3.8 Добавление процессоров для Monolog

Monolog позволяет вам добавлять процессоры в логгер или в хэндлеры для добавления дополнительных данных в записи. Процессор получает запись в качестве аргумента и должен вернуть её после добавления дополнительных данных в атрибут записи `extra`.

Давайте посмотрим, как вы можете использовать встроенный `IntrospectionProcessor` для добавления файла, строки, класса и метода, где был вызван логгер.

Вы можете добавить процессор глобально:

- *YAML*

```
services:
    my_service:
        class: Monolog\Processor\IntrospectionProcessor
        tags:
            - { name: monolog.processor }
```

- *XML*

```
<service id="my_service" class="Monolog\Processor\IntrospectionProcessor">
    <tag name="monolog.processor" />
</service>
```

- *PHP*

```
<?php
$definition = new Definition('Monolog\Processor\IntrospectionProcessor');
$definition->addTag('monolog.processor');
$container->register('my_service', $definition);
```

Совет: Если служба не может быть вызвана (при помощи `__invoke`) вы можете добавить атрибут `method` в таг, чтобы использовать указанный метод.

Вы также можете добавить процессор для некоторого хэндлера при помощи атрибута `handler`:

- *YAML*

```
services:
    my_service:
        class: Monolog\Processor\IntrospectionProcessor
        tags:
            - { name: monolog.processor, handler: firephp }
```

- *XML*

```
<service id="my_service" class="Monolog\Processor\IntrospectionProcessor">
    <tag name="monolog.processor" handler="firephp" />
</service>
```

- *PHP*

```
<?php
$definition = new Definition('Monolog\Processor\IntrospectionProcessor');
$definition->addTag('monolog.processor', array('handler' => 'firephp');
$container->register('my_service', $definition);
```

Вы также можете добавить процессор для некоторого канала логгинга при помощи атрибута `channel`. Этот атрибут зарегистрирует процессор только для канала `security`, используемого в компоненте Security:

- *YAML*

```
services:
    my_service:
        class: Monolog\Processor\IntrospectionProcessor
        tags:
            - { name: monolog.processor, channel: security }
```

- *XML*

```
<service id="my_service" class="Monolog\Processor\IntrospectionProcessor">
    <tag name="monolog.processor" channel="security" />
</service>
```

- *PHP*

```
<?php
$definition = new Definition('Monolog\Processor\IntrospectionProcessor');
$definition->addTag('monolog.processor', array('channel' => 'security');
$container->register('my_service', $definition);
```

Примечание: Вы не можете использовать оба атрибута `handler` и `channel` для одного и того же тага так как хэндлеры доступны для всех каналов.

- **Конфигурирование**:

  Если вы хоть раз задавались вопросом, что означают различные опции в ваших конфигурационных файлах, таких как `app/config/config.yml`, то этот раздел для вас. Здесь все конфигурационные параметры разбиты на подразделы по ключу (например `framework`), которые определяют все возможные секции в настройках Symfony2.

  - `framework`
  - `doctrine`
  - `security`
  - `assetic`

  - `swiftmailer`
  - `twig`
  - `monolog`
  - `web_profiler`

- **Формы и валидация**
  - *Формы, справочник типов полей*
  - `Валидация, справочник ограничений`
  - *Twig: справочник функций для работы с формами*

- **Остальные разделы**
  - *Таги Dependency Injection*
  - `/reference/YAML`
  - `/reference/requirements`

- **Конфигурирование**:

  Если вы хоть раз задавались вопросом, что означают различные опции в ваших конфигурационных файлах, таких как `app/config/config.yml`, то этот раздел для вас. Здесь все конфигурационные параметры разбиты на подразделы по ключу (например `framework`), которые определяют все возможные секции в настройках Symfony2.

  - `framework`
  - `doctrine`
  - `security`
  - `assetic`
  - `swiftmailer`
  - `twig`
  - `monolog`
  - `web_profiler`

- **Формы и валидация**
  - *Формы, справочник типов полей*
  - `Валидация, справочник ограничений`
  - *Twig: справочник функций для работы с формами*

- **Остальные разделы**

- *Тагu Dependency Injection*
- /reference/YAML
- /reference/requirements

Часть V

Пакеты

Symfony Standard Edition поставляется с несколькими пакетами. Узнайте о них больше:

Глава 5

Symfony SE Bundles

- SensioFrameworkExtraBundle
- SensioGeneratorBundle
- JMSSecurityExtraBundle
- DoctrineFixturesBundle
- DoctrineMigrationsBundle
- DoctrineMongoDBBundle
- SensioFrameworkExtraBundle
- SensioGeneratorBundle
- JMSSecurityExtraBundle
- DoctrineFixturesBundle
- DoctrineMigrationsBundle
- DoctrineMongoDBBundle

Часть VI

Участие в проекте

Примите участие в развитии Symfony2:

Содействие

Помощь в разработке Symfony2:

6.1 Содействие в коде

6.1.1 Сообщение об ошибке

Когда находите ошибку в Symfony2, любезно просим вас сообщить о ней. Это поможет сделать Symfony2 лучше.

Осторожно: Если вы полагаете что нашли брешь в безопасности, воспользуйтесь специальной *процедурой*.

Перед отправкой ошибки:

- Дважды проверьте официальную documentation чтобы удостовериться что вы правильно используете фреймворк;
- Попросите содействия в users mailing-list, на forum или #symfony IRC channel если не уверены в том что это действительно ошибка.

Если проблема без сомнений выглядит как ошибка, сообщите о ней, используя tracker и соблюдая несколько основных правил:

- Используйте поле title чтобы чётко обозначить вопрос;
- Опишите шаги, необходимые для воспроизведения ошибки с короткими примерами кода (ещё лучше предоставить модульный тест, показывающий её);
- Предоставьте как можно более детальную информацию о своём окружении (ОС, версия PHP, версия Symfony, включённые расширения, ...);

- *(по усмотрению)* Прикрепите `патч`.

6.1.2 Сообщение о бреши в безопасности

Нашли проблему в безопасности Symfony2? Не используйте mailing-list или баг трекер. Вместо них все проблемы безопасности должны отправляться на **security [at] symfony-project.com**. Письма, отправленные туда, перенаправляются в закрытый список рассылки для разработчиков Symfony.

Для каждого сообщения мы пытаемся подтвердить уязвимость. Когда она подтвердится, разработчики трудятся над её решением, согласно этим шагам:

1. Отправляют подтверждение отправителю;
2. Работают над патчем;
3. Описывают уязвимость, возможные атаки и как закрыть/модернизировать пострадавшие приложения;
4. Применяют патч ко всем поддерживаемым версиям Symfony;
5. Публикуют сообщение на официальном блоге Symfony.

Примечание: Пока мы работаем над патчем, пожалуйста, не разглашайте проблему.

6.1.3 Соглашения

Документ `стандарты` описывает стандарты кодирования для проекта Symfony2, его внутренних и сторонних бандлов. Этот документ описывает стандарты кодирования и соглашения, используемые в ядре фреймворка чтобы сделать его более последовательным и предсказуемым. Можете следовать им в своём коде, но в этом нет особой нужды.

Имена методов

Когда объект имеет "главную" множественную связь со связанными "предметами" (объекты, параметры и т. д.), имена методов стандартизуются:

- `get()`
- `set()`
- `has()`
- `all()`

- `replace()`
- `remove()`
- `clear()`
- `isEmpty()`
- `add()`
- `register()`
- `count()`
- `keys()`

Использование этих методов применимо только когда ясно что существует основная связь:

- `CookieJar` имеет множество `Cookie`;
- Служба `Container` имеет множество служб и множество параметров (т. к. службы это главная связь, то мы используем соглашения для имён для этой связи);
- Консоль `Input` имеет множество аргументов и множество опций. Здесь нет "основной" связи, поэтому соглашения для имён не применяются.

Для множественных связей, к которым не применяется соглашение, должны использоваться следующие методы (где `XXX` это имя соотвествующего предмета):

Главная связь	Другие связи
`get()`	`getXXX()`
`set()`	`setXXX()`
`has()`	`hasXXX()`
`all()`	`getXXXs()`
`replace()`	`setXXXs()`
`remove()`	`removeXXX()`
`clear()`	`clearXXX()`
`isEmpty()`	`isEmptyXXX()`
`add()`	`addXXX()`
`register()`	`registerXXX()`
`count()`	`countXXX()`
`keys()`	не доступно

6.1.4 Лицензия Symfony2

Symfony2 выпущен под лицензией MIT.

Согласно Wikipedia:

> "Это разрешающая лицензия, означает что она разрешает использование в проприетарных программах, при условии что лицензия распространяется с этой программой. Также она GPL-совместима, это значит что GPL разрешает комбинирование и распространение с программами, использующими лицензию MIT."

Лицензия

6.2 Содействие в документации

6.2.1 Содействие в документации

Документация также важна как и код. Она следует тем же принципам: DRY, тесты, простота в обслуживании, расширяемость, оптимизация и рефакторинг вот некоторые из них. И конечно же документация имеет ошибки, опечатки, трудночитаемые инструкции и т. д.

Содействие

Перед содействием вам необходимо ознакомиться с *языком разметки*, используемым в документации.

Документация Symfony2 располагается в Git репозитории:

```
git://github.com/symfony/symfony-docs.git
```

Если вы хотите прислать патч, клонируйте официальный репозиторий с документацией:

```
$ git clone git://github.com/symfony/symfony-docs.git
```

Примечание: Документация Symfony2 выходит под `лицензией` Creative Commons Attribution-Share Alike 3.0 Unported.

Сообщение об ошибке

Самое простое содействие что вы можете оказать это сообщение об: опечатке, грамматической ошибке, ошибке в примере кода, отсутствующем пояснении и т. д.

Шаги:

- Сообщить об ошибке в баг трекер;
- *(по желанию)* Прислать патч.

Перевод

Прочитайте посвящённый этому `документ`.

6.2.2 Формат документации

Документация Symfony2 использует reStructuredText как язык разметки и Sphinx для создания вывода (HTML, PDF и т. д.).

reStructuredText

reStructuredText это "легкочитаемый, что видишь то и получишь, синтаксис разметки открытым текстом и система анализа".

Узнайте больше о его синтаксисе, прочитав Symfony2 documents или reStructuredText Primer на web сайте Sphinx.

Если вы знакомы с Markdown, будьте осторожны, т. к. некоторые вещи очень знакомы, но отличаются:

- Списки начинаются с начала строки (необходимость отступа отсутствует);
- Встроенные блоки кода используют двойные кавычки (``` ``как здесь`` ```).

Sphinx

Sphinx - это система сборки, добавляющая полезные инструменты для создания документации из документов reStructuredText. Она добавляет указания и роли интерпретированного текста к стандартной reST markup.

Подсветка синтаксиса

Все примеры кода подсвечиваются по умолчанию как язык PHP. Вы можете изменить их через директиву `code-block`:

```
.. code-block:: yaml

    { foo: bar, bar: { foo: bar, bar: baz } }
```

Если ваш PHP код начинается с `<?php`, тогда используйте `html+php` как подсвечиваемый псевдо-язык:

```
.. code-block:: html+php

    <?php echo $this->foobar(); ?>
```

Примечание: Список поддерживаемых языков доступен на Pygments website.

Блоки конфигураций

Всякий раз как вы показываете конфигурацию, используйте директиву `configuration-block` чтобы отразить конфигурацию во всех поддерживаемых форматах (`PHP`, `YAML` и `XML`):

```
.. configuration-block::

    .. code-block:: yaml

        # Configuration in YAML

    .. code-block:: xml

        <!-- Configuration in XML //-->

    .. code-block:: php

        // Configuration in PHP
```

Предыдущая reST разметка отобразится следующим образом:

- *YAML*

```
# Configuration in YAML
```

- *XML*

```
<!-- Configuration in XML //-->
```

- *PHP*

```
// Configuration in PHP
```

Текущий список поддерживаемых форматов:

Формат разметки	Отображается
html	HTML
xml	XML
php	PHP
yaml	YAML
jinja	Twig
html+jinja	Twig
jinja+html	Twig
php+html	PHP
html+php	PHP
ini	INI
php-annotations	Аннотации

6.3 Содействие в коде

6.3.1 Другие источники

Чтобы быть в курсе того что происходит в сообществе, вам могут пригодиться эти дополнительные источники:

- Список открытых pull requests
- Список последних commits
- Список открытых bugs
- Список open source bundles
- **Код**:
  - *Ошибки* |
  - `Патчи` |

  - *Безопасность* |
  - Тесты |
  - Правила написания кода |
  - *Договорённости по коду* |
  - *Лицензия*
- **Документация**:
  - *Обзор* |
  - *Формат* |
  - Переводы |
  - Лицензия
- **Сообщество**:
  - Совещания в IRC |
  - *Другие источники*
- **Код**:
  - *Ошибки* |
  - Патчи |
  - *Безопасность* |
  - Тесты |
  - Правила написания кода |
  - *Договорённости по коду* |
  - *Лицензия*
- **Документация**:
  - *Обзор* |
  - *Формат* |
  - Переводы |
  - Лицензия
- **Сообщество**:
  - Совещания в IRC |
  - *Другие источники*

Алфавитный указатель

Symbols

C

D

E

F

H

I

K

L

N

P

R

S

T

V