Symfony

The Book

for Symfony 2.0

generated on May 21, 2012

Contents at a Glance

Chapter 1
Symfony2 and HTTP Fundamentals

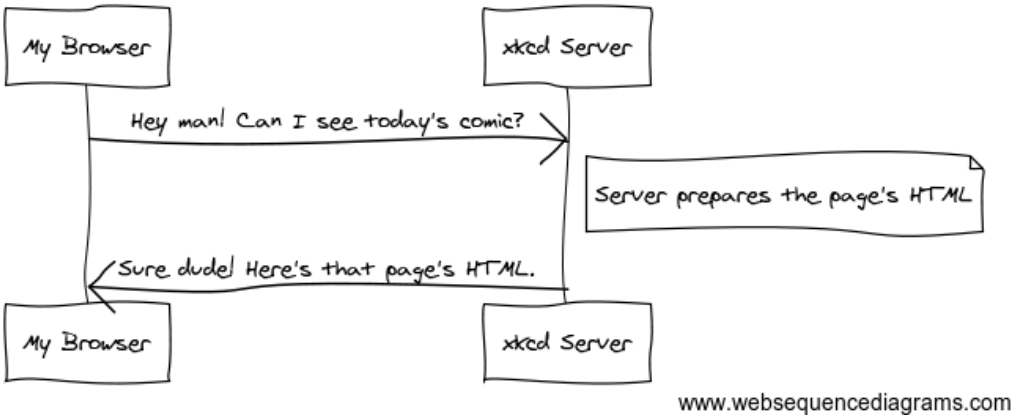
Congratulations! By learning about Symfony2, you're well on your way towards being a more *productive*, *well-rounded* and *popular* web developer (actually, you're on your own for the last part). Symfony2 is built to get back to basics: to develop tools that let you develop faster and build more robust applications, while staying out of your way. Symfony is built on the best ideas from many technologies: the tools and concepts you're about to learn represent the efforts of thousands of people, over many years. In other words, you're not just learning "Symfony", you're learning the fundamentals of the web, development best practices, and how to use many amazing new PHP libraries, inside or independent of Symfony2. So, get ready.

True to the Symfony2 philosophy, this chapter begins by explaining the fundamental concept common to web development: HTTP. Regardless of your background or preferred programming language, this chapter is a **must-read** for everyone.

HTTP is Simple

HTTP (Hypertext Transfer Protocol to the geeks) is a text language that allows two machines to communicate with each other. That's it! For example, when checking for the latest *xkcd*[1] comic, the following (approximate) conversation takes place:

1. `http://xkcd.com/`



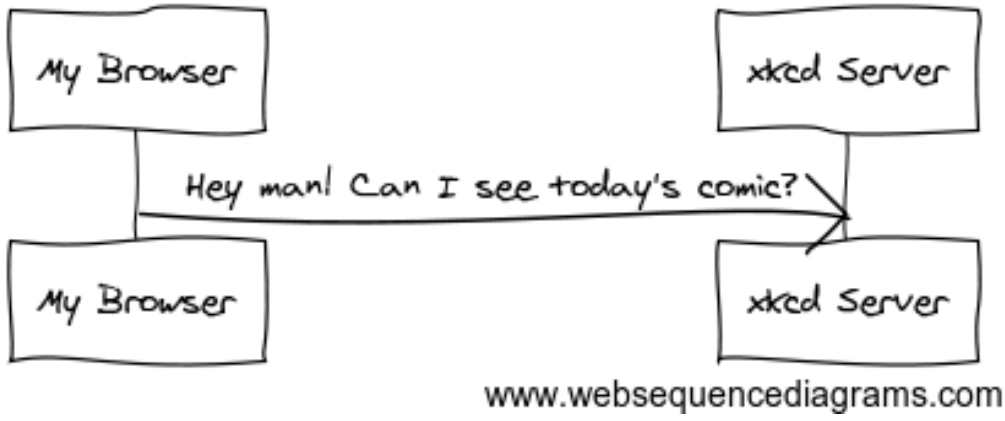
And while the actual language used is a bit more formal, it's still dead-simple. HTTP is the term used to describe this simple text-based language. And no matter how you develop on the web, the goal of your server is *always* to understand simple text requests, and return simple text responses.

Symfony2 is built from the ground-up around that reality. Whether you realize it or not, HTTP is something you use everyday. With Symfony2, you'll learn how to master it.

Step1: The Client sends a Request

Every conversation on the web starts with a *request*. The request is a text message created by a client (e.g. a browser, an iPhone app, etc) in a special format known as HTTP. The client sends that request to a server, and then waits for the response.

Take a look at the first part of the interaction (the request) between a browser and the xkcd web server:



In HTTP-speak, this HTTP request would actually look something like this:

Listing 1-1

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

This simple message communicates *everything* necessary about exactly which resource the client is requesting. The first line of an HTTP request is the most important and contains two things: the URI and the HTTP method.

The URI (e.g. `/`, `/contact`, etc) is the unique address or location that identifies the resource the client wants. The HTTP method (e.g. `GET`) defines what you want to *do* with the resource. The HTTP methods are the *verbs* of the request and define the few common ways that you can act upon the resource:

GET	Retrieve the resource from the server
POST	Create a resource on the server
PUT	Update the resource on the server
DELETE	Delete the resource from the server

With this in mind, you can imagine what an HTTP request might look like to delete a specific blog entry, for example:

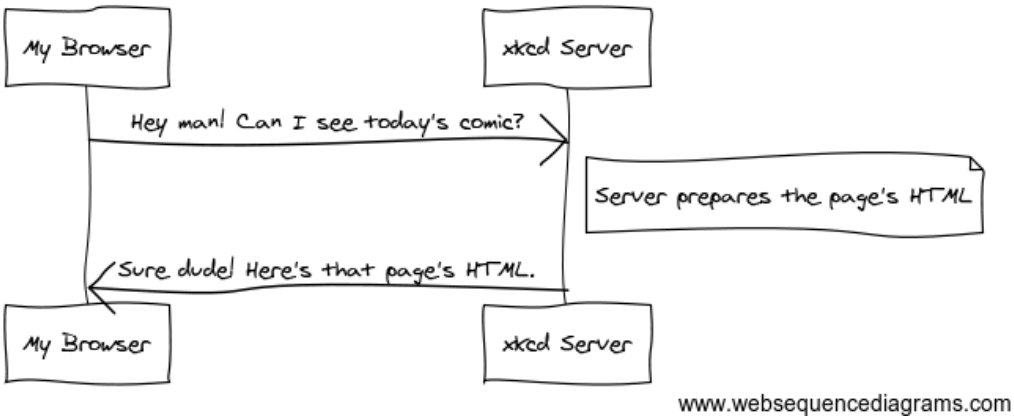
Listing 1-2
```
DELETE /blog/15 HTTP/1.1
```

There are actually nine HTTP methods defined by the HTTP specification, but many of them are not widely used or supported. In reality, many modern browsers don't support the `PUT` and `DELETE` methods.

In addition to the first line, an HTTP request invariably contains other lines of information called request headers. The headers can supply a wide range of information such as the requested `Host`, the response formats the client accepts (`Accept`) and the application the client is using to make the request (`User-Agent`). Many other headers exist and can be found on Wikipedia's *List of HTTP header fields*[2] article.

Step 2: The Server returns a Response

Once a server has received the request, it knows exactly which resource the client needs (via the URI) and what the client wants to do with that resource (via the method). For example, in the case of a GET request, the server prepares the resource and returns it in an HTTP response. Consider the response from the xkcd web server:



Translated into HTTP, the response sent back to the browser will look something like this:

Listing 1-3
```
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>
```

2. http://en.wikipedia.org/wiki/List_of_HTTP_header_fields

The HTTP response contains the requested resource (the HTML content in this case), as well as other information about the response. The first line is especially important and contains the HTTP response status code (200 in this case). The status code communicates the overall outcome of the request back to the client. Was the request successful? Was there an error? Different status codes exist that indicate success, an error, or that the client needs to do something (e.g. redirect to another page). A full list can be found on Wikipedia's *List of HTTP status codes*[3] article.

Like the request, an HTTP response contains additional pieces of information known as HTTP headers. For example, one important HTTP response header is `Content-Type`. The body of the same resource could be returned in multiple different formats like HTML, XML, or JSON and the `Content-Type` header uses Internet Media Types like `text/html` to tell the client which format is being returned. A list of common media types can be found on Wikipedia's *List of common media types*[4] article.

Many other headers exist, some of which are very powerful. For example, certain headers can be used to create a powerful caching system.

Requests, Responses and Web Development

This request-response conversation is the fundamental process that drives all communication on the web. And as important and powerful as this process is, it's inescapably simple.

The most important fact is this: regardless of the language you use, the type of application you build (web, mobile, JSON API), or the development philosophy you follow, the end goal of an application is **always** to understand each request and create and return the appropriate response.

Symfony is architected to match this reality.

To learn more about the HTTP specification, read the original *HTTP 1.1 RFC*[5] or the *HTTP Bis*[6], which is an active effort to clarify the original specification. A great tool to check both the request and response headers while browsing is the *Live HTTP Headers*[7] extension for Firefox.

Requests and Responses in PHP

So how do you interact with the "request" and create a "response" when using PHP? In reality, PHP abstracts you a bit from the whole process:

Listing 1-4

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

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

As strange as it sounds, this small application is in fact taking information from the HTTP request and using it to create an HTTP response. Instead of parsing the raw HTTP request message, PHP prepares superglobal variables such as `$_SERVER` and `$_GET` that contain all the information from the request. Similarly, instead of returning the HTTP-formatted text response, you can use the `header()` function to create response headers and simply print out the actual content that will be the content portion of the response message. PHP will create a true HTTP response and return it to the client:

3. http://en.wikipedia.org/wiki/List_of_HTTP_status_codes
4. http://en.wikipedia.org/wiki/Internet_media_type#List_of_common_media_types
5. http://www.w3.org/Protocols/rfc2616/rfc2616.html
6. http://datatracker.ietf.org/wg/httpbis/
7. https://addons.mozilla.org/en-US/firefox/addon/live-http-headers/

Listing 1-5
```
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
```

Requests and Responses in Symfony

Symfony provides an alternative to the raw PHP approach via two classes that allow you to interact with the HTTP request and response in an easier way. The ***Request***[8] class is a simple object-oriented representation of the HTTP request message. With it, you have all the request information at your fingertips:

Listing 1-6
```php
use Symfony\Component\HttpFoundation\Request;

$request = Request::createFromGlobals();

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

// retrieve GET and POST variables respectively
$request->query->get('foo');
$request->request->get('bar', 'default value if bar does not exist');

// retrieve SERVER variables
$request->server->get('HTTP_HOST');

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

// retrieve a COOKIE value
$request->cookies->get('PHPSESSID');

// retrieve an HTTP request header, with normalized, lowercase keys
$request->headers->get('host');
$request->headers->get('content_type');

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

As a bonus, the `Request` class does a lot of work in the background that you'll never need to worry about. For example, the `isSecure()` method checks the *three* different values in PHP that can indicate whether or not the user is connecting via a secured connection (i.e. `https`).

8. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/Request.html

ParameterBags and Request attributes

As seen above, the `$_GET` and `$_POST` variables are accessible via the public `query` and `request` properties respectively. Each of these objects is a *ParameterBag*[9] object, which has methods like *get()*[10], *has()*[11], *all()*[12] and more. In fact, every public property used in the previous example is some instance of the ParameterBag.

The Request class also has a public `attributes` property, which holds special data related to how the application works internally. For the Symfony2 framework, the `attributes` holds the values returned by the matched route, like `_controller`, `id` (if you have an `{id}` wildcard), and even the name of the matched route (`_route`). The `attributes` property exists entirely to be a place where you can prepare and store context-specific information about the request.

Symfony also provides a `Response` class: a simple PHP representation of an HTTP response message. This allows your application to use an object-oriented interface to construct the response that needs to be returned to the client:

Listing 1-7

```
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();
```

If Symfony offered nothing else, you would already have a toolkit for easily accessing request information and an object-oriented interface for creating the response. Even as you learn the many powerful features in Symfony, keep in mind that the goal of your application is always *to interpret a request and create the appropriate response based on your application logic*.

The `Request` and `Response` classes are part of a standalone component included with Symfony called `HttpFoundation`. This component can be used entirely independent of Symfony and also provides classes for handling sessions and file uploads.

The Journey from the Request to the Response

Like HTTP itself, the `Request` and `Response` objects are pretty simple. The hard part of building an application is writing what comes in between. In other words, the real work comes in writing the code that interprets the request information and creates the response.

Your application probably does many things, like sending emails, handling form submissions, saving things to a database, rendering HTML pages and protecting content with security. How can you manage all of this and still keep your code organized and maintainable?

Symfony was created to solve these problems so that you don't have to.

9. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/ParameterBag.html
10. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/ParameterBag.html#get()
11. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/ParameterBag.html#has()
12. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/ParameterBag.html#all()

The Front Controller

Traditionally, applications were built so that each "page" of a site was its own physical file:

Listing 1-8
```
index.php
contact.php
blog.php
```

There are several problems with this approach, including the inflexibility of the URLs (what if you wanted to change `blog.php` to `news.php` without breaking all of your links?) and the fact that each file *must* manually include some set of core files so that security, database connections and the "look" of the site can remain consistent.

A much better solution is to use a *front controller*: a single PHP file that handles every request coming into your application. For example:

`/index.php`	executes `index.php`
`/index.php/contact`	executes `index.php`
`/index.php/blog`	executes `index.php`

Using Apache's `mod_rewrite` (or equivalent with other web servers), the URLs can easily be cleaned up to be just `/`, `/contact` and `/blog`.

Now, every request is handled exactly the same. Instead of individual URLs executing different PHP files, the front controller is *always* executed, and the routing of different URLs to different parts of your application is done internally. This solves both problems with the original approach. Almost all modern web apps do this - including apps like WordPress.

Stay Organized

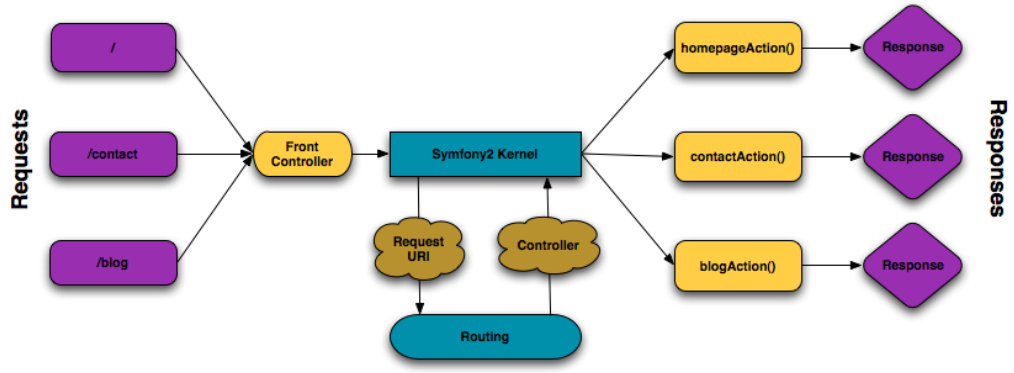
But inside your front controller, how do you know which page should be rendered and how can you render each in a sane way? One way or another, you'll need to check the incoming URI and execute different parts of your code depending on that value. This can get ugly quickly:

Listing 1-9
```php
// index.php

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

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

Solving this problem can be difficult. Fortunately it's *exactly* what Symfony is designed to do.

The Symfony Application Flow

When you let Symfony handle each request, life is much easier. Symfony follows the same simple pattern for every request:



Incoming requests are interpreted by the routing and passed to controller functions that return `Response` objects.

Each "page" of your site is defined in a routing configuration file that maps different URLs to different PHP functions. The job of each PHP function, called a *controller*, is to use information from the request - along with many other tools Symfony makes available - to create and return a `Response` object. In other words, the controller is where *your* code goes: it's where you interpret the request and create a response.

It's that easy! Let's review:

- Each request executes a front controller file;
- The routing system determines which PHP function should be executed based on information from the request and routing configuration you've created;
- The correct PHP function is executed, where your code creates and returns the appropriate `Response` object.

A Symfony Request in Action

Without diving into too much detail, let's see this process in action. Suppose you want to add a `/contact` page to your Symfony application. First, start by adding an entry for `/contact` to your routing configuration file:

Listing 1-10

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

This example uses *YAML* to define the routing configuration. Routing configuration can also be written in other formats such as XML or PHP.

When someone visits the `/contact` page, this route is matched, and the specified controller is executed. As you'll learn in the *routing chapter*, the `AcmeDemoBundle:Main:contact` string is a short syntax that points to a specific PHP method `contactAction` inside a class called `MainController`:

Listing 1-11

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

In this very simple example, the controller simply creates a `Response` object with the HTML "<h1>Contact us!</h1>". In the *controller chapter*, you'll learn how a controller can render templates, allowing your "presentation" code (i.e. anything that actually writes out HTML) to live in a separate template file. This frees up the controller to worry only about the hard stuff: interacting with the database, handling submitted data, or sending email messages.

Symfony2: Build your App, not your Tools.

You now know that the goal of any app is to interpret each incoming request and create an appropriate response. As an application grows, it becomes more difficult to keep your code organized and maintainable. Invariably, the same complex tasks keep coming up over and over again: persisting things to the database, rendering and reusing templates, handling form submissions, sending emails, validating user input and handling security.

The good news is that none of these problems is unique. Symfony provides a framework full of tools that allow you to build your application, not your tools. With Symfony2, nothing is imposed on you: you're free to use the full Symfony framework, or just one piece of Symfony all by itself.

Standalone Tools: The Symfony2 *Components*

So what *is* Symfony2? First, Symfony2 is a collection of over twenty independent libraries that can be used inside *any* PHP project. These libraries, called the *Symfony2 Components*, contain something useful for almost any situation, regardless of how your project is developed. To name a few:

- *HttpFoundation*[13] - Contains the `Request` and `Response` classes, as well as other classes for handling sessions and file uploads;
- *Routing*[14] - Powerful and fast routing system that allows you to map a specific URI (e.g. `/contact`) to some information about how that request should be handled (e.g. execute the `contactAction()` method);
- *Form*[15] - A full-featured and flexible framework for creating forms and handling form submissions;
- *Validator*[16] A system for creating rules about data and then validating whether or not user-submitted data follows those rules;
- *ClassLoader*[17] An autoloading library that allows PHP classes to be used without needing to manually `require` the files containing those classes;
- *Templating*[18] A toolkit for rendering templates, handling template inheritance (i.e. a template is decorated with a layout) and performing other common template tasks;
- *Security*[19] - A powerful library for handling all types of security inside an application;
- *Translation*[20] A framework for translating strings in your application.

Each and every one of these components is decoupled and can be used in *any* PHP project, regardless of whether or not you use the Symfony2 framework. Every part is made to be used if needed and replaced when necessary.

13. `https://github.com/symfony/HttpFoundation`
14. `https://github.com/symfony/Routing`
15. `https://github.com/symfony/Form`
16. `https://github.com/symfony/Validator`
17. `https://github.com/symfony/ClassLoader`
18. `https://github.com/symfony/Templating`
19. `https://github.com/symfony/Security`
20. `https://github.com/symfony/Translation`

The Full Solution: The Symfony2 *Framework*

So then, what *is* the Symfony2 *Framework*? The *Symfony2 Framework* is a PHP library that accomplishes two distinct tasks:

1. Provides a selection of components (i.e. the Symfony2 Components) and third-party libraries (e.g. `Swiftmailer` for sending emails);
2. Provides sensible configuration and a "glue" library that ties all of these pieces together.

The goal of the framework is to integrate many independent tools in order to provide a consistent experience for the developer. Even the framework itself is a Symfony2 bundle (i.e. a plugin) that can be configured or replaced entirely.

Symfony2 provides a powerful set of tools for rapidly developing web applications without imposing on your application. Normal users can quickly start development by using a Symfony2 distribution, which provides a project skeleton with sensible defaults. For more advanced users, the sky is the limit.

Chapter 2
Symfony2 versus Flat PHP

Why is Symfony2 better than just opening up a file and writing flat PHP?

If you've never used a PHP framework, aren't familiar with the MVC philosophy, or just wonder what all the *hype* is around Symfony2, this chapter is for you. Instead of *telling* you that Symfony2 allows you to develop faster and better software than with flat PHP, you'll see for yourself.

In this chapter, you'll write a simple application in flat PHP, and then refactor it to be more organized. You'll travel through time, seeing the decisions behind why web development has evolved over the past several years to where it is now.

By the end, you'll see how Symfony2 can rescue you from mundane tasks and let you take back control of your code.

A simple Blog in flat PHP

In this chapter, you'll build the token blog application using only flat PHP. To begin, create a single page that displays blog entries that have been persisted to the database. Writing in flat PHP is quick and dirty:

Listing 2-1
```php
<?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);
```

That's quick to write, fast to execute, and, as your app grows, impossible to maintain. There are several problems that need to be addressed:

- **No error-checking**: What if the connection to the database fails?
- **Poor organization**: If the application grows, this single file will become increasingly unmaintainable. Where should you put code to handle a form submission? How can you validate data? Where should code go for sending emails?
- **Difficult to reuse code**: Since everything is in one file, there's no way to reuse any part of the application for other "pages" of the blog.

Another problem not mentioned here is the fact that the database is tied to MySQL. Though not covered here, Symfony2 fully integrates *Doctrine*[1], a library dedicated to database abstraction and mapping.

Let's get to work on solving these problems and more.

Isolating the Presentation

The code can immediately gain from separating the application "logic" from the code that prepares the HTML "presentation":

Listing 2-2

```
<?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';
```

The HTML code is now stored in a separate file (`templates/list.php`), which is primarily an HTML file that uses a template-like PHP syntax:

Listing 2-3

```
<html>
    <head>
```

1. http://www.doctrine-project.org

```
        <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>
```

By convention, the file that contains all of the application logic - `index.php` - is known as a "controller". The term *controller* is a word you'll hear a lot, regardless of the language or framework you use. It refers simply to the area of *your* code that processes user input and prepares the response.

In this case, our controller prepares data from the database and then includes a template to present that data. With the controller isolated, you could easily change *just* the template file if you needed to render the blog entries in some other format (e.g. `list.json.php` for JSON format).

Isolating the Application (Domain) Logic

So far the application contains only one page. But what if a second page needed to use the same database connection, or even the same array of blog posts? Refactor the code so that the core behavior and data-access functions of the application are isolated in a new file called `model.php`:

Listing 2-4

```
<?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;
}
```

The filename `model.php` is used because the logic and data access of an application is traditionally known as the "model" layer. In a well-organized application, the majority of the code representing your "business logic" should live in the model (as opposed to living in a controller). And unlike in this example, only a portion (or none) of the model is actually concerned with accessing a database.

The controller (`index.php`) is now very simple:

Listing 2-5

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

$posts = get_all_posts();

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

Now, the sole task of the controller is to get data from the model layer of the application (the model) and to call a template to render that data. This is a very simple example of the model-view-controller pattern.

Isolating the Layout

At this point, the application has been refactored into three distinct pieces offering various advantages and the opportunity to reuse almost everything on different pages.

The only part of the code that *can't* be reused is the page layout. Fix that by creating a new `layout.php` file:

Listing 2-6

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

The template (`templates/list.php`) can now be simplified to "extend" the layout:

Listing 2-7

```
<?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' ?>
```

You've now introduced a methodology that allows for the reuse of the layout. Unfortunately, to accomplish this, you're forced to use a few ugly PHP functions (`ob_start()`, `ob_get_clean()`) in the template. Symfony2 uses a `Templating` component that allows this to be accomplished cleanly and easily. You'll see it in action shortly.

Adding a Blog "show" Page

The blog "list" page has now been refactored so that the code is better-organized and reusable. To prove it, add a blog "show" page, which displays an individual blog post identified by an `id` query parameter.

To begin, create a new function in the `model.php` file that retrieves an individual blog result based on a given id:

Listing 2-8
```php
// model.php
function get_post_by_id($id)
{
    $link = open_database_connection();

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

    close_database_connection($link);

    return $row;
}
```

Next, create a new file called `show.php` - the controller for this new page:

Listing 2-9
```php
<?php
require_once 'model.php';

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

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

Finally, create the new template file - `templates/show.php` - to render the individual blog post:

Listing 2-10
```php
<?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' ?>
```

Creating the second page is now very easy and no code is duplicated. Still, this page introduces even more lingering problems that a framework can solve for you. For example, a missing or invalid `id` query parameter will cause the page to crash. It would be better if this caused a 404 page to be rendered, but this can't really be done easily yet. Worse, had you forgotten to clean the `id` parameter via the `mysql_real_escape_string()` function, your entire database would be at risk for an SQL injection attack.

Another major problem is that each individual controller file must include the `model.php` file. What if each controller file suddenly needed to include an additional file or perform some other global task (e.g. enforce security)? As it stands now, that code would need to be added to every controller file. If you forget to include something in one file, hopefully it doesn't relate to security...

A "Front Controller" to the Rescue

The solution is to use a *front controller*: a single PHP file through which *all* requests are processed. With a front controller, the URIs for the application change slightly, but start to become more flexible:

Listing 2-11

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

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

The `index.php` portion of the URI can be removed if using Apache rewrite rules (or equivalent). In that case, the resulting URI of the blog show page would be simply `/show`.

When using a front controller, a single PHP file (`index.php` in this case) renders *every* request. For the blog post show page, `/index.php/show` will actually execute the `index.php` file, which is now responsible for routing requests internally based on the full URI. As you'll see, a front controller is a very powerful tool.

Creating the Front Controller

You're about to take a **big** step with the application. With one file handling all requests, you can centralize things such as security handling, configuration loading, and routing. In this application, `index.php` must now be smart enough to render the blog post list page *or* the blog post show page based on the requested URI:

Listing 2-12

```
<?php
// index.php

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

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

For organization, both controllers (formerly `index.php` and `show.php`) are now PHP functions and each has been moved into a separate file, `controllers.php`:

Listing 2-13

```
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';
}
```

As a front controller, `index.php` has taken on an entirely new role, one that includes loading the core libraries and routing the application so that one of the two controllers (the `list_action()` and `show_action()` functions) is called. In reality, the front controller is beginning to look and act a lot like Symfony2's mechanism for handling and routing requests.

Another advantage of a front controller is flexible URLs. Notice that the URL to the blog post show page could be changed from `/show` to `/read` by changing code in only one location. Before, an entire file needed to be renamed. In Symfony2, URLs are even more flexible.

By now, the application has evolved from a single PHP file into a structure that is organized and allows for code reuse. You should be happier, but far from satisfied. For example, the "routing" system is fickle, and wouldn't recognize that the list page (`/index.php`) should be accessible also via `/` (if Apache rewrite rules were added). Also, instead of developing the blog, a lot of time is being spent working on the "architecture" of the code (e.g. routing, calling controllers, templates, etc.). More time will need to be spent to handle form submissions, input validation, logging and security. Why should you have to reinvent solutions to all these routine problems?

Add a Touch of Symfony2

Symfony2 to the rescue. Before actually using Symfony2, you need to make sure PHP knows how to find the Symfony2 classes. This is accomplished via an autoloader that Symfony provides. An autoloader is a tool that makes it possible to start using PHP classes without explicitly including the file containing the class.

First, *download symfony*[2] and place it into a `vendor/symfony/` directory. Next, create an `app/bootstrap.php` file. Use it to `require` the two files in the application and to configure the autoloader:

Listing 2-14

```
<?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();
```

This tells the autoloader where the `Symfony` classes are. With this, you can start using Symfony classes without using the `require` statement for the files that contain them.

Core to Symfony's philosophy is the idea that an application's main job is to interpret each request and return a response. To this end, Symfony2 provides both a *`Request`*[3] and a *`Response`*[4] class. These classes are object-oriented representations of the raw HTTP request being processed and the HTTP response being returned. Use them to improve the blog:

2. http://symfony.com/download
3. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/Request.html
4. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/Response.html

Listing 2-15

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

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

$request = Request::createFromGlobals();

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

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

The controllers are now responsible for returning a `Response` object. To make this easier, you can add a new `render_template()` function, which, incidentally, acts quite a bit like the Symfony2 templating engine:

Listing 2-16

```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);
}

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

    return $html;
}
```

By bringing in a small part of Symfony2, the application is more flexible and reliable. The `Request` provides a dependable way to access information about the HTTP request. Specifically, the `getPathInfo()` method returns a cleaned URI (always returning `/show` and never `/index.php/show`).

So, even if the user goes to `/index.php/show`, the application is intelligent enough to route the request through `show_action()`.

The `Response` object gives flexibility when constructing the HTTP response, allowing HTTP headers and content to be added via an object-oriented interface. And while the responses in this application are simple, this flexibility will pay dividends as your application grows.

The Sample Application in Symfony2

The blog has come a *long* way, but it still contains a lot of code for such a simple application. Along the way, we've also invented a simple routing system and a method using `ob_start()` and `ob_get_clean()` to render templates. If, for some reason, you needed to continue building this "framework" from scratch, you could at least use Symfony's standalone *Routing*[5] and *Templating*[6] components, which already solve these problems.

Instead of re-solving common problems, you can let Symfony2 take care of them for you. Here's the same sample application, now built in Symfony2:

Listing 2-17
```php
<?php
// src/Acme/BlogBundle/Controller/BlogController.php

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

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

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

    public function showAction($id)
    {
        $post = $this->get('doctrine')
            ->getEntityManager()
            ->getRepository('AcmeBlogBundle:Post')
            ->find($id);

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

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

The two controllers are still lightweight. Each uses the Doctrine ORM library to retrieve objects from the database and the `Templating` component to render a template and return a `Response` object. The list template is now quite a bit simpler:

Listing 2-18
```php
<!-- src/Acme/BlogBundle/Resources/views/Blog/list.html.php -->
<?php $view->extend('::layout.html.php') ?>
```

5. https://github.com/symfony/Routing
6. https://github.com/symfony/Templating

```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>
```

The layout is nearly identical:

Listing 2-19

```php
<!-- 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>
```

We'll leave the show template as an exercise, as it should be trivial to create based on the list template.

When Symfony2's engine (called the `Kernel`) boots up, it needs a map so that it knows which controllers to execute based on the request information. A routing configuration map provides this information in a readable format:

Listing 2-20

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

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

Now that Symfony2 is handling all the mundane tasks, the front controller is dead simple. And since it does so little, you'll never have to touch it once it's created (and if you use a Symfony2 distribution, you won't even need to create it!):

Listing 2-21

```php
<?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();
```

The front controller's only job is to initialize Symfony2's engine (`Kernel`) and pass it a `Request` object to handle. Symfony2's core then uses the routing map to determine which controller to call. Just like before, the controller method is responsible for returning the final `Response` object. There's really not much else to it.

For a visual representation of how Symfony2 handles each request, see the *request flow diagram*.

Where Symfony2 Delivers

In the upcoming chapters, you'll learn more about how each piece of Symfony works and the recommended organization of a project. For now, let's see how migrating the blog from flat PHP to Symfony2 has improved life:

- Your application now has **clear and consistently organized code** (though Symfony doesn't force you into this). This promotes **reusability** and allows for new developers to be productive in your project more quickly.
- 100% of the code you write is for *your* application. You **don't need to develop or maintain low-level utilities** such as *autoloading*, *routing*, or rendering *controllers*.
- Symfony2 gives you **access to open source tools** such as Doctrine and the Templating, Security, Form, Validation and Translation components (to name a few).
- The application now enjoys **fully-flexible URLs** thanks to the `Routing` component.
- Symfony2's HTTP-centric architecture gives you access to powerful tools such as **HTTP caching** powered by **Symfony2's internal HTTP cache** or more powerful tools such as *Varnish*[7]. This is covered in a later chapter all about *caching*.

And perhaps best of all, by using Symfony2, you now have access to a whole set of **high-quality open source tools developed by the Symfony2 community**! A good selection of Symfony2 community tools can be found on *KnpBundles.com*[8].

Better templates

If you choose to use it, Symfony2 comes standard with a templating engine called *Twig*[9] that makes templates faster to write and easier to read. It means that the sample application could contain even less code! Take, for example, the list template written in Twig:

Listing 2-22

```
{# 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 %}
```

7. http://www.varnish-cache.org
8. http://knpbundles.com/
9. http://twig.sensiolabs.org

The corresponding `layout.html.twig` template is also easier to write:

Listing 2-23

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

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

Twig is well-supported in Symfony2. And while PHP templates will always be supported in Symfony2, we'll continue to discuss the many advantages of Twig. For more information, see the *templating chapter*.

Learn more from the Cookbook

- *How to use PHP instead of Twig for Templates*
- *How to define Controllers as Services*

Chapter 3
Installing and Configuring Symfony

The goal of this chapter is to get you up and running with a working application built on top of Symfony. Fortunately, Symfony offers "distributions", which are functional Symfony "starter" projects that you can download and begin developing in immediately.

If you're looking for instructions on how best to create a new project and store it via source control, see Using Source Control.

Downloading a Symfony2 Distribution

First, check that you have installed and configured a Web server (such as Apache) with PHP 5.3.2 or higher. For more information on Symfony2 requirements, see the *requirements reference*. For information on configuring your specific web server document root, see the following documentation: *Apache*[1] | *Nginx*[2] .

Symfony2 packages "distributions", which are fully-functional applications that include the Symfony2 core libraries, a selection of useful bundles, a sensible directory structure and some default configuration. When you download a Symfony2 distribution, you're downloading a functional application skeleton that can be used immediately to begin developing your application.

Start by visiting the Symfony2 download page at *http://symfony.com/download*[3]. On this page, you'll see the *Symfony Standard Edition*, which is the main Symfony2 distribution. Here, you'll need to make two choices:

- Download either a `.tgz` or `.zip` archive - both are equivalent, download whatever you're more comfortable using;

1. `http://httpd.apache.org/docs/current/mod/core.html#documentroot`
2. `http://wiki.nginx.org/Symfony`
3. `http://symfony.com/download`

- Download the distribution with or without vendors. If you have *Git*[4] installed on your computer, you should download Symfony2 "without vendors", as it adds a bit more flexibility when including third-party/vendor libraries.

Download one of the archives somewhere under your local web server's root directory and unpack it. From a UNIX command line, this can be done with one of the following commands (replacing `###` with your actual filename):

Listing 3-1

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

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

When you're finished, you should have a `Symfony/` directory that looks something like this:

Listing 3-2

```
www/ <- your web root directory
    Symfony/ <- the unpacked archive
        app/
            cache/
            config/
            logs/
        src/
            ...
        vendor/
            ...
        web/
            app.php
            ...
```

Updating Vendors

Finally, if you downloaded the archive "without vendors", install the vendors by running the following command from the command line:

Listing 3-3

```
php bin/vendors install
```

This command downloads all of the necessary vendor libraries - including Symfony itself - into the `vendor/` directory. For more information on how third-party vendor libraries are managed inside Symfony2, see "*Managing Vendor Libraries with bin/vendors and deps*".

Configuration and Setup

At this point, all of the needed third-party libraries now live in the `vendor/` directory. You also have a default application setup in `app/` and some sample code inside the `src/` directory.

Symfony2 comes with a visual server configuration tester to help make sure your Web server and PHP are configured to use Symfony. Use the following URL to check your configuration:

Listing 3-4

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

If there are any issues, correct them now before moving on.

4. `http://git-scm.com/`

Setting up Permissions

One common issue is that the `app/cache` and `app/logs` directories must be writable both by the web server and the command line user. On a UNIX system, if your web server user is different from your command line user, you can run the following commands just once in your project to ensure that permissions will be setup properly. Change `www-data` to your web server user:

1. Using ACL on a system that supports chmod +a

Many systems allow you to use the `chmod +a` command. Try this first, and if you get an error - try the next method:

Listing 3-5
```
rm -rf app/cache/*
rm -rf app/logs/*

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

2. Using Acl on a system that does not support chmod +a

Some systems don't support `chmod +a`, but do support another utility called `setfacl`. You may need to *enable ACL support*[5] on your partition and install setfacl before using it (as is the case with Ubuntu), like so:

Listing 3-6
```
sudo setfacl -R -m u:www-data:rwx -m u:`whoami`:rwx app/cache app/logs
sudo setfacl -dR -m u:www-data:rwx -m u:`whoami`:rwx app/cache app/logs
```

Note that not all web servers run as the user `www-data`. You have to check which user the web server is being run as and put it in for `www-data`. This can be done by checking your process list to see which user is running your web server processes.

3. Without using ACL

If you don't have access to changing the ACL of the directories, you will need to change the umask so that the cache and log directories will be group-writable or world-writable (depending if the web server user and the command line user are in the same group or not). To achieve this, put the following line at the beginning of the `app/console`, `web/app.php` and `web/app_dev.php` files:

Listing 3-7
```
umask(0002); // This will let the permissions be 0775

// or

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

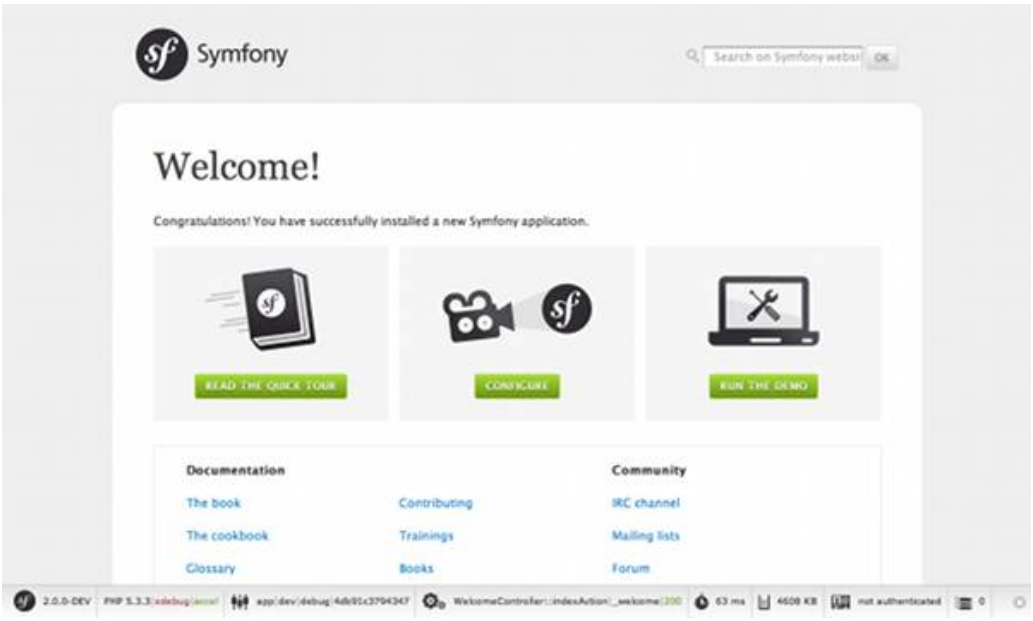
Note that using the ACL is recommended when you have access to them on your server because changing the umask is not thread-safe.

When everything is fine, click on "Go to the Welcome page" to request your first "real" Symfony2 webpage:

Listing 3-8
```
http://localhost/Symfony/web/app_dev.php/
```

Symfony2 should welcome and congratulate you for your hard work so far!

5. `https://help.ubuntu.com/community/FilePermissionsACLs`



Beginning Development

Now that you have a fully-functional Symfony2 application, you can begin development! Your distribution may contain some sample code - check the `README.rst` file included with the distribution (open it as a text file) to learn about what sample code was included with your distribution and how you can remove it later.

If you're new to Symfony, join us in the "*Creating Pages in Symfony2*", where you'll learn how to create pages, change configuration, and do everything else you'll need in your new application.

Using Source Control

If you're using a version control system like `Git` or `Subversion`, you can setup your version control system and begin committing your project to it as normal. The Symfony Standard edition *is* the starting point for your new project.

For specific instructions on how best to setup your project to be stored in git, see *How to Create and store a Symfony2 Project in git*.

Ignoring the `vendor/` Directory

If you've downloaded the archive *without vendors*, you can safely ignore the entire `vendor/` directory and not commit it to source control. With `Git`, this is done by creating and adding the following to a `.gitignore` file:

```
vendor/
```

Listing 3-9

Now, the vendor directory won't be committed to source control. This is fine (actually, it's great!) because when someone else clones or checks out the project, he/she can simply run the `php bin/vendors install` script to download all the necessary vendor libraries.

Chapter 4
Creating Pages in Symfony2

Creating a new page in Symfony2 is a simple two-step process:

- *Create a route*: A route defines the URL (e.g. `/about`) to your page and specifies a controller (which is a PHP function) that Symfony2 should execute when the URL of an incoming request matches the route pattern;
- *Create a controller*: A controller is a PHP function that takes the incoming request and transforms it into the Symfony2 `Response` object that's returned to the user.

This simple approach is beautiful because it matches the way that the Web works. Every interaction on the Web is initiated by an HTTP request. The job of your application is simply to interpret the request and return the appropriate HTTP response.

Symfony2 follows this philosophy and provides you with tools and conventions to keep your application organized as it grows in users and complexity.

Sounds simple enough? Let's dive in!

The "Hello Symfony!" Page

Let's start with a spin off of the classic "Hello World!" application. When you're finished, the user will be able to get a personal greeting (e.g. "Hello Symfony") by going to the following URL:

Listing 4-1
```
http://localhost/app_dev.php/hello/Symfony
```

Actually, you'll be able to replace `Symfony` with any other name to be greeted. To create the page, follow the simple two-step process.

The tutorial assumes that you've already downloaded Symfony2 and configured your webserver. The above URL assumes that `localhost` points to the `web` directory of your new Symfony2 project. For detailed information on this process, see the documentation on the web server you are using. Here's the relevant documentation page for some web server you might be using:

- For Apache HTTP Server, refer to *Apache's DirectoryIndex documentation*[1].

- For Nginx, refer to *Nginx HttpCoreModule location documentation*[2].

Before you begin: Create the Bundle

Before you begin, you'll need to create a *bundle*. In Symfony2, a *bundle* is like a plugin, except that all of the code in your application will live inside a bundle.

A bundle is nothing more than a directory that houses everything related to a specific feature, including PHP classes, configuration, and even stylesheets and Javascript files (see *The Bundle System*).

To create a bundle called `AcmeHelloBundle` (a play bundle that you'll build in this chapter), run the following command and follow the on-screen instructions (use all of the default options):

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

Listing 4-2

Behind the scenes, a directory is created for the bundle at `src/Acme/HelloBundle`. A line is also automatically added to the `app/AppKernel.php` file so that the bundle is registered with the kernel:

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

    return $bundles;
}
```

Listing 4-3

Now that you have a bundle setup, you can begin building your application inside the bundle.

Step 1: Create the Route

By default, the routing configuration file in a Symfony2 application is located at `app/config/routing.yml`. Like all configuration in Symfony2, you can also choose to use XML or PHP out of the box to configure routes.

If you look at the main routing file, you'll see that Symfony already added an entry when you generated the `AcmeHelloBundle`:

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

Listing 4-4

This entry is pretty basic: it tells Symfony to load routing configuration from the `Resources/config/routing.yml` file that lives inside the `AcmeHelloBundle`. This means that you place routing configuration directly in `app/config/routing.yml` or organize your routes throughout your application, and import them from here.

Now that the `routing.yml` file from the bundle is being imported, add the new route that defines the URL of the page that you're about to create:

1. http://httpd.apache.org/docs/2.0/mod/mod_dir.html
2. http://wiki.nginx.org/HttpCoreModule#location

Listing 4-5
```yaml
# src/Acme/HelloBundle/Resources/config/routing.yml
hello:
    pattern:  /hello/{name}
    defaults: { _controller: AcmeHelloBundle:Hello:index }
```

The routing consists of two basic pieces: the `pattern`, which is the URL that this route will match, and a `defaults` array, which specifies the controller that should be executed. The placeholder syntax in the pattern (`{name}`) is a wildcard. It means that `/hello/Ryan`, `/hello/Fabien` or any other similar URL will match this route. The `{name}` placeholder parameter will also be passed to the controller so that you can use its value to personally greet the user.

The routing system has many more great features for creating flexible and powerful URL structures in your application. For more details, see the chapter all about *Routing*.

Step 2: Create the Controller

When a URL such as `/hello/Ryan` is handled by the application, the `hello` route is matched and the `AcmeHelloBundle:Hello:index` controller is executed by the framework. The second step of the page-creation process is to create that controller.

The controller - `AcmeHelloBundle:Hello:index` is the *logical* name of the controller, and it maps to the `indexAction` method of a PHP class called `Acme\HelloBundle\Controller\Hello`. Start by creating this file inside your `AcmeHelloBundle`:

Listing 4-6
```php
// src/Acme/HelloBundle/Controller/HelloController.php
namespace Acme\HelloBundle\Controller;

use Symfony\Component\HttpFoundation\Response;

class HelloController
{
}
```

In reality, the controller is nothing more than a PHP method that you create and Symfony executes. This is where your code uses information from the request to build and prepare the resource being requested. Except in some advanced cases, the end product of a controller is always the same: a Symfony2 `Response` object.

Create the `indexAction` method that Symfony will execute when the `hello` route is matched:

Listing 4-7
```php
// src/Acme/HelloBundle/Controller/HelloController.php

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

The controller is simple: it creates a new `Response` object, whose first argument is the content that should be used in the response (a small HTML page in this example).

Congratulations! After creating only a route and a controller, you already have a fully-functional page! If you've setup everything correctly, your application should greet you:

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

Listing 4-8

You can also view your app in the "prod" *environment* by visiting:

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

Listing 4-9

If you get an error, it's likely because you need to clear your cache by running:

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

Listing 4-10

An optional, but common, third step in the process is to create a template.

Controllers are the main entry point for your code and a key ingredient when creating pages. Much more information can be found in the *Controller Chapter*.

Optional Step 3: Create the Template

Templates allows you to move all of the presentation (e.g. HTML code) into a separate file and reuse different portions of the page layout. Instead of writing the HTML inside the controller, render a template instead:

Listing 4-11

Listing 4-12

```
1  // src/Acme/HelloBundle/Controller/HelloController.php
2  namespace Acme\HelloBundle\Controller;
3
4  use Symfony\Bundle\FrameworkBundle\Controller\Controller;
5
6  class HelloController extends Controller
7  {
8      public function indexAction($name)
9      {
10         return $this->render('AcmeHelloBundle:Hello:index.html.twig', array('name' =>
11 $name));
12
13         // render a PHP template instead
14         // return $this->render('AcmeHelloBundle:Hello:index.html.php', array('name' =>
15 $name));
       }
   }
```

In order to use the `render()` method, your controller must extend the `Symfony\Bundle\FrameworkBundle\Controller\Controller` class (API docs: *`Controller`*[3]), which adds shortcuts for tasks that are common inside controllers. This is done in the above example by adding the `use` statement on line 4 and then extending `Controller` on line 6.

The `render()` method creates a `Response` object filled with the content of the given, rendered template. Like any other controller, you will ultimately return that `Response` object.

3. http://api.symfony.com/2.0/Symfony/Bundle/FrameworkBundle/Controller/Controller.html

Notice that there are two different examples for rendering the template. By default, Symfony2 supports two different templating languages: classic PHP templates and the succinct but powerful *Twig*[4] templates. Don't be alarmed - you're free to choose either or even both in the same project.

The controller renders the `AcmeHelloBundle:Hello:index.html.twig` template, which uses the following naming convention:

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

This is the *logical* name of the template, which is mapped to a physical location using the following convention.

> **/path/to/BundleName**/Resources/views/**ControllerName**/**TemplateName**

In this case, `AcmeHelloBundle` is the bundle name, `Hello` is the controller, and `index.html.twig` the template:

Listing 4-13 Listing 4-14

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

Let's step through the Twig template line-by-line:

- *line 2*: The `extends` token defines a parent template. The template explicitly defines a layout file inside of which it will be placed.
- *line 4*: The `block` token says that everything inside should be placed inside a block called `body`. As you'll see, it's the responsibility of the parent template (`base.html.twig`) to ultimately render the block called `body`.

The parent template, `::base.html.twig`, is missing both the **BundleName** and **ControllerName** portions of its name (hence the double colon (`::`) at the beginning). This means that the template lives outside of the bundles and in the `app` directory:

Listing 4-15

```
{# 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>
```

The base template file defines the HTML layout and renders the `body` block that you defined in the `index.html.twig` template. It also renders a `title` block, which you could choose to define in the `index.html.twig` template. Since you did not define the `title` block in the child template, it defaults to "Welcome!".

4. `http://twig.sensiolabs.org`

Templates are a powerful way to render and organize the content for your page. A template can render anything, from HTML markup, to CSS code, or anything else that the controller may need to return.

In the lifecycle of handling a request, the templating engine is simply an optional tool. Recall that the goal of each controller is to return a `Response` object. Templates are a powerful, but optional, tool for creating the content for that `Response` object.

The Directory Structure

After just a few short sections, you already understand the philosophy behind creating and rendering pages in Symfony2. You've also already begun to see how Symfony2 projects are structured and organized. By the end of this section, you'll know where to find and put different types of files and why.

Though entirely flexible, by default, each Symfony *application* has the same basic and recommended directory structure:

- `app/`: This directory contains the application configuration;
- `src/`: All the project PHP code is stored under this directory;
- `vendor/`: Any vendor libraries are placed here by convention;
- `web/`: This is the web root directory and contains any publicly accessible files;

The Web Directory

The web root directory is the home of all public and static files including images, stylesheets, and JavaScript files. It is also where each *front controller* lives:

Listing 4-16

```
// 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();
```

The front controller file (`app.php` in this example) is the actual PHP file that's executed when using a Symfony2 application and its job is to use a Kernel class, `AppKernel`, to bootstrap the application.

Having a front controller means different and more flexible URLs than are used in a typical flat PHP application. When using a front controller, URLs are formatted in the following way:

Listing 4-17

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

The front controller, `app.php`, is executed and the "internal:" URL `/hello/Ryan` is routed internally using the routing configuration. By using Apache `mod_rewrite` rules, you can force the `app.php` file to be executed without needing to specify it in the URL:

Listing 4-18

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

Though front controllers are essential in handling every request, you'll rarely need to modify or even think about them. We'll mention them again briefly in the Environments section.

The Application (app) Directory

As you saw in the front controller, the `AppKernel` class is the main entry point of the application and is responsible for all configuration. As such, it is stored in the `app/` directory.

This class must implement two methods that define everything that Symfony needs to know about your application. You don't even need to worry about these methods when starting - Symfony fills them in for you with sensible defaults.

- `registerBundles()`: Returns an array of all bundles needed to run the application (see *The Bundle System*);
- `registerContainerConfiguration()`: Loads the main application configuration resource file (see the Application Configuration section).

In day-to-day development, you'll mostly use the `app/` directory to modify configuration and routing files in the `app/config/` directory (see Application Configuration). It also contains the application cache directory (`app/cache`), a log directory (`app/logs`) and a directory for application-level resource files, such as templates (`app/Resources`). You'll learn more about each of these directories in later chapters.

Autoloading

When Symfony is loading, a special file - `app/autoload.php` - is included. This file is responsible for configuring the autoloader, which will autoload your application files from the `src/` directory and third-party libraries from the `vendor/` directory.

Because of the autoloader, you never need to worry about using `include` or `require` statements. Instead, Symfony2 uses the namespace of a class to determine its location and automatically includes the file on your behalf the instant you need a class.

The autoloader is already configured to look in the `src/` directory for any of your PHP classes. For autoloading to work, the class name and path to the file have to follow the same pattern:

Listing 4-19

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

Typically, the only time you'll need to worry about the `app/autoload.php` file is when you're including a new third-party library in the `vendor/` directory. For more information on autoloading, see *How to autoload Classes*.

The Source (src) Directory

Put simply, the `src/` directory contains all of the actual code (PHP code, templates, configuration files, stylesheets, etc) that drives *your* application. When developing, the vast majority of your work will be done inside one or more bundles that you create in this directory.

But what exactly is a *bundle*?

The Bundle System

A bundle is similar to a plugin in other software, but even better. The key difference is that *everything* is a bundle in Symfony2, including both the core framework functionality and the code written for your application. Bundles are first-class citizens in Symfony2. This gives you the flexibility to use pre-built

features packaged in *third-party bundles*[5] or to distribute your own bundles. It makes it easy to pick and choose which features to enable in your application and to optimize them the way you want.

While you'll learn the basics here, an entire cookbook entry is devoted to the organization and best practices of *bundles*.

A bundle is simply a structured set of files within a directory that implement a single feature. You might create a `BlogBundle`, a `ForumBundle` or a bundle for user management (many of these exist already as open source bundles). Each directory contains everything related to that feature, including PHP files, templates, stylesheets, JavaScripts, tests and anything else. Every aspect of a feature exists in a bundle and every feature lives in a bundle.

An application is made up of bundles as defined in the `registerBundles()` method of the `AppKernel` class:

Listing 4-20

```
// 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;
}
```

With the `registerBundles()` method, you have total control over which bundles are used by your application (including the core Symfony bundles).

A bundle can live *anywhere* as long as it can be autoloaded (via the autoloader configured at `app/autoload.php`).

Creating a Bundle

The Symfony Standard Edition comes with a handy task that creates a fully-functional bundle for you. Of course, creating a bundle by hand is pretty easy as well.

To show you how simple the bundle system is, create a new bundle called `AcmeTestBundle` and enable it.

5. `http://symfony2bundles.org/`

The `Acme` portion is just a dummy name that should be replaced by some "vendor" name that represents you or your organization (e.g. `ABCTestBundle` for some company named `ABC`).

Start by creating a `src/Acme/TestBundle/` directory and adding a new file called `AcmeTestBundle.php`:

Listing 4-21
```php
// src/Acme/TestBundle/AcmeTestBundle.php
namespace Acme\TestBundle;

use Symfony\Component\HttpKernel\Bundle\Bundle;

class AcmeTestBundle extends Bundle
{
}
```

The name `AcmeTestBundle` follows the standard *Bundle naming conventions*. You could also choose to shorten the name of the bundle to simply `TestBundle` by naming this class `TestBundle` (and naming the file `TestBundle.php`).

This empty class is the only piece you need to create the new bundle. Though commonly empty, this class is powerful and can be used to customize the behavior of the bundle.

Now that you've created the bundle, enable it via the `AppKernel` class:

Listing 4-22
```php
// app/AppKernel.php
public function registerBundles()
{
    $bundles = array(
        // ...

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

    return $bundles;
}
```

And while it doesn't do anything yet, `AcmeTestBundle` is now ready to be used.

And as easy as this is, Symfony also provides a command-line interface for generating a basic bundle skeleton:

Listing 4-23
```
php app/console generate:bundle --namespace=Acme/TestBundle
```

The bundle skeleton generates with a basic controller, template and routing resource that can be customized. You'll learn more about Symfony2's command-line tools later.

Whenever creating a new bundle or using a third-party bundle, always make sure the bundle has been enabled in `registerBundles()`. When using the `generate:bundle` command, this is done for you.

Bundle Directory Structure

The directory structure of a bundle is simple and flexible. By default, the bundle system follows a set of conventions that help to keep code consistent between all Symfony2 bundles. Take a look at `AcmeHelloBundle`, as it contains some of the most common elements of a bundle:

- `Controller/` contains the controllers of the bundle (e.g. `HelloController.php`);
- `Resources/config/` houses configuration, including routing configuration (e.g. `routing.yml`);
- `Resources/views/` holds templates organized by controller name (e.g. `Hello/index.html.twig`);
- `Resources/public/` contains web assets (images, stylesheets, etc) and is copied or symbolically linked into the project `web/` directory via the `assets:install` console command;
- `Tests/` holds all tests for the bundle.

A bundle can be as small or large as the feature it implements. It contains only the files you need and nothing else.

As you move through the book, you'll learn how to persist objects to a database, create and validate forms, create translations for your application, write tests and much more. Each of these has their own place and role within the bundle.

Application Configuration

An application consists of a collection of bundles representing all of the features and capabilities of your application. Each bundle can be customized via configuration files written in YAML, XML or PHP. By default, the main configuration file lives in the `app/config/` directory and is called either `config.yml`, `config.xml` or `config.php` depending on which format you prefer:

Listing 4-24

```yaml
# app/config/config.yml
imports:
    - { resource: parameters.ini }
    - { 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%"

# ...
```

You'll learn exactly how to load each file/format in the next section Environments.

Each top-level entry like `framework` or `twig` defines the configuration for a particular bundle. For example, the `framework` key defines the configuration for the core Symfony `FrameworkBundle` and includes configuration for the routing, templating, and other core systems.

For now, don't worry about the specific configuration options in each section. The configuration file ships with sensible defaults. As you read more and explore each part of Symfony2, you'll learn about the specific configuration options of each feature.

Configuration Formats

Throughout the chapters, all configuration examples will be shown in all three formats (YAML, XML and PHP). Each has its own advantages and disadvantages. The choice of which to use is up to you:

- *YAML*: Simple, clean and readable;
- *XML*: More powerful than YAML at times and supports IDE autocompletion;
- *PHP*: Very powerful but less readable than standard configuration formats.

Environments

An application can run in various environments. The different environments share the same PHP code (apart from the front controller), but use different configuration. For instance, a `dev` environment will log warnings and errors, while a `prod` environment will only log errors. Some files are rebuilt on each request in the `dev` environment (for the developer's convenience), but cached in the `prod` environment. All environments live together on the same machine and execute the same application.

A Symfony2 project generally begins with three environments (`dev`, `test` and `prod`), though creating new environments is easy. You can view your application in different environments simply by changing the front controller in your browser. To see the application in the `dev` environment, access the application via the development front controller:

Listing 4-25
```
http://localhost/app_dev.php/hello/Ryan
```

If you'd like to see how your application will behave in the production environment, call the `prod` front controller instead:

Listing 4-26
```
http://localhost/app.php/hello/Ryan
```

Since the `prod` environment is optimized for speed; the configuration, routing and Twig templates are compiled into flat PHP classes and cached. When viewing changes in the `prod` environment, you'll need to clear these cached files and allow them to rebuild:

Listing 4-27
```
php app/console cache:clear --env=prod --no-debug
```

If you open the `web/app.php` file, you'll find that it's configured explicitly to use the `prod` environment:

Listing 4-28
```
$kernel = new AppKernel('prod', false);
```

You can create a new front controller for a new environment by copying this file and changing `prod` to some other value.

The `test` environment is used when running automated tests and cannot be accessed directly through the browser. See the *testing chapter* for more details.

Environment Configuration

The `AppKernel` class is responsible for actually loading the configuration file of your choice:

Listing 4-29

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

You already know that the `.yml` extension can be changed to `.xml` or `.php` if you prefer to use either XML or PHP to write your configuration. Notice also that each environment loads its own configuration file. Consider the configuration file for the `dev` environment.

Listing 4-30

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

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

# ...
```

The `imports` key is similar to a PHP `include` statement and guarantees that the main configuration file (`config.yml`) is loaded first. The rest of the file tweaks the default configuration for increased logging and other settings conducive to a development environment.

Both the `prod` and `test` environments follow the same model: each environment imports the base configuration file and then modifies its configuration values to fit the needs of the specific environment. This is just a convention, but one that allows you to reuse most of your configuration and customize just pieces of it between environments.

Summary

Congratulations! You've now seen every fundamental aspect of Symfony2 and have hopefully discovered how easy and flexible it can be. And while there are *a lot* of features still to come, be sure to keep the following basic points in mind:

- creating a page is a three-step process involving a **route**, a **controller** and (optionally) a **template**.
- each project contains just a few main directories: `web/` (web assets and the front controllers), `app/` (configuration), `src/` (your bundles), and `vendor/` (third-party code) (there's also a `bin/` directory that's used to help updated vendor libraries);
- each feature in Symfony2 (including the Symfony2 framework core) is organized into a *bundle*, which is a structured set of files for that feature;
- the **configuration** for each bundle lives in the `app/config` directory and can be specified in YAML, XML or PHP;
- each **environment** is accessible via a different front controller (e.g. `app.php` and `app_dev.php`) and loads a different configuration file.

From here, each chapter will introduce you to more and more powerful tools and advanced concepts. The more you know about Symfony2, the more you'll appreciate the flexibility of its architecture and the power it gives you to rapidly develop applications.

Chapter 5
Controller

A controller is a PHP function you create that takes information from the HTTP request and constructs and returns an HTTP response (as a Symfony2 `Response` object). The response could be an HTML page, an XML document, a serialized JSON array, an image, a redirect, a 404 error or anything else you can dream up. The controller contains whatever arbitrary logic *your application* needs to render the content of a page.

To see how simple this is, let's look at a Symfony2 controller in action. The following controller would render a page that simply prints `Hello world!`:

Listing 5-1

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

public function helloAction()
{
    return new Response('Hello world!');
}
```

The goal of a controller is always the same: create and return a `Response` object. Along the way, it might read information from the request, load a database resource, send an email, or set information on the user's session. But in all cases, the controller will eventually return the `Response` object that will be delivered back to the client.

There's no magic and no other requirements to worry about! Here are a few common examples:

- *Controller A* prepares a `Response` object representing the content for the homepage of the site.
- *Controller B* reads the `slug` parameter from the request to load a blog entry from the database and create a `Response` object displaying that blog. If the `slug` can't be found in the database, it creates and returns a `Response` object with a 404 status code.
- *Controller C* handles the form submission of a contact form. It reads the form information from the request, saves the contact information to the database and emails the contact information to the webmaster. Finally, it creates a `Response` object that redirects the client's browser to the contact form "thank you" page.

Requests, Controller, Response Lifecycle

Every request handled by a Symfony2 project goes through the same simple lifecycle. The framework takes care of the repetitive tasks and ultimately executes a controller, which houses your custom application code:

1. Each request is handled by a single front controller file (e.g. `app.php` or `app_dev.php`) that bootstraps the application;
2. The `Router` reads information from the request (e.g. the URI), finds a route that matches that information, and reads the `_controller` parameter from the route;
3. The controller from the matched route is executed and the code inside the controller creates and returns a `Response` object;
4. The HTTP headers and content of the `Response` object are sent back to the client.

Creating a page is as easy as creating a controller (#3) and making a route that maps a URL to that controller (#2).

Though similarly named, a "front controller" is different from the "controllers" we'll talk about in this chapter. A front controller is a short PHP file that lives in your web directory and through which all requests are directed. A typical application will have a production front controller (e.g. `app.php`) and a development front controller (e.g. `app_dev.php`). You'll likely never need to edit, view or worry about the front controllers in your application.

A Simple Controller

While a controller can be any PHP callable (a function, method on an object, or a `Closure`), in Symfony2, a controller is usually a single method inside a controller object. Controllers are also called *actions*.

Listing 5-2 Listing 5-3

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

Note that the *controller* is the `indexAction` method, which lives inside a *controller class* (`HelloController`). Don't be confused by the naming: a *controller class* is simply a convenient way to group several controllers/actions together. Typically, the controller class will house several controllers/actions (e.g. `updateAction`, `deleteAction`, etc).

This controller is pretty straightforward, but let's walk through it:

- *line 3*: Symfony2 takes advantage of PHP 5.3 namespace functionality to namespace the entire controller class. The `use` keyword imports the `Response` class, which our controller must return.

- *line 6*: The class name is the concatenation of a name for the controller class (i.e. `Hello`) and the word `Controller`. This is a convention that provides consistency to controllers and allows them to be referenced only by the first part of the name (i.e. `Hello`) in the routing configuration.
- *line 8*: Each action in a controller class is suffixed with `Action` and is referenced in the routing configuration by the action's name (`index`). In the next section, you'll create a route that maps a URI to this action. You'll learn how the route's placeholders (`{name}`) become arguments to the action method (`$name`).
- *line 10*: The controller creates and returns a `Response` object.

Mapping a URL to a Controller

The new controller returns a simple HTML page. To actually view this page in your browser, you need to create a route, which maps a specific URL pattern to the controller:

Listing 5-4

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

Going to `/hello/ryan` now executes the `HelloController::indexAction()` controller and passes in `ryan` for the `$name` variable. Creating a "page" means simply creating a controller method and associated route.

Notice the syntax used to refer to the controller: `AcmeHelloBundle:Hello:index`. Symfony2 uses a flexible string notation to refer to different controllers. This is the most common syntax and tells Symfony2 to look for a controller class called `HelloController` inside a bundle named `AcmeHelloBundle`. The method `indexAction()` is then executed.

For more details on the string format used to reference different controllers, see *Controller Naming Pattern*.

This example places the routing configuration directly in the `app/config/` directory. A better way to organize your routes is to place each route in the bundle it belongs to. For more information on this, see *Including External Routing Resources*.

You can learn much more about the routing system in the *Routing chapter*.

Route Parameters as Controller Arguments

You already know that the `_controller` parameter `AcmeHelloBundle:Hello:index` refers to a `HelloController::indexAction()` method that lives inside the `AcmeHelloBundle` bundle. What's more interesting is the arguments that are passed to that method:

Listing 5-5

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

The controller has a single argument, `$name`, which corresponds to the `{name}` parameter from the matched route (`ryan` in our example). In fact, when executing your controller, Symfony2 matches each argument of the controller with a parameter from the matched route. Take the following example:

Listing 5-6
```
# app/config/routing.yml
hello:
    pattern:      /hello/{first_name}/{last_name}
    defaults:     { _controller: AcmeHelloBundle:Hello:index, color: green }
```

The controller for this can take several arguments:

Listing 5-7
```
public function indexAction($first_name, $last_name, $color)
{
    // ...
}
```

Notice that both placeholder variables (`{first_name}`, `{last_name}`) as well as the default `color` variable are available as arguments in the controller. When a route is matched, the placeholder variables are merged with the `defaults` to make one array that's available to your controller.

Mapping route parameters to controller arguments is easy and flexible. Keep the following guidelines in mind while you develop.

- **The order of the controller arguments does not matter**

  Symfony is able to match the parameter names from the route to the variable names in the controller method's signature. In other words, it realizes that the `{last_name}` parameter matches up with the `$last_name` argument. The arguments of the controller could be totally reordered and still work perfectly:

  Listing 5-8
  ```
  public function indexAction($last_name, $color, $first_name)
  {
      // ..
  }
  ```

- **Each required controller argument must match up with a routing parameter**

  The following would throw a `RuntimeException` because there is no `foo` parameter defined in the route:

  Listing 5-9
  ```
  public function indexAction($first_name, $last_name, $color, $foo)
  {
      // ..
  }
  ```

  Making the argument optional, however, is perfectly ok. The following example would not throw an exception:

  Listing 5-10
  ```
  public function indexAction($first_name, $last_name, $color, $foo = 'bar')
  {
      // ..
  }
  ```

- **Not all routing parameters need to be arguments on your controller**

If, for example, the `last_name` weren't important for your controller, you could omit it entirely:

Listing 5-11

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

Every route also has a special `_route` parameter, which is equal to the name of the route that was matched (e.g. `hello`). Though not usually useful, this is equally available as a controller argument.

The `Request` as a Controller Argument

For convenience, you can also have Symfony pass you the `Request` object as an argument to your controller. This is especially convenient when you're working with forms, for example:

Listing 5-12

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

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

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

The Base Controller Class

For convenience, Symfony2 comes with a base `Controller` class that assists with some of the most common controller tasks and gives your controller class access to any resource it might need. By extending this `Controller` class, you can take advantage of several helper methods.

Add the `use` statement atop the `Controller` class and then modify the `HelloController` to extend it:

Listing 5-13

```
// 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>');
    }
}
```

This doesn't actually change anything about how your controller works. In the next section, you'll learn about the helper methods that the base controller class makes available. These methods are just shortcuts to using core Symfony2 functionality that's available to you with or without the use of the base `Controller` class. A great way to see the core functionality in action is to look in the *`Controller`*[1] class itself.

1. http://api.symfony.com/2.0/Symfony/Bundle/FrameworkBundle/Controller/Controller.html

Extending the base class is *optional* in Symfony; it contains useful shortcuts but nothing mandatory. You can also extend `Symfony\Component\DependencyInjection\ContainerAware`. The service container object will then be accessible via the `container` property.

You can also define your *Controllers as Services*.

Common Controller Tasks

Though a controller can do virtually anything, most controllers will perform the same basic tasks over and over again. These tasks, such as redirecting, forwarding, rendering templates and accessing core services, are very easy to manage in Symfony2.

Redirecting

If you want to redirect the user to another page, use the `redirect()` method:

Listing 5-14
```php
public function indexAction()
{
    return $this->redirect($this->generateUrl('homepage'));
}
```

The `generateUrl()` method is just a helper function that generates the URL for a given route. For more information, see the *Routing* chapter.

By default, the `redirect()` method performs a 302 (temporary) redirect. To perform a 301 (permanent) redirect, modify the second argument:

Listing 5-15
```php
public function indexAction()
{
    return $this->redirect($this->generateUrl('homepage'), 301);
}
```

The `redirect()` method is simply a shortcut that creates a `Response` object that specializes in redirecting the user. It's equivalent to:

Listing 5-16
```php
use Symfony\Component\HttpFoundation\RedirectResponse;

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

Forwarding

You can also easily forward to another controller internally with the `forward()` method. Instead of redirecting the user's browser, it makes an internal sub-request, and calls the specified controller. The `forward()` method returns the `Response` object that's returned from that controller:

Listing 5-17
```php
public function indexAction($name)
{
    $response = $this->forward('AcmeHelloBundle:Hello:fancy', array(
        'name'  => $name,
```

```
        'color' => 'green'
    ));

    // further modify the response or return it directly

    return $response;
}
```

Notice that the *forward()* method uses the same string representation of the controller used in the routing configuration. In this case, the target controller class will be `HelloController` inside some `AcmeHelloBundle`. The array passed to the method becomes the arguments on the resulting controller. This same interface is used when embedding controllers into templates (see *Embedding Controllers*). The target controller method should look something like the following:

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

Listing 5-18

And just like when creating a controller for a route, the order of the arguments to `fancyAction` doesn't matter. Symfony2 matches the index key names (e.g. `name`) with the method argument names (e.g. `$name`). If you change the order of the arguments, Symfony2 will still pass the correct value to each variable.

Like other base `Controller` methods, the `forward` method is just a shortcut for core Symfony2 functionality. A forward can be accomplished directly via the `http_kernel` service. A forward returns a `Response` object:

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

Listing 5-19

Rendering Templates

Though not a requirement, most controllers will ultimately render a template that's responsible for generating the HTML (or other format) for the controller. The `renderView()` method renders a template and returns its content. The content from the template can be used to create a `Response` object:

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

return new Response($content);
```

Listing 5-20

This can even be done in just one step with the `render()` method, which returns a `Response` object containing the content from the template:

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

Listing 5-21

In both cases, the `Resources/views/Hello/index.html.twig` template inside the `AcmeHelloBundle` will be rendered.

The Symfony templating engine is explained in great detail in the *Templating* chapter.

The `renderView` method is a shortcut to direct use of the `templating` service. The `templating` service can also be used directly:

Listing 5-22
```
$templating = $this->get('templating');
$content = $templating->render('AcmeHelloBundle:Hello:index.html.twig', array('name' =>
$name));
```

Accessing other Services

When extending the base controller class, you can access any Symfony2 service via the `get()` method. Here are several common services you might need:

Listing 5-23
```
$request = $this->getRequest();

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

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

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

There are countless other services available and you are encouraged to define your own. To list all available services, use the `container:debug` console command:

Listing 5-24
```
php app/console container:debug
```

For more information, see the *Service Container* chapter.

Managing Errors and 404 Pages

When things are not found, you should play well with the HTTP protocol and return a 404 response. To do this, you'll throw a special type of exception. If you're extending the base controller class, do the following:

Listing 5-25
```
public function indexAction()
{
    $product = // retrieve the object from database
    if (!$product) {
        throw $this->createNotFoundException('The product does not exist');
    }

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

The `createNotFoundException()` method creates a special `NotFoundHttpException` object, which ultimately triggers a 404 HTTP response inside Symfony.

Of course, you're free to throw any `Exception` class in your controller - Symfony2 will automatically return a 500 HTTP response code.

Listing 5-26
```
throw new \Exception('Something went wrong!');
```

In every case, a styled error page is shown to the end user and a full debug error page is shown to the developer (when viewing the page in debug mode). Both of these error pages can be customized. For details, read the "*How to customize Error Pages*" cookbook recipe.

Managing the Session

Symfony2 provides a nice session object that you can use to store information about the user (be it a real person using a browser, a bot, or a web service) between requests. By default, Symfony2 stores the attributes in a cookie by using the native PHP sessions.

Storing and retrieving information from the session can be easily achieved from any controller:

Listing 5-27

```
$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');
```

These attributes will remain on the user for the remainder of that user's session.

Flash Messages

You can also store small messages that will be stored on the user's session for exactly one additional request. This is useful when processing a form: you want to redirect and have a special message shown on the *next* request. These types of messages are called "flash" messages.

For example, imagine you're processing a form submit:

Listing 5-28

```
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(...);
}
```

After processing the request, the controller sets a `notice` flash message and then redirects. The name (`notice`) isn't significant - it's just what you're using to identify the type of the message.

In the template of the next action, the following code could be used to render the `notice` message:

Listing 5-29

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

By design, flash messages are meant to live for exactly one request (they're "gone in a flash"). They're designed to be used across redirects exactly as you've done in this example.

The Response Object

The only requirement for a controller is to return a `Response` object. The *`Response`*[2] class is a PHP abstraction around the HTTP response - the text-based message filled with HTTP headers and content that's sent back to the client:

Listing 5-30
```php
// create a simple Response with a 200 status code (the default)
$response = new Response('Hello '.$name, 200);

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

The `headers` property is a *`HeaderBag`*[3] object with several useful methods for reading and mutating the `Response` headers. The header names are normalized so that using `Content-Type` is equivalent to `content-type` or even `content_type`.

The Request Object

Besides the values of the routing placeholders, the controller also has access to the `Request` object when extending the base `Controller` class:

Listing 5-31
```php
$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
```

Like the `Response` object, the request headers are stored in a `HeaderBag` object and are easily accessible.

Final Thoughts

Whenever you create a page, you'll ultimately need to write some code that contains the logic for that page. In Symfony, this is called a controller, and it's a PHP function that can do anything it needs in order to return the final `Response` object that will be returned to the user.

To make life easier, you can choose to extend a base `Controller` class, which contains shortcut methods for many common controller tasks. For example, since you don't want to put HTML code in your controller, you can use the `render()` method to render and return the content from a template.

In other chapters, you'll see how the controller can be used to persist and fetch objects from a database, process form submissions, handle caching and more.

2. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/Response.html
3. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/HeaderBag.html

Learn more from the Cookbook

- *How to customize Error Pages*
- *How to define Controllers as Services*

Chapter 6
Routing

Beautiful URLs are an absolute must for any serious web application. This means leaving behind ugly URLs like `index.php?article_id=57` in favor of something like `/read/intro-to-symfony`.

Having flexibility is even more important. What if you need to change the URL of a page from `/blog` to `/news`? How many links should you need to hunt down and update to make the change? If you're using Symfony's router, the change is simple.

The Symfony2 router lets you define creative URLs that you map to different areas of your application. By the end of this chapter, you'll be able to:

- Create complex routes that map to controllers
- Generate URLs inside templates and controllers
- Load routing resources from bundles (or anywhere else)
- Debug your routes

Routing in Action

A *route* is a map from a URL pattern to a controller. For example, suppose you want to match any URL like `/blog/my-post` or `/blog/all-about-symfony` and send it to a controller that can look up and render that blog entry. The route is simple:

Listing 6-1
```yaml
# app/config/routing.yml
blog_show:
    pattern:   /blog/{slug}
    defaults:  { _controller: AcmeBlogBundle:Blog:show }
```

The pattern defined by the `blog_show` route acts like `/blog/*` where the wildcard is given the name `slug`. For the URL `/blog/my-blog-post`, the `slug` variable gets a value of `my-blog-post`, which is available for you to use in your controller (keep reading).

The `_controller` parameter is a special key that tells Symfony which controller should be executed when a URL matches this route. The `_controller` string is called the *logical name*. It follows a pattern that points to a specific PHP class and method:

Listing 6-2

```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 = // use the $slug varible to query the database

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

Congratulations! You've just created your first route and connected it to a controller. Now, when you visit `/blog/my-post`, the `showAction` controller will be executed and the `$slug` variable will be equal to `my-post`.

This is the goal of the Symfony2 router: to map the URL of a request to a controller. Along the way, you'll learn all sorts of tricks that make mapping even the most complex URLs easy.

Routing: Under the Hood

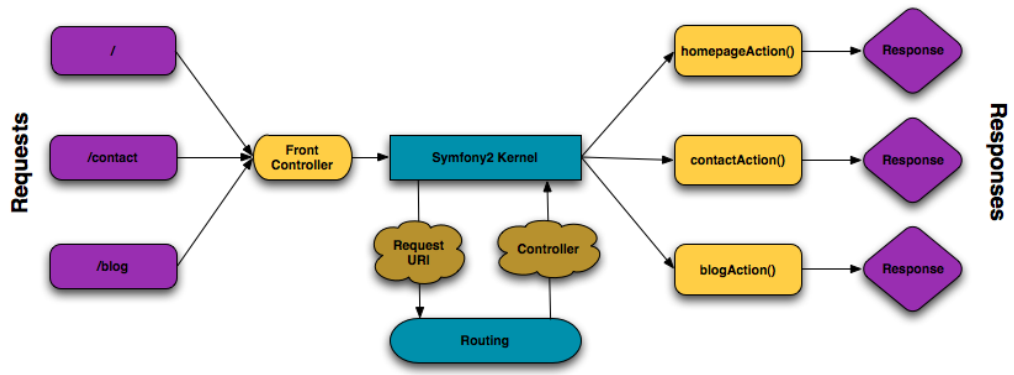
When a request is made to your application, it contains an address to the exact "resource" that the client is requesting. This address is called the URL, (or URI), and could be `/contact`, `/blog/read-me`, or anything else. Take the following HTTP request for example:

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

Listing 6-3

The goal of the Symfony2 routing system is to parse this URL and determine which controller should be executed. The whole process looks like this:

1. The request is handled by the Symfony2 front controller (e.g. `app.php`);
2. The Symfony2 core (i.e. Kernel) asks the router to inspect the request;
3. The router matches the incoming URL to a specific route and returns information about the route, including the controller that should be executed;
4. The Symfony2 Kernel executes the controller, which ultimately returns a `Response` object.



The routing layer is a tool that translates the incoming URL into a specific controller to execute.

Creating Routes

Symfony loads all the routes for your application from a single routing configuration file. The file is usually `app/config/routing.yml`, but can be configured to be anything (including an XML or PHP file) via the application configuration file:

Listing 6-4
```yaml
# app/config/config.yml
framework:
    # ...
    router:        { resource: "%kernel.root_dir%/config/routing.yml" }
```

Even though all routes are loaded from a single file, it's common practice to include additional routing resources from inside the file. See the *Including External Routing Resources* section for more information.

Basic Route Configuration

Defining a route is easy, and a typical application will have lots of routes. A basic route consists of just two parts: the `pattern` to match and a `defaults` array:

Listing 6-5
```yaml
_welcome:
    pattern:   /
    defaults:  { _controller: AcmeDemoBundle:Main:homepage }
```

This route matches the homepage (`/`) and maps it to the `AcmeDemoBundle:Main:homepage` controller. The `_controller` string is translated by Symfony2 into an actual PHP function and executed. That process will be explained shortly in the *Controller Naming Pattern* section.

Routing with Placeholders

Of course the routing system supports much more interesting routes. Many routes will contain one or more named "wildcard" placeholders:

Listing 6-6
```yaml
blog_show:
    pattern:   /blog/{slug}
    defaults:  { _controller: AcmeBlogBundle:Blog:show }
```

The pattern will match anything that looks like `/blog/*`. Even better, the value matching the `{slug}` placeholder will be available inside your controller. In other words, if the URL is `/blog/hello-world`, a `$slug` variable, with a value of `hello-world`, will be available in the controller. This can be used, for example, to load the blog post matching that string.

The pattern will *not*, however, match simply `/blog`. That's because, by default, all placeholders are required. This can be changed by adding a placeholder value to the `defaults` array.

Required and Optional Placeholders

To make things more exciting, add a new route that displays a list of all the available blog posts for this imaginary blog application:

Listing 6-7
```yaml
blog:
    pattern:   /blog
    defaults:  { _controller: AcmeBlogBundle:Blog:index }
```

So far, this route is as simple as possible - it contains no placeholders and will only match the exact URL `/blog`. But what if you need this route to support pagination, where `/blog/2` displays the second page of blog entries? Update the route to have a new `{page}` placeholder:

Listing 6-8

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

Like the `{slug}` placeholder before, the value matching `{page}` will be available inside your controller. Its value can be used to determine which set of blog posts to display for the given page.

But hold on! Since placeholders are required by default, this route will no longer match on simply `/blog`. Instead, to see page 1 of the blog, you'd need to use the URL `/blog/1`! Since that's no way for a rich web app to behave, modify the route to make the `{page}` parameter optional. This is done by including it in the `defaults` collection:

Listing 6-9

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

By adding `page` to the `defaults` key, the `{page}` placeholder is no longer required. The URL `/blog` will match this route and the value of the `page` parameter will be set to `1`. The URL `/blog/2` will also match, giving the `page` parameter a value of `2`. Perfect.

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

Adding Requirements

Take a quick look at the routes that have been created so far:

Listing 6-10

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

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

Can you spot the problem? Notice that both routes have patterns that match URL's that look like `/blog/*`. The Symfony router will always choose the **first** matching route it finds. In other words, the `blog_show` route will *never* be matched. Instead, a URL like `/blog/my-blog-post` will match the first route (`blog`) and return a nonsense value of `my-blog-post` to the `{page}` parameter.

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

The answer to the problem is to add route *requirements*. The routes in this example would work perfectly if the `/blog/{page}` pattern *only* matched URLs where the `{page}` portion is an integer. Fortunately, regular expression requirements can easily be added for each parameter. For example:

Listing 6-11

```
blog:
    pattern:   /blog/{page}
```

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

The `\d+` requirement is a regular expression that says that the value of the `{page}` parameter must be a digit (i.e. a number). The `blog` route will still match on a URL like `/blog/2` (because 2 is a number), but it will no longer match a URL like `/blog/my-blog-post` (because `my-blog-post` is *not* a number).

As a result, a URL like `/blog/my-blog-post` will now properly match the `blog_show` route.

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

Earlier Routes always Win

What this all means is that the order of the routes is very important. If the `blog_show` route were placed above the `blog` route, the URL `/blog/2` would match `blog_show` instead of `blog` since the `{slug}` parameter of `blog_show` has no requirements. By using proper ordering and clever requirements, you can accomplish just about anything.

Since the parameter requirements are regular expressions, the complexity and flexibility of each requirement is entirely up to you. Suppose the homepage of your application is available in two different languages, based on the URL:

Listing 6-12

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

For incoming requests, the `{culture}` portion of the URL is matched against the regular expression `(en|fr)`.

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

Adding HTTP Method Requirements

In addition to the URL, you can also match on the *method* of the incoming request (i.e. GET, HEAD, POST, PUT, DELETE). Suppose you have a contact form with two controllers - one for displaying the form (on a GET request) and one for processing the form when it's submitted (on a POST request). This can be accomplished with the following route configuration:

Listing 6-13

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

contact_process:
```

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

Despite the fact that these two routes have identical patterns (`/contact`), the first route will match only GET requests and the second route will match only POST requests. This means that you can display the form and submit the form via the same URL, while using distinct controllers for the two actions.

If no `_method` requirement is specified, the route will match on *all* methods.

Like the other requirements, the `_method` requirement is parsed as a regular expression. To match `GET` *or* `POST` requests, you can use `GET|POST`.

Advanced Routing Example

At this point, you have everything you need to create a powerful routing structure in Symfony. The following is an example of just how flexible the routing system can be:

Listing 6-14

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

As you've seen, this route will only match if the `{culture}` portion of the URL is either `en` or `fr` and if the `{year}` is a number. This route also shows how you can use a period between placeholders instead of a slash. URLs matching this route might look like:

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

The Special `_format` Routing Parameter

This example also highlights the special `_format` routing parameter. When using this parameter, the matched value becomes the "request format" of the `Request` object. Ultimately, the request format is used for such things such as setting the `Content-Type` of the response (e.g. a `json` request format translates into a `Content-Type` of `application/json`). It can also be used in the controller to render a different template for each value of `_format`. The `_format` parameter is a very powerful way to render the same content in different formats.

Special Routing Parameters

As you've seen, each routing parameter or default value is eventually available as an argument in the controller method. Additionally, there are three parameters that are special: each adds a unique piece of functionality inside your application:

- `_controller`: As you've seen, this parameter is used to determine which controller is executed when the route is matched;
- `_format`: Used to set the request format (*read more*);
- `_locale`: Used to set the locale on the session (*read more*);

Controller Naming Pattern

Every route must have a `_controller` parameter, which dictates which controller should be executed when that route is matched. This parameter uses a simple string pattern called the *logical controller name*, which Symfony maps to a specific PHP method and class. The pattern has three parts, each separated by a colon:

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

For example, a `_controller` value of `AcmeBlogBundle:Blog:show` means:

Bundle	Controller Class	Method Name
AcmeBlogBundle	BlogController	showAction

The controller might look like this:

Listing 6-15

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

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

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

Notice that Symfony adds the string `Controller` to the class name (`Blog` => `BlogController`) and `Action` to the method name (`show` => `showAction`).

You could also refer to this controller using its fully-qualified class name and method: `Acme\BlogBundle\Controller\BlogController::showAction`. But if you follow some simple conventions, the logical name is more concise and allows more flexibility.

In addition to using the logical name or the fully-qualified class name, Symfony supports a third way of referring to a controller. This method uses just one colon separator (e.g. `service_name:indexAction`) and refers to the controller as a service (see *How to define Controllers as Services*).

Route Parameters and Controller Arguments

The route parameters (e.g. `{slug}`) are especially important because each is made available as an argument to the controller method:

Listing 6-16

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

In reality, the entire `defaults` collection is merged with the parameter values to form a single array. Each key of that array is available as an argument on the controller.

In other words, for each argument of your controller method, Symfony looks for a route parameter of that name and assigns its value to that argument. In the advanced example above, any combination (in any order) of the following variables could be used as arguments to the `showAction()` method:

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

Since the placeholders and `defaults` collection are merged together, even the `$_controller` variable is available. For a more detailed discussion, see *Route Parameters as Controller Arguments*.

You can also use a special `$_route` variable, which is set to the name of the route that was matched.

Including External Routing Resources

All routes are loaded via a single configuration file - usually `app/config/routing.yml` (see Creating Routes above). Commonly, however, you'll want to load routes from other places, like a routing file that lives inside a bundle. This can be done by "importing" that file:

Listing 6-17

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

When importing resources from YAML, the key (e.g. `acme_hello`) is meaningless. Just be sure that it's unique so no other lines override it.

The `resource` key loads the given routing resource. In this example the resource is the full path to a file, where the `@AcmeHelloBundle` shortcut syntax resolves to the path of that bundle. The imported file might look like this:

Listing 6-18

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

The routes from this file are parsed and loaded in the same way as the main routing file.

Prefixing Imported Routes

You can also choose to provide a "prefix" for the imported routes. For example, suppose you want the `acme_hello` route to have a final pattern of `/admin/hello/{name}` instead of simply `/hello/{name}`:

Listing 6-19

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

The string `/admin` will now be prepended to the pattern of each route loaded from the new routing resource.

Visualizing & Debugging Routes

While adding and customizing routes, it's helpful to be able to visualize and get detailed information about your routes. A great way to see every route in your application is via the `router:debug` console command. Execute the command by running the following from the root of your project.

Listing 6-20
```
php app/console router:debug
```

The command will print a helpful list of *all* the configured routes in your application:

Listing 6-21
```
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}
```

You can also get very specific information on a single route by including the route name after the command:

Listing 6-22
```
php app/console router:debug article_show
```

Generating URLs

The routing system should also be used to generate URLs. In reality, routing is a bi-directional system: mapping the URL to a controller+parameters and a route+parameters back to a URL. The *`match()`*[1] and *`generate()`*[2] methods form this bi-directional system. Take the `blog_show` example route from earlier:

Listing 6-23
```
$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
```

To generate a URL, you need to specify the name of the route (e.g. `blog_show`) and any wildcards (e.g. `slug = my-blog-post`) used in the pattern for that route. With this information, any URL can easily be generated:

Listing 6-24
```
class MainController extends Controller
{
    public function showAction($slug)
    {
      // ...

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

In an upcoming section, you'll learn how to generate URLs from inside templates.

---

1. http://api.symfony.com/2.0/Symfony/Component/Routing/Router.html#match()
2. http://api.symfony.com/2.0/Symfony/Component/Routing/Router.html#generate()

If the frontend of your application uses AJAX requests, you might want to be able to generate URLs in JavaScript based on your routing configuration. By using the *FOSJsRoutingBundle*[3], you can do exactly that:

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

Listing 6-25

For more information, see the documentation for that bundle.

Generating Absolute URLs

By default, the router will generate relative URLs (e.g. `/blog`). To generate an absolute URL, simply pass `true` to the third argument of the `generate()` method:

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

Listing 6-26

The host that's used when generating an absolute URL is the host of the current `Request` object. This is detected automatically based on server information supplied by PHP. When generating absolute URLs for scripts run from the command line, you'll need to manually set the desired host on the `Request` object:

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

Listing 6-27

Generating URLs with Query Strings

The `generate` method takes an array of wildcard values to generate the URI. But if you pass extra ones, they will be added to the URI as a query string:

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

Listing 6-28

Generating URLs from a template

The most common place to generate a URL is from within a template when linking between pages in your application. This is done just as before, but using a template helper function:

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

Listing 6-29

Absolute URLs can also be generated.

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

Listing 6-30

Summary

Routing is a system for mapping the URL of incoming requests to the controller function that should be called to process the request. It both allows you to specify beautiful URLs and keeps the functionality of

3. `https://github.com/FriendsOfSymfony/FOSJsRoutingBundle`

your application decoupled from those URLs. Routing is a two-way mechanism, meaning that it should also be used to generate URLs.

Learn more from the Cookbook

- *How to force routes to always use HTTPS or HTTP*

Chapter 7
Creating and using Templates

As you know, the *controller* is responsible for handling each request that comes into a Symfony2 application. In reality, the controller delegates the most of the heavy work to other places so that code can be tested and reused. When a controller needs to generate HTML, CSS or any other content, it hands the work off to the templating engine. In this chapter, you'll learn how to write powerful templates that can be used to return content to the user, populate email bodies, and more. You'll learn shortcuts, clever ways to extend templates and how to reuse template code.

Templates

A template is simply a text file that can generate any text-based format (HTML, XML, CSV, LaTeX ...). The most familiar type of template is a *PHP* template - a text file parsed by PHP that contains a mix of text and PHP code:

Listing 7-1

```
<!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>
```

But Symfony2 packages an even more powerful templating language called *Twig*[1]. Twig allows you to write concise, readable templates that are more friendly to web designers and, in several ways, more powerful than PHP templates:

Listing 7-2
```
<!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 defines two types of special syntax:

- `{{ ... }}`: "Says something": prints a variable or the result of an expression to the template;
- `{% ... %}`: "Does something": a **tag** that controls the logic of the template; it is used to execute statements such as for-loops for example.

There is a third syntax used for creating comments: `{# this is a comment #}`. This syntax can be used across multiple lines like the PHP-equivalent `/* comment */` syntax.

Twig also contains **filters**, which modify content before being rendered. The following makes the `title` variable all uppercase before rendering it:

Listing 7-3
```
{{ title|upper }}
```

Twig comes with a long list of *tags*[2] and *filters*[3] that are available by default. You can even *add your own extensions*[4] to Twig as needed.

Registering a Twig extension is as easy as creating a new service and tagging it with `twig.extension` *tag*.

As you'll see throughout the documentation, Twig also supports functions and new functions can be easily added. For example, the following uses a standard `for` tag and the `cycle` function to print ten div tags, with alternating `odd`, `even` classes:

Listing 7-4
```
{% for i in 0..10 %}
    <div class="{{ cycle(['odd', 'even'], i) }}">
      <!-- some HTML here -->
    </div>
{% endfor %}
```

1. http://twig.sensiolabs.org
2. http://twig.sensiolabs.org/doc/tags/index.html
3. http://twig.sensiolabs.org/doc/filters/index.html
4. http://twig.sensiolabs.org/doc/advanced.html#creating-an-extension

Throughout this chapter, template examples will be shown in both Twig and PHP.

Why Twig?

Twig templates are meant to be simple and won't process PHP tags. This is by design: the Twig template system is meant to express presentation, not program logic. The more you use Twig, the more you'll appreciate and benefit from this distinction. And of course, you'll be loved by web designers everywhere.

Twig can also do things that PHP can't, such as true template inheritance (Twig templates compile down to PHP classes that inherit from each other), whitespace control, sandboxing, and the inclusion of custom functions and filters that only affect templates. Twig contains little features that make writing templates easier and more concise. Take the following example, which combines a loop with a logical `if` statement:

Listing 7-5

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

Twig Template Caching

Twig is fast. Each Twig template is compiled down to a native PHP class that is rendered at runtime. The compiled classes are located in the `app/cache/{environment}/twig` directory (where `{environment}` is the environment, such as `dev` or `prod`) and in some cases can be useful while debugging. See *Environments* for more information on environments.

When `debug` mode is enabled (common in the `dev` environment), a Twig template will be automatically recompiled when changes are made to it. This means that during development you can happily make changes to a Twig template and instantly see the changes without needing to worry about clearing any cache.

When `debug` mode is disabled (common in the `prod` environment), however, you must clear the Twig cache directory so that the Twig templates will regenerate. Remember to do this when deploying your application.

Template Inheritance and Layouts

More often than not, templates in a project share common elements, like the header, footer, sidebar or more. In Symfony2, we like to think about this problem differently: a template can be decorated by another one. This works exactly the same as PHP classes: template inheritance allows you to build a base "layout" template that contains all the common elements of your site defined as **blocks** (think "PHP class with base methods"). A child template can extend the base layout and override any of its blocks (think "PHP subclass that overrides certain methods of its parent class").

First, build a base layout file:

Listing 7-6

```
{# 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>
```

Though the discussion about template inheritance will be in terms of Twig, the philosophy is the same between Twig and PHP templates.

This template defines the base HTML skeleton document of a simple two-column page. In this example, three `{% block %}` areas are defined (`title`, `sidebar` and `body`). Each block may be overridden by a child template or left with its default implementation. This template could also be rendered directly. In that case the `title`, `sidebar` and `body` blocks would simply retain the default values used in this template.

A child template might look like this:

Listing 7-7
```
{# 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 %}
```

The parent template is identified by a special string syntax (`::base.html.twig`) that indicates that the template lives in the `app/Resources/views` directory of the project. This naming convention is explained fully in *Template Naming and Locations*.

The key to template inheritance is the `{% extends %}` tag. This tells the templating engine to first evaluate the base template, which sets up the layout and defines several blocks. The child template is then rendered, at which point the `title` and `body` blocks of the parent are replaced by those from the child. Depending on the value of `blog_entries`, the output might look like this:

Listing 7-8
```
<!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>
```

Notice that since the child template didn't define a `sidebar` block, the value from the parent template is used instead. Content within a `{% block %}` tag in a parent template is always used by default.

You can use as many levels of inheritance as you want. In the next section, a common three-level inheritance model will be explained along with how templates are organized inside a Symfony2 project.

When working with template inheritance, here are some tips to keep in mind:

- If you use `{% extends %}` in a template, it must be the first tag in that template.
- The more `{% block %}` tags you have in your base templates, the better. Remember, child templates don't have to define all parent blocks, so create as many blocks in your base templates as you want and give each a sensible default. The more blocks your base templates have, the more flexible your layout will be.
- If you find yourself duplicating content in a number of templates, it probably means you should move that content to a `{% block %}` in a parent template. In some cases, a better solution may be to move the content to a new template and `include` it (see *Including other Templates*).
- If you need to get the content of a block from the parent template, you can use the `{{ parent() }}` function. This is useful if you want to add to the contents of a parent block instead of completely overriding it:

Listing 7-9

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

Template Naming and Locations

By default, templates can live in two different locations:

- `app/Resources/views/`: The applications `views` directory can contain application-wide base templates (i.e. your application's layouts) as well as templates that override bundle templates (see *Overriding Bundle Templates*);
- `path/to/bundle/Resources/views/`: Each bundle houses its templates in its `Resources/views` directory (and subdirectories). The majority of templates will live inside a bundle.

Symfony2 uses a **bundle**:**controller**:**template** string syntax for templates. This allows for several different types of templates, each which lives in a specific location:

- `AcmeBlogBundle:Blog:index.html.twig`: This syntax is used to specify a template for a specific page. The three parts of the string, each separated by a colon (`:`), mean the following:

    - `AcmeBlogBundle`: (*bundle*) the template lives inside the `AcmeBlogBundle` (e.g. `src/Acme/BlogBundle`);
    - `Blog`: (*controller*) indicates that the template lives inside the `Blog` subdirectory of `Resources/views`;
    - `index.html.twig`: (*template*) the actual name of the file is `index.html.twig`.

  Assuming that the `AcmeBlogBundle` lives at `src/Acme/BlogBundle`, the final path to the layout would be `src/Acme/BlogBundle/Resources/views/Blog/index.html.twig`.
- `AcmeBlogBundle::layout.html.twig`: This syntax refers to a base template that's specific to the `AcmeBlogBundle`. Since the middle, "controller", portion is missing (e.g. `Blog`), the template lives at `Resources/views/layout.html.twig` inside `AcmeBlogBundle`.
- `::base.html.twig`: This syntax refers to an application-wide base template or layout. Notice that the string begins with two colons (`::`), meaning that both the *bundle* and *controller* portions are missing. This means that the template is not located in any bundle, but instead in the root `app/Resources/views/` directory.

In the *Overriding Bundle Templates* section, you'll find out how each template living inside the `AcmeBlogBundle`, for example, can be overridden by placing a template of the same name in the `app/Resources/AcmeBlogBundle/views/` directory. This gives the power to override templates from any vendor bundle.

Hopefully the template naming syntax looks familiar - it's the same naming convention used to refer to *Controller Naming Pattern*.

Template Suffix

The **bundle**:**controller**:**template** format of each template specifies *where* the template file is located. Every template name also has two extensions that specify the *format* and *engine* for that template.

- **AcmeBlogBundle:Blog:index.html.twig** - HTML format, Twig engine
- **AcmeBlogBundle:Blog:index.html.php** - HTML format, PHP engine
- **AcmeBlogBundle:Blog:index.css.twig** - CSS format, Twig engine

By default, any Symfony2 template can be written in either Twig or PHP, and the last part of the extension (e.g. `.twig` or `.php`) specifies which of these two *engines* should be used. The first part of the extension, (e.g. `.html`, `.css`, etc) is the final format that the template will generate. Unlike the engine, which determines how Symfony2 parses the template, this is simply an organizational tactic used in case the same resource needs to be rendered as HTML (`index.html.twig`), XML (`index.xml.twig`), or any other format. For more information, read the *Debugging* section.

The available "engines" can be configured and even new engines added. See *Templating Configuration* for more details.

Tags and Helpers

You already understand the basics of templates, how they're named and how to use template inheritance. The hardest parts are already behind you. In this section, you'll learn about a large group of tools available to help perform the most common template tasks such as including other templates, linking to pages and including images.

Symfony2 comes bundled with several specialized Twig tags and functions that ease the work of the template designer. In PHP, the templating system provides an extensible *helper* system that provides useful features in a template context.

We've already seen a few built-in Twig tags (`{% block %}` & `{% extends %}`) as well as an example of a PHP helper (`$view['slots']`). Let's learn a few more.

Including other Templates

You'll often want to include the same template or code fragment on several different pages. For example, in an application with "news articles", the template code displaying an article might be used on the article detail page, on a page displaying the most popular articles, or in a list of the latest articles.

When you need to reuse a chunk of PHP code, you typically move the code to a new PHP class or function. The same is true for templates. By moving the reused template code into its own template, it can be included from any other template. First, create the template that you'll need to reuse.

Listing 7-10

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

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

Including this template from any other template is simple:

Listing 7-11

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

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

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

The template is included using the `{% include %}` tag. Notice that the template name follows the same typical convention. The `articleDetails.html.twig` template uses an `article` variable. This is passed in by the `list.html.twig` template using the `with` command.

The `{'article': article}` syntax is the standard Twig syntax for hash maps (i.e. an array with named keys). If we needed to pass in multiple elements, it would look like this: `{'foo': foo, 'bar': bar}`.

Embedding Controllers

In some cases, you need to do more than include a simple template. Suppose you have a sidebar in your layout that contains the three most recent articles. Retrieving the three articles may include querying the database or performing other heavy logic that can't be done from within a template.

The solution is to simply embed the result of an entire controller from your template. First, create a controller that renders a certain number of recent articles:

Listing 7-12

```
// 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' => $articles));
    }
}
```

The `recentList` template is perfectly straightforward:

Listing 7-13

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

Notice that we've cheated and hardcoded the article URL in this example (e.g. `/article/*slug*`). This is a bad practice. In the next section, you'll learn how to do this correctly.

To include the controller, you'll need to refer to it using the standard string syntax for controllers (i.e. **bundle**:**controller**:**action**):

Listing 7-14

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

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

Whenever you find that you need a variable or a piece of information that you don't have access to in a template, consider rendering a controller. Controllers are fast to execute and promote good code organization and reuse.

Linking to Pages

Creating links to other pages in your application is one of the most common jobs for a template. Instead of hardcoding URLs in templates, use the `path` Twig function (or the `router` helper in PHP) to generate URLs based on the routing configuration. Later, if you want to modify the URL of a particular page, all you'll need to do is change the routing configuration; the templates will automatically generate the new URL.

First, link to the "_welcome" page, which is accessible via the following routing configuration:

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

Listing 7-15

To link to the page, just use the `path` Twig function and refer to the route:

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

Listing 7-16

As expected, this will generate the URL `/`. Let's see how this works with a more complicated route:

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

Listing 7-17

In this case, you need to specify both the route name (`article_show`) and a value for the `{slug}` parameter. Using this route, let's revisit the `recentList` template from the previous section and link to the articles correctly:

```
{# 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 %}
```

Listing 7-18

You can also generate an absolute URL by using the `url` Twig function:

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

Listing 7-19

The same can be done in PHP templates by passing a third argument to the `generate()` method:

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

Listing 7-20

Linking to Assets

Templates also commonly refer to images, Javascript, stylesheets and other assets. Of course you could hard-code the path to these assets (e.g. `/images/logo.png`), but Symfony2 provides a more dynamic option via the `asset` Twig function:

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

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

Listing 7-21

The `asset` function's main purpose is to make your application more portable. If your application lives at the root of your host (e.g. *http://example.com*[5]), then the rendered paths should be `/images/logo.png`. But if your application lives in a subdirectory (e.g. *http://example.com/my_app*[6]), each asset path should render with the subdirectory (e.g. `/my_app/images/logo.png`). The `asset` function takes care of this by determining how your application is being used and generating the correct paths accordingly.

Additionally, if you use the `asset` function, Symfony can automatically append a query string to your asset, in order to guarantee that updated static assets won't be cached when deployed. For example,

5. http://example.com
6. http://example.com/my_app

`/images/logo.png` might look like `/images/logo.png?v2`. For more information, see the *assets_version* configuration option.

Including Stylesheets and Javascripts in Twig

No site would be complete without including Javascript files and stylesheets. In Symfony, the inclusion of these assets is handled elegantly by taking advantage of Symfony's template inheritance.

This section will teach you the philosophy behind including stylesheet and Javascript assets in Symfony. Symfony also packages another library, called Assetic, which follows this philosophy but allows you to do much more interesting things with those assets. For more information on using Assetic see *How to Use Assetic for Asset Management*.

Start by adding two blocks to your base template that will hold your assets: one called `stylesheets` inside the `head` tag and another called `javascripts` just above the closing `body` tag. These blocks will contain all of the stylesheets and Javascripts that you'll need throughout your site:

Listing 7-22
```twig
{# '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>
```

That's easy enough! But what if you need to include an extra stylesheet or Javascript from a child template? For example, suppose you have a contact page and you need to include a `contact.css` stylesheet *just* on that page. From inside that contact page's template, do the following:

Listing 7-23
```twig
{# 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 %}

{# ... #}
```

In the child template, you simply override the `stylesheets` block and put your new stylesheet tag inside of that block. Of course, since you want to add to the parent block's content (and not actually *replace* it), you should use the `parent()` Twig function to include everything from the `stylesheets` block of the base template.

You can also include assets located in your bundles' `Resources/public` folder. You will need to run the `php app/console assets:install target [--symlink]` command, which moves (or symlinks) files into the correct location. (target is by default "web").

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

Listing 7-24

The end result is a page that includes both the `main.css` and `contact.css` stylesheets.

Global Template Variables

During each request, Symfony2 will set a global template variable `app` in both Twig and PHP template engines by default. The `app` variable is a *`GlobalVariables`*[7] instance which will give you access to some application specific variables automatically:

- `app.security` - The security context.
- `app.user` - The current user object.
- `app.request` - The request object.
- `app.session` - The session object.
- `app.environment` - The current environment (dev, prod, etc).
- `app.debug` - True if in debug mode. False otherwise.

```
<p>Username: {{ app.user.username }}</p>
{% if app.debug %}
    <p>Request method: {{ app.request.method }}</p>
    <p>Application Environment: {{ app.environment }}</p>
{% endif %}
```

Listing 7-25

You can add your own global template variables. See the cookbook example on *Global Variables*.

Configuring and using the `templating` Service

The heart of the template system in Symfony2 is the templating `Engine`. This special object is responsible for rendering templates and returning their content. When you render a template in a controller, for example, you're actually using the templating engine service. For example:

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

Listing 7-26

is equivalent to:

```
$engine = $this->container->get('templating'); $content = $engine->render('AcmeArticleBundle:Article:index.html.twig');
return $response = new Response($content);
```

The templating engine (or "service") is preconfigured to work automatically inside Symfony2. It can, of course, be configured further in the application configuration file:

```
# app/config/config.yml
framework:
```

Listing 7-27

7. http://api.symfony.com/2.0/Symfony/Bundle/FrameworkBundle/Templating/GlobalVariables.html

```
# ...
templating: { engines: ['twig'] }
```

Several configuration options are available and are covered in the *Configuration Appendix*.

The `twig` engine is mandatory to use the webprofiler (as well as many third-party bundles).

Overriding Bundle Templates

The Symfony2 community prides itself on creating and maintaining high quality bundles (see *KnpBundles.com*[8]) for a large number of different features. Once you use a third-party bundle, you'll likely need to override and customize one or more of its templates.

Suppose you've included the imaginary open-source `AcmeBlogBundle` in your project (e.g. in the `src/Acme/BlogBundle` directory). And while you're really happy with everything, you want to override the blog "list" page to customize the markup specifically for your application. By digging into the `Blog` controller of the `AcmeBlogBundle`, you find the following:

Listing 7-28

```
public function indexAction()
{
    $blogs = // some logic to retrieve the blogs

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

When the `AcmeBlogBundle:Blog:index.html.twig` is rendered, Symfony2 actually looks in two different locations for the template:

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

To override the bundle template, just copy the `index.html.twig` template from the bundle to `app/Resources/AcmeBlogBundle/views/Blog/index.html.twig` (the `app/Resources/AcmeBlogBundle` directory won't exist, so you'll need to create it). You're now free to customize the template.

This logic also applies to base bundle templates. Suppose also that each template in `AcmeBlogBundle` inherits from a base template called `AcmeBlogBundle::layout.html.twig`. Just as before, Symfony2 will look in the following two places for the template:

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

Once again, to override the template, just copy it from the bundle to `app/Resources/AcmeBlogBundle/views/layout.html.twig`. You're now free to customize this copy as you see fit.

If you take a step back, you'll see that Symfony2 always starts by looking in the `app/Resources/{BUNDLE_NAME}/views/` directory for a template. If the template doesn't exist there, it continues by checking inside the `Resources/views` directory of the bundle itself. This means that all bundle templates can be overridden by placing them in the correct `app/Resources` subdirectory.

You can also override templates from within a bundle by using bundle inheritance. For more information, see *How to use Bundle Inheritance to Override parts of a Bundle*.

8. `http://knpbundles.com`

Overriding Core Templates

Since the Symfony2 framework itself is just a bundle, core templates can be overridden in the same way. For example, the core `TwigBundle` contains a number of different "exception" and "error" templates that can be overridden by copying each from the `Resources/views/Exception` directory of the `TwigBundle` to, you guessed it, the `app/Resources/TwigBundle/views/Exception` directory.

Three-level Inheritance

One common way to use inheritance is to use a three-level approach. This method works perfectly with the three different types of templates we've just covered:

- Create a `app/Resources/views/base.html.twig` file that contains the main layout for your application (like in the previous example). Internally, this template is called `::base.html.twig`;
- Create a template for each "section" of your site. For example, an `AcmeBlogBundle`, would have a template called `AcmeBlogBundle::layout.html.twig` that contains only blog section-specific elements;

Listing 7-29

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

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

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

- Create individual templates for each page and make each extend the appropriate section template. For example, the "index" page would be called something close to `AcmeBlogBundle:Blog:index.html.twig` and list the actual blog posts.

Listing 7-30

```
{# 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 %}
```

Notice that this template extends the section template -(`AcmeBlogBundle::layout.html.twig`) which in-turn extends the base application layout (`::base.html.twig`). This is the common three-level inheritance model.

When building your application, you may choose to follow this method or simply make each page template extend the base application template directly (e.g. `{% extends '::base.html.twig' %}`). The three-template model is a best-practice method used by vendor bundles so that the base template for a bundle can be easily overridden to properly extend your application's base layout.

Output Escaping

When generating HTML from a template, there is always a risk that a template variable may output unintended HTML or dangerous client-side code. The result is that dynamic content could break the

HTML of the resulting page or allow a malicious user to perform a *Cross Site Scripting*[9] (XSS) attack. Consider this classic example:

Listing 7-31
```
Hello {{ name }}
```

Imagine that the user enters the following code as his/her name:

Listing 7-32
```
<script>alert('hello!')</script>
```

Without any output escaping, the resulting template will cause a JavaScript alert box to pop up:

Listing 7-33
```
Hello <script>alert('hello!')</script>
```

And while this seems harmless, if a user can get this far, that same user should also be able to write JavaScript that performs malicious actions inside the secure area of an unknowing, legitimate user.

The answer to the problem is output escaping. With output escaping on, the same template will render harmlessly, and literally print the `script` tag to the screen:

Listing 7-34
```
Hello &lt;script&gt;alert(&#39;helloe&#39;)&lt;/script&gt;
```

The Twig and PHP templating systems approach the problem in different ways. If you're using Twig, output escaping is on by default and you're protected. In PHP, output escaping is not automatic, meaning you'll need to manually escape where necessary.

Output Escaping in Twig

If you're using Twig templates, then output escaping is on by default. This means that you're protected out-of-the-box from the unintentional consequences of user-submitted code. By default, the output escaping assumes that content is being escaped for HTML output.

In some cases, you'll need to disable output escaping when you're rendering a variable that is trusted and contains markup that should not be escaped. Suppose that administrative users are able to write articles that contain HTML code. By default, Twig will escape the article body. To render it normally, add the `raw` filter: `{{ article.body|raw }}`.

You can also disable output escaping inside a `{% block %}` area or for an entire template. For more information, see *Output Escaping*[10] in the Twig documentation.

Output Escaping in PHP

Output escaping is not automatic when using PHP templates. This means that unless you explicitly choose to escape a variable, you're not protected. To use output escaping, use the special `escape()` view method:

Listing 7-35
```
Hello <?php echo $view->escape($name) ?>
```

By default, the `escape()` method assumes that the variable is being rendered within an HTML context (and thus the variable is escaped to be safe for HTML). The second argument lets you change the context. For example, to output something in a JavaScript string, use the `js` context:

Listing 7-36
```
var myMsg = 'Hello <?php echo $view->escape($name, 'js') ?>';
```

9. http://en.wikipedia.org/wiki/Cross-site_scripting
10. http://twig.sensiolabs.org/doc/api.html#escaper-extension

Debugging

New in version 2.0.9: This feature is available as of Twig `1.5.x`, which was first shipped with Symfony 2.0.9.

When using PHP, you can use `var_dump()` if you need to quickly find the value of a variable passed. This is useful, for example, inside your controller. The same can be achieved when using Twig by using the debug extension. This needs to be enabled in the config:

Listing 7-37

```
# app/config/config.yml
services:
    acme_hello.twig.extension.debug:
        class:        Twig_Extension_Debug
        tags:
            - { name: 'twig.extension' }
```

Template parameters can then be dumped using the `dump` function:

Listing 7-38

```
{# src/Acme/ArticleBundle/Resources/views/Article/recentList.html.twig #}

{{ dump(articles) }}

{% for article in articles %}
    <a href="/article/{{ article.slug }}">
        {{ article.title }}
    </a>
{% endfor %}
```

The variables will only be dumped if Twig's `debug` setting (in `config.yml`) is `true`. By default this means that the variables will be dumped in the `dev` environment but not the `prod` environment.

Template Formats

Templates are a generic way to render content in *any* format. And while in most cases you'll use templates to render HTML content, a template can just as easily generate JavaScript, CSS, XML or any other format you can dream of.

For example, the same "resource" is often rendered in several different formats. To render an article index page in XML, simply include the format in the template name:

- *XML template name*: `AcmeArticleBundle:Article:index.xml.twig`
- *XML template filename*: `index.xml.twig`

In reality, this is nothing more than a naming convention and the template isn't actually rendered differently based on its format.

In many cases, you may want to allow a single controller to render multiple different formats based on the "request format". For that reason, a common pattern is to do the following:

Listing 7-39

```
public function indexAction()
{
    $format = $this->getRequest()->getRequestFormat();

    return $this->render('AcmeBlogBundle:Blog:index.'.$format.'.twig');
}
```

The `getRequestFormat` on the `Request` object defaults to `html`, but can return any other format based on the format requested by the user. The request format is most often managed by the routing, where a route can be configured so that `/contact` sets the request format to `html` while `/contact.xml` sets the format to `xml`. For more information, see the *Advanced Example in the Routing chapter*.

To create links that include the format parameter, include a `_format` key in the parameter hash:

Listing 7-40
```
<a href="{{ path('article_show', {'id': 123, '_format': 'pdf'}) }}">
    PDF Version
</a>
```

Final Thoughts

The templating engine in Symfony is a powerful tool that can be used each time you need to generate presentational content in HTML, XML or any other format. And though templates are a common way to generate content in a controller, their use is not mandatory. The `Response` object returned by a controller can be created with or without the use of a template:

Listing 7-41
```
// 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');
```

Symfony's templating engine is very flexible and two different template renderers are available by default: the traditional *PHP* templates and the sleek and powerful *Twig* templates. Both support a template hierarchy and come packaged with a rich set of helper functions capable of performing the most common tasks.

Overall, the topic of templating should be thought of as a powerful tool that's at your disposal. In some cases, you may not need to render a template, and in Symfony2, that's absolutely fine.

Learn more from the Cookbook

- *How to use PHP instead of Twig for Templates*
- *How to customize Error Pages*
- *How to write a custom Twig Extension*

Chapter 8
Databases and Doctrine

Let's face it, one of the most common and challenging tasks for any application involves persisting and reading information to and from a database. Fortunately, Symfony comes integrated with *Doctrine*[1], a library whose sole goal is to give you powerful tools to make this easy. In this chapter, you'll learn the basic philosophy behind Doctrine and see how easy working with a database can be.

Doctrine is totally decoupled from Symfony and using it is optional. This chapter is all about the Doctrine ORM, which aims to let you map objects to a relational database (such as *MySQL*, *PostgreSQL* or *Microsoft SQL*). If you prefer to use raw database queries, this is easy, and explained in the "*How to use Doctrine's DBAL Layer*" cookbook entry.

You can also persist data to *MongoDB*[2] using Doctrine ODM library. For more information, read the "*DoctrineMongoDBBundle*" documentation.

A Simple Example: A Product

The easiest way to understand how Doctrine works is to see it in action. In this section, you'll configure your database, create a `Product` object, persist it to the database and fetch it back out.

Code along with the example

If you want to follow along with the example in this chapter, create an `AcmeStoreBundle` via:

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

Listing 8-1

1. http://www.doctrine-project.org/
2. http://www.mongodb.org/

Configuring the Database

Before you really begin, you'll need to configure your database connection information. By convention, this information is usually configured in an `app/config/parameters.ini` file:

Listing 8-2
```ini
;app/config/parameters.ini
[parameters]
    database_driver   = pdo_mysql
    database_host     = localhost
    database_name     = test_project
    database_user     = root
    database_password = password
```

Defining the configuration via `parameters.ini` is just a convention. The parameters defined in that file are referenced by the main configuration file when setting up Doctrine:

Listing 8-3
```yaml
doctrine:
    dbal:
        driver:   %database_driver%
        host:     %database_host%
        dbname:   %database_name%
        user:     %database_user%
        password: %database_password%
```

By separating the database information into a separate file, you can easily keep different versions of the file on each server. You can also easily store database configuration (or any sensitive information) outside of your project, like inside your Apache configuration, for example. For more information, see *How to Set External Parameters in the Service Container*.

Now that Doctrine knows about your database, you can have it create the database for you:

Listing 8-4
```
php app/console doctrine:database:create
```

Creating an Entity Class

Suppose you're building an application where products need to be displayed. Without even thinking about Doctrine or databases, you already know that you need a `Product` object to represent those products. Create this class inside the `Entity` directory of your `AcmeStoreBundle`:

Listing 8-5
```php
// src/Acme/StoreBundle/Entity/Product.php
namespace Acme\StoreBundle\Entity;

class Product
{
    protected $name;

    protected $price;

    protected $description;
}
```

The class - often called an "entity", meaning *a basic class that holds data* - is simple and helps fulfill the business requirement of needing products in your application. This class can't be persisted to a database yet - it's just a simple PHP class.

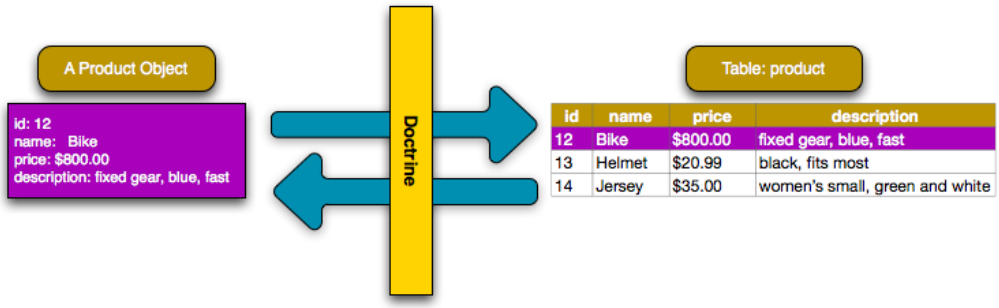
Once you learn the concepts behind Doctrine, you can have Doctrine create this entity class for you:

```
php app/console doctrine:generate:entity --entity="AcmeStoreBundle:Product"
--fields="name:string(255) price:float description:text"
```

Listing 8-6

Add Mapping Information

Doctrine allows you to work with databases in a much more interesting way than just fetching rows of a column-based table into an array. Instead, Doctrine allows you to persist entire *objects* to the database and fetch entire objects out of the database. This works by mapping a PHP class to a database table, and the properties of that PHP class to columns on the table:



For Doctrine to be able to do this, you just have to create "metadata", or configuration that tells Doctrine exactly how the `Product` class and its properties should be *mapped* to the database. This metadata can be specified in a number of different formats including YAML, XML or directly inside the `Product` class via annotations:

A bundle can accept only one metadata definition format. For example, it's not possible to mix YAML metadata definitions with annotated PHP entity class definitions.

```
// 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)
     */
```

Listing 8-7

```
    protected $name;

    /**
     * @ORM\Column(type="decimal", scale=2)
     */
    protected $price;

    /**
     * @ORM\Column(type="text")
     */
    protected $description;
}
```

The table name is optional and if omitted, will be determined automatically based on the name of the entity class.

Doctrine allows you to choose from a wide variety of different field types, each with their own options. For information on the available field types, see the *Doctrine Field Types Reference* section.

> *You can also check out Doctrine's Basic Mapping Documentation*[3] *for all details about mapping information. If you use annotations, you'll need to prepend all annotations with* `ORM\` *(e.g.* `ORM\Column(..)`*), which is not shown in Doctrine's documentation. You'll also need to include the* `use Doctrine\ORM\Mapping as ORM;` *statement, which imports the* `ORM` *annotations prefix.*

Be careful that your class name and properties aren't mapped to a protected SQL keyword (such as `group` or `user`). For example, if your entity class name is `Group`, then, by default, your table name will be `group`, which will cause an SQL error in some engines. See Doctrine's *Reserved SQL keywords documentation*[4] on how to properly escape these names.

When using another library or program (ie. Doxygen) that uses annotations, you should place the `@IgnoreAnnotation` annotation on the class to indicate which annotations Symfony should ignore.

For example, to prevent the `@fn` annotation from throwing an exception, add the following:

Listing 8-8
```
/**
 * @IgnoreAnnotation("fn")
 */
class Product
```

Generating Getters and Setters

Even though Doctrine now knows how to persist a `Product` object to the database, the class itself isn't really useful yet. Since `Product` is just a regular PHP class, you need to create getter and setter methods (e.g. `getName()`, `setName()`) in order to access its properties (since the properties are `protected`). Fortunately, Doctrine can do this for you by running:

Listing 8-9
```
php app/console doctrine:generate:entities Acme/StoreBundle/Entity/Product
```

3. *http://docs.doctrine-project.org/projects/doctrine-orm/en/2.1/reference/basic-mapping.html*
4. http://docs.doctrine-project.org/projects/doctrine-orm/en/2.1/reference/basic-mapping.html#quoting-reserved-words

This command makes sure that all of the getters and setters are generated for the `Product` class. This is a safe command - you can run it over and over again: it only generates getters and setters that don't exist (i.e. it doesn't replace your existing methods).

More about `doctrine:generate:entities`

With the `doctrine:generate:entities` command you can:

- generate getters and setters,
- **generate repository classes configured with the** `@ORM\Entity(repositoryClass="...")` annotation,
- generate the appropriate constructor for 1:n and n:m relations.

The `doctrine:generate:entities` command saves a backup of the original `Product.php` named `Product.php~`. In some cases, the presence of this file can cause a "Cannot redeclare class" error. It can be safely removed.

Note that you don't *need* to use this command. Doctrine doesn't rely on code generation. Like with normal PHP classes, you just need to make sure that your protected/private properties have getter and setter methods. Since this is a common thing to do when using Doctrine, this command was created.

You can also generate all known entities (i.e. any PHP class with Doctrine mapping information) of a bundle or an entire namespace:

```
php app/console doctrine:generate:entities AcmeStoreBundle
php app/console doctrine:generate:entities Acme
```

Listing 8-10

Doctrine doesn't care whether your properties are `protected` or `private`, or whether or not you have a getter or setter function for a property. The getters and setters are generated here only because you'll need them to interact with your PHP object.

Creating the Database Tables/Schema

You now have a usable `Product` class with mapping information so that Doctrine knows exactly how to persist it. Of course, you don't yet have the corresponding `product` table in your database. Fortunately, Doctrine can automatically create all the database tables needed for every known entity in your application. To do this, run:

```
php app/console doctrine:schema:update --force
```

Listing 8-11

Actually, this command is incredibly powerful. It compares what your database *should* look like (based on the mapping information of your entities) with how it *actually* looks, and generates the SQL statements needed to *update* the database to where it should be. In other words, if you add a new property with mapping metadata to `Product` and run this task again, it will generate the "alter table" statement needed to add that new column to the existing `product` table.

An even better way to take advantage of this functionality is via *migrations*, which allow you to generate these SQL statements and store them in migration classes that can be run systematically on your production server in order to track and migrate your database schema safely and reliably.

Your database now has a fully-functional `product` table with columns that match the metadata you've specified.

Persisting Objects to the Database

Now that you have a mapped `Product` entity and corresponding `product` table, you're ready to persist data to the database. From inside a controller, this is pretty easy. Add the following method to the `DefaultController` of the bundle:

Listing 8-12 Listing 8-13

```
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 }
```

If you're following along with this example, you'll need to create a route that points to this action to see it work.

Let's walk through this example:

- **lines 8-11** In this section, you instantiate and work with the `$product` object like any other, normal PHP object;
- **line 13** This line fetches Doctrine's *entity manager* object, which is responsible for handling the process of persisting and fetching objects to and from the database;
- **line 14** The `persist()` method tells Doctrine to "manage" the `$product` object. This does not actually cause a query to be made to the database (yet).
- **line 15** When the `flush()` method is called, Doctrine looks through all of the objects that it's managing to see if they need to be persisted to the database. In this example, the `$product` object has not been persisted yet, so the entity manager executes an `INSERT` query and a row is created in the `product` table.

In fact, since Doctrine is aware of all your managed entities, when you call the `flush()` method, it calculates an overall changeset and executes the most efficient query/queries possible. For example, if you persist a total of 100 `Product` objects and then subsequently call `flush()`, Doctrine will create a *single* prepared statement and re-use it for each insert. This pattern is called *Unit of Work*, and it's used because it's fast and efficient.

When creating or updating objects, the workflow is always the same. In the next section, you'll see how Doctrine is smart enough to automatically issue an `UPDATE` query if the record already exists in the database.

Doctrine provides a library that allows you to programmatically load testing data into your project (i.e. "fixture data"). For information, see *DoctrineFixturesBundle*.

Fetching Objects from the Database

Fetching an object back out of the database is even easier. For example, suppose you've configured a route to display a specific `Product` based on its `id` value:

Listing 8-14

```
public function showAction($id)
{
    $product = $this->getDoctrine()
        ->getRepository('AcmeStoreBundle:Product')
        ->find($id);

    if (!$product) {
        throw $this->createNotFoundException('No product found for id '.$id);
    }

    // do something, like pass the $product object into a template
}
```

When you query for a particular type of object, you always use what's known as its "repository". You can think of a repository as a PHP class whose only job is to help you fetch entities of a certain class. You can access the repository object for an entity class via:

Listing 8-15

```
$repository = $this->getDoctrine()
    ->getRepository('AcmeStoreBundle:Product');
```

The `AcmeStoreBundle:Product` string is a shortcut you can use anywhere in Doctrine instead of the full class name of the entity (i.e. `Acme\StoreBundle\Entity\Product`). As long as your entity lives under the `Entity` namespace of your bundle, this will work.

Once you have your repository, you have access to all sorts of helpful methods:

Listing 8-16

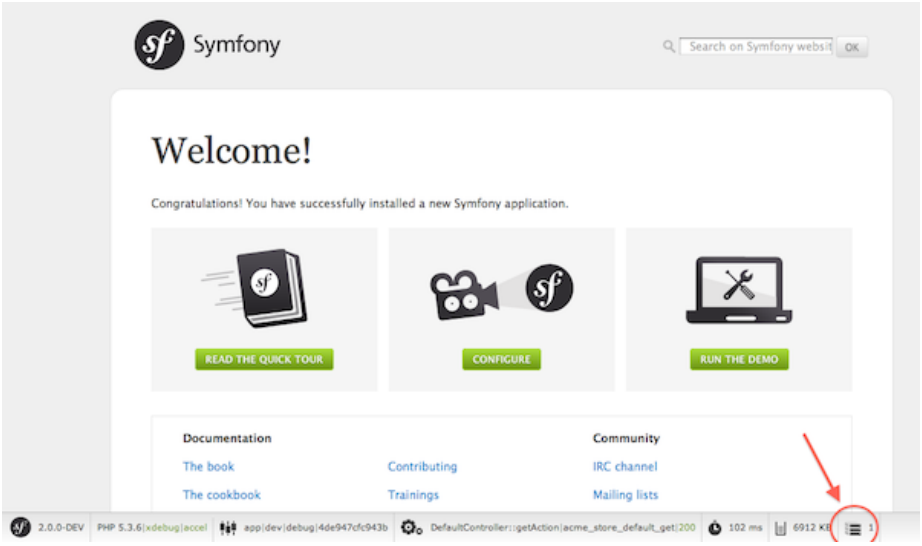
```
// query by the primary key (usually "id")
$product = $repository->find($id);

// dynamic method names to find based on a column value
$product = $repository->findOneById($id);
$product = $repository->findOneByName('foo');

// find *all* products
$products = $repository->findAll();

// find a group of products based on an arbitrary column value
$products = $repository->findByPrice(19.99);
```

Of course, you can also issue complex queries, which you'll learn more about in the *Querying for Objects* section.

You can also take advantage of the useful `findBy` and `findOneBy` methods to easily fetch objects based on multiple conditions:

Listing 8-17
```php
// query for one product matching be name and price
$product = $repository->findOneBy(array('name' => 'foo', 'price' => 19.99));

// query for all products matching the name, ordered by price
$product = $repository->findBy(
    array('name' => 'foo'),
    array('price' => 'ASC')
);
```

When you render any page, you can see how many queries were made in the bottom right corner of the web debug toolbar.



If you click the icon, the profiler will open, showing you the exact queries that were made.

Updating an Object

Once you've fetched an object from Doctrine, updating it is easy. Suppose you have a route that maps a product id to an update action in a controller:

Listing 8-18
```php
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'));
}
```

Updating an object involves just three steps:

1. fetching the object from Doctrine;
2. modifying the object;
3. calling `flush()` on the entity manager

Notice that calling `$em->persist($product)` isn't necessary. Recall that this method simply tells Doctrine to manage or "watch" the `$product` object. In this case, since you fetched the `$product` object from Doctrine, it's already managed.

Deleting an Object

Deleting an object is very similar, but requires a call to the `remove()` method of the entity manager:

```
$em->remove($product);
$em->flush();
```

Listing 8-19

As you might expect, the `remove()` method notifies Doctrine that you'd like to remove the given entity from the database. The actual `DELETE` query, however, isn't actually executed until the `flush()` method is called.

Querying for Objects

You've already seen how the repository object allows you to run basic queries without any work:

```
$repository->find($id);

$repository->findOneByName('Foo');
```

Listing 8-20

Of course, Doctrine also allows you to write more complex queries using the Doctrine Query Language (DQL). DQL is similar to SQL except that you should imagine that you're querying for one or more objects of an entity class (e.g. `Product`) instead of querying for rows on a table (e.g. `product`).

When querying in Doctrine, you have two options: writing pure Doctrine queries or using Doctrine's Query Builder.

Querying for Objects with DQL

Imagine that you want to query for products, but only return products that cost more than `19.99`, ordered from cheapest to most expensive. From inside a controller, do the following:

```
$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();
```

Listing 8-21

If you're comfortable with SQL, then DQL should feel very natural. The biggest difference is that you need to think in terms of "objects" instead of rows in a database. For this reason, you select *from* `AcmeStoreBundle:Product` and then alias it as `p`.

The `getResult()` method returns an array of results. If you're querying for just one object, you can use the `getSingleResult()` method instead:

```
$product = $query->getSingleResult();
```

Listing 8-22

The `getSingleResult()` method throws a `Doctrine\ORM\NoResultException` exception if no results are returned and a `Doctrine\ORM\NonUniqueResultException` if *more* than one result is returned. If you use this method, you may need to wrap it in a try-catch block and ensure that only

one result is returned (if you're querying on something that could feasibly return more than one result):

Listing 8-23
```
$query = $em->createQuery('SELECT ....')
    ->setMaxResults(1);

try {
    $product = $query->getSingleResult();
} catch (\Doctrine\Orm\NoResultException $e) {
    $product = null;
}
// ...
```

The DQL syntax is incredibly powerful, allowing you to easily join between entities (the topic of *relations* will be covered later), group, etc. For more information, see the official Doctrine *Doctrine Query Language*[5] documentation.

Setting Parameters

Take note of the `setParameter()` method. When working with Doctrine, it's always a good idea to set any external values as "placeholders", which was done in the above query:

Listing 8-24
```
... WHERE p.price > :price ...
```

You can then set the value of the `price` placeholder by calling the `setParameter()` method:

Listing 8-25
```
->setParameter('price', '19.99')
```

Using parameters instead of placing values directly in the query string is done to prevent SQL injection attacks and should *always* be done. If you're using multiple parameters, you can set their values at once using the `setParameters()` method:

Listing 8-26
```
->setParameters(array(
    'price' => '19.99',
    'name'  => 'Foo',
))
```

Using Doctrine's Query Builder

Instead of writing the queries directly, you can alternatively use Doctrine's `QueryBuilder` to do the same job using a nice, object-oriented interface. If you use an IDE, you can also take advantage of auto-completion as you type the method names. From inside a controller:

Listing 8-27
```
$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();
```

5. http://docs.doctrine-project.org/projects/doctrine-orm/en/2.1/reference/dql-doctrine-query-language.html

The `QueryBuilder` object contains every method necessary to build your query. By calling the `getQuery()` method, the query builder returns a normal `Query` object, which is the same object you built directly in the previous section.

For more information on Doctrine's Query Builder, consult Doctrine's *Query Builder*[6] documentation.

Custom Repository Classes

In the previous sections, you began constructing and using more complex queries from inside a controller. In order to isolate, test and reuse these queries, it's a good idea to create a custom repository class for your entity and add methods with your query logic there.

To do this, add the name of the repository class to your mapping definition.

Listing 8-28

```
// 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
{
    //...
}
```

Doctrine can generate the repository class for you by running the same command used earlier to generate the missing getter and setter methods:

Listing 8-29

```
php app/console doctrine:generate:entities Acme
```

Next, add a new method - `findAllOrderedByName()` - to the newly generated repository class. This method will query for all of the `Product` entities, ordered alphabetically.

Listing 8-30

```
// 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();
    }
}
```

The entity manager can be accessed via `$this->getEntityManager()` from inside the repository.

You can use this new method just like the default finder methods of the repository:

6. http://docs.doctrine-project.org/projects/doctrine-orm/en/2.1/reference/query-builder.html

Listing 8-31
```
$em = $this->getDoctrine()->getEntityManager();
$products = $em->getRepository('AcmeStoreBundle:Product')
            ->findAllOrderedByName();
```

When using a custom repository class, you still have access to the default finder methods such as `find()` and `findAll()`.

Entity Relationships/Associations

Suppose that the products in your application all belong to exactly one "category". In this case, you'll need a `Category` object and a way to relate a `Product` object to a `Category` object. Start by creating the `Category` entity. Since you know that you'll eventually need to persist the class through Doctrine, you can let Doctrine create the class for you.

Listing 8-32
```
php app/console doctrine:generate:entity --entity="AcmeStoreBundle:Category"
--fields="name:string(255)"
```

This task generates the `Category` entity for you, with an `id` field, a `name` field and the associated getter and setter functions.

Relationship Mapping Metadata

To relate the `Category` and `Product` entities, start by creating a `products` property on the `Category` class:

Listing 8-33
```
// 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();
    }
}
```

First, since a `Category` object will relate to many `Product` objects, a `products` array property is added to hold those `Product` objects. Again, 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.

The code in the `__construct()` method is important because Doctrine requires the `$products` property to be an `ArrayCollection` object. This object looks and acts almost *exactly* like an array, but has some added flexibility. If this makes you uncomfortable, don't worry. Just imagine that it's an `array` and you'll be in good shape.

The targetEntity value in the decorator used above can reference any entity with a valid namespace, not just entities defined in the same class. To relate to an entity defined in a different class or bundle, enter a full namespace as the targetEntity.

Next, since each `Product` class can relate to exactly one `Category` object, you'll want to add a `$category` property to the `Product` class:

Listing 8-34

```
// src/Acme/StoreBundle/Entity/Product.php
// ...

class Product
{
    // ...

    /**
     * @ORM\ManyToOne(targetEntity="Category", inversedBy="products")
     * @ORM\JoinColumn(name="category_id", referencedColumnName="id")
     */
    protected $category;
}
```

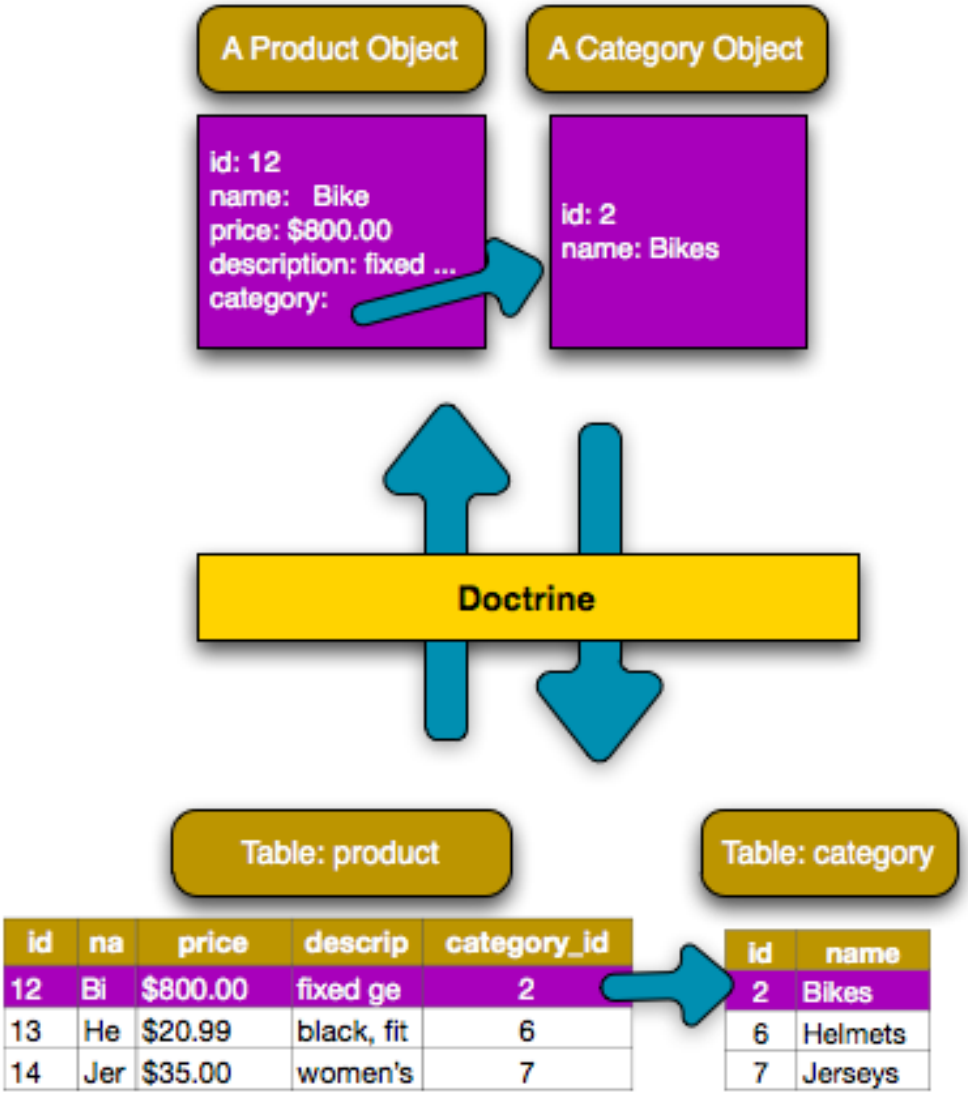
Finally, now that you've added a new property to both the `Category` and `Product` classes, tell Doctrine to generate the missing getter and setter methods for you:

Listing 8-35

```
php app/console doctrine:generate:entities Acme
```

Ignore the Doctrine metadata for a moment. You now have two classes - `Category` and `Product` with a natural one-to-many relationship. The `Category` class holds an array of `Product` objects and the `Product` object can hold one `Category` object. In other words - you've built your classes in a way that makes sense for your needs. The fact that the data needs to be persisted to a database is always secondary.

Now, look at the metadata above the `$category` property on the `Product` class. The information here tells doctrine that the related class is `Category` and that it should store the `id` of the category record on a `category_id` field that lives on the `product` table. In other words, the related `Category` object will be stored on the `$category` property, but behind the scenes, Doctrine will persist this relationship by storing the category's id value on a `category_id` column of the `product` table.



The metadata above the `$products` property of the `Category` object is less important, and simply tells Doctrine to look at the `Product.category` property to figure out how the relationship is mapped.

Before you continue, be sure to tell Doctrine to add the new `category` table, and `product.category_id` column, and new foreign key:

Listing 8-36
```
php app/console doctrine:schema:update --force
```

This task should only be really used during development. For a more robust method of systematically updating your production database, read about *Doctrine migrations*.

Saving Related Entities

Now, let's see the code in action. Imagine you're inside a controller:

Listing 8-37
```
// ...
use Acme\StoreBundle\Entity\Category;
use Acme\StoreBundle\Entity\Product;
```

```php
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);
        // relate this product to the category
        $product->setCategory($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()
        );
    }
}
```

Now, a single row is added to both the `category` and `product` tables. The `product.category_id` column for the new product is set to whatever the `id` is of the new category. Doctrine manages the persistence of this relationship for you.

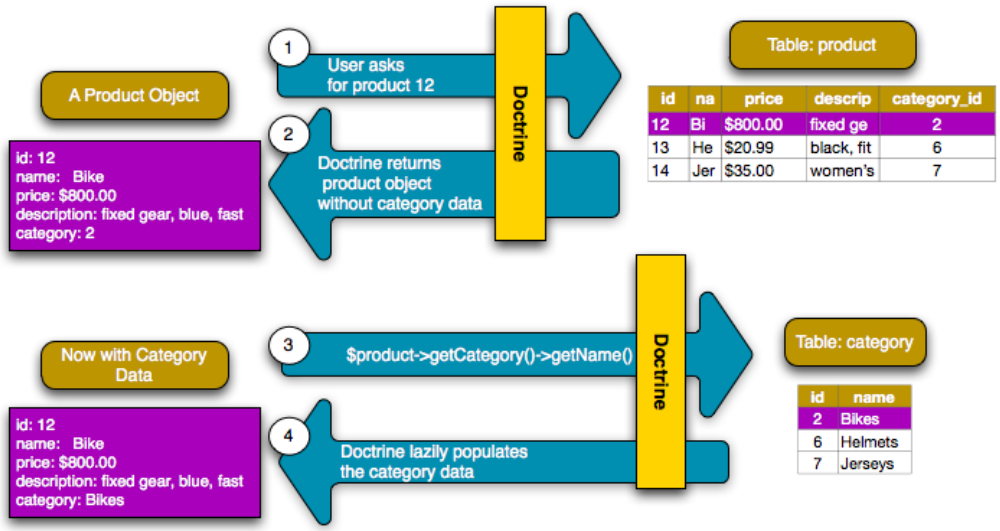
Fetching Related Objects

When you need to fetch associated objects, your workflow looks just like it did before. First, fetch a `$product` object and then access its related `Category`:

Listing 8-38

```php
public function showAction($id)
{
    $product = $this->getDoctrine()
        ->getRepository('AcmeStoreBundle:Product')
        ->find($id);

    $categoryName = $product->getCategory()->getName();

    // ...
}
```

In this example, you first query for a `Product` object based on the product's `id`. This issues a query for *just* the product data and hydrates the `$product` object with that data. Later, when you call `$product->getCategory()->getName()`, Doctrine silently makes a second query to find the `Category` that's related to this `Product`. It prepares the `$category` object and returns it to you.



What's important is the fact that you have easy access to the product's related category, but the category data isn't actually retrieved until you ask for the category (i.e. it's "lazily loaded").

You can also query in the other direction:

Listing 8-39

```
public function showProductAction($id)
{
    $category = $this->getDoctrine()
        ->getRepository('AcmeStoreBundle:Category')
        ->find($id);

    $products = $category->getProducts();

    // ...
}
```

In this case, the same things occurs: you first query out for a single `Category` object, and then Doctrine makes a second query to retrieve the related `Product` objects, but only once/if you ask for them (i.e. when you call `->getProducts()`). The `$products` variable is an array of all `Product` objects that relate to the given `Category` object via their `category_id` value.

Relationships and Proxy Classes

This "lazy loading" is possible because, when necessary, Doctrine returns a "proxy" object in place of the true object. Look again at the above example:

Listing 8-40

```
$product = $this->getDoctrine()
    ->getRepository('AcmeStoreBundle:Product')
    ->find($id);

$category = $product->getCategory();

// prints "Proxies\AcmeStoreBundleEntityCategoryProxy"
echo get_class($category);
```

This proxy object extends the true `Category` object, and looks and acts exactly like it. The difference is that, by using a proxy object, Doctrine can delay querying for the real `Category` data until you actually need that data (e.g. until you call `$category->getName()`).

The proxy classes are generated by Doctrine and stored in the cache directory. And though you'll probably never even notice that your `$category` object is actually a proxy object, it's important to keep in mind.

In the next section, when you retrieve the product and category data all at once (via a *join*), Doctrine will return the *true* `Category` object, since nothing needs to be lazily loaded.

Joining to Related Records

In the above examples, two queries were made - one for the original object (e.g. a `Category`) and one for the related object(s) (e.g. the `Product` objects).

Remember that you can see all of the queries made during a request via the web debug toolbar.

Of course, if you know up front that you'll need to access both objects, you can avoid the second query by issuing a join in the original query. Add the following method to the `ProductRepository` class:

Listing 8-41

```
// 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;
    }
}
```

Now, you can use this method in your controller to query for a `Product` object and its related `Category` with just one query:

Listing 8-42
```php
public function showAction($id)
{
    $product = $this->getDoctrine()
        ->getRepository('AcmeStoreBundle:Product')
        ->findOneByIdJoinedToCategory($id);

    $category = $product->getCategory();

    // ...
}
```

More Information on Associations

This section has been an introduction to one common type of entity relationship, the one-to-many relationship. For more advanced details and examples of how to use other types of relations (e.g. `one-to-one`, `many-to-many`), see Doctrine's *Association Mapping Documentation*[7].

If you're using annotations, you'll need to prepend all annotations with `ORM\` (e.g. `ORM\OneToMany`), which is not reflected in Doctrine's documentation. You'll also need to include the `use Doctrine\ORM\Mapping as ORM;` statement, which *imports* the `ORM` annotations prefix.

Configuration

Doctrine is highly configurable, though you probably won't ever need to worry about most of its options. To find out more about configuring Doctrine, see the Doctrine section of the *reference manual*.

Lifecycle Callbacks

Sometimes, you need to perform an action right before or after an entity is inserted, updated, or deleted. These types of actions are known as "lifecycle" callbacks, as they're callback methods that you need to execute during different stages of the lifecycle of an entity (e.g. the entity is inserted, updated, deleted, etc).

If you're using annotations for your metadata, start by enabling the lifecycle callbacks. This is not necessary if you're using YAML or XML for your mapping:

Listing 8-43
```php
/**
 * @ORM\Entity()
 * @ORM\HasLifecycleCallbacks()
 */
class Product
{
    // ...
}
```

Now, you can tell Doctrine to execute a method on any of the available lifecycle events. For example, suppose you want to set a `created` date column to the current date, only when the entity is first persisted (i.e. inserted):

Listing 8-44
```php
/**
 * @ORM\PrePersist
 */
```

7. http://docs.doctrine-project.org/projects/doctrine-orm/en/2.1/reference/association-mapping.html

```php
public function setCreatedValue()
{
    $this->created = new \DateTime();
}
```

The above example assumes that you've created and mapped a `created` property (not shown here).

Now, right before the entity is first persisted, Doctrine will automatically call this method and the `created` field will be set to the current date.

This can be repeated for any of the other lifecycle events, which include:

- `preRemove`
- `postRemove`
- `prePersist`
- `postPersist`
- `preUpdate`
- `postUpdate`
- `postLoad`
- `loadClassMetadata`

For more information on what these lifecycle events mean and lifecycle callbacks in general, see Doctrine's *Lifecycle Events documentation*[8]

Lifecycle Callbacks and Event Listeners

Notice that the `setCreatedValue()` method receives no arguments. This is always the case for lifecycle callbacks and is intentional: lifecycle callbacks should be simple methods that are concerned with internally transforming data in the entity (e.g. setting a created/updated field, generating a slug value).

If you need to do some heavier lifting - like perform logging or send an email - you should register an external class as an event listener or subscriber and give it access to whatever resources you need. For more information, see *Registering Event Listeners and Subscribers*.

Doctrine Extensions: Timestampable, Sluggable, etc.

Doctrine is quite flexible, and a number of third-party extensions are available that allow you to easily perform repeated and common tasks on your entities. These include thing such as *Sluggable*, *Timestampable*, *Loggable*, *Translatable*, and *Tree*.

For more information on how to find and use these extensions, see the cookbook article about *using common Doctrine extensions*.

Doctrine Field Types Reference

Doctrine comes with a large number of field types available. Each of these maps a PHP data type to a specific column type in whatever database you're using. The following types are supported in Doctrine:

- **Strings**

8. `http://docs.doctrine-project.org/projects/doctrine-orm/en/2.1/reference/events.html#lifecycle-events`

    - `string` (used for shorter strings)
    - `text` (used for larger strings)

- **Numbers**
    - `integer`
    - `smallint`
    - `bigint`
    - `decimal`
    - `float`

- **Dates and Times** (use a *DateTime*[9] object for these fields in PHP)
    - `date`
    - `time`
    - `datetime`

- **Other Types**
    - `boolean`
    - `object` (serialized and stored in a `CLOB` field)
    - `array` (serialized and stored in a `CLOB` field)

For more information, see Doctrine's *Mapping Types documentation*[10].

Field Options

Each field can have a set of options applied to it. The available options include `type` (defaults to `string`), `name`, `length`, `unique` and `nullable`. Take a few examples:

Listing 8-45

```
/**
 * A string field with length 255 that cannot be null
 * (reflecting the default values for the "type", "length" and *nullable* options)
 *
 * @ORM\Column()
 */
protected $name;

/**
 * A string field of length 150 that persists to an "email_address" column
 * and has a unique index.
 *
 * @ORM\Column(name="email_address", unique=true, length=150)
 */
protected $email;
```

There are a few more options not listed here. For more details, see Doctrine's *Property Mapping documentation*[11]

9. http://php.net/manual/en/class.datetime.php
10. http://docs.doctrine-project.org/projects/doctrine-orm/en/2.1/reference/basic-mapping.html#doctrine-mapping-types
11. http://docs.doctrine-project.org/projects/doctrine-orm/en/2.1/reference/basic-mapping.html#property-mapping

Console Commands

The Doctrine2 ORM integration offers several console commands under the `doctrine` namespace. To view the command list you can run the console without any arguments:

```
php app/console
```

Listing 8-46

A list of available command will print out, many of which start with the `doctrine:` prefix. You can find out more information about any of these commands (or any Symfony command) by running the `help` command. For example, to get details about the `doctrine:database:create` task, run:

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

Listing 8-47

Some notable or interesting tasks include:

- `doctrine:ensure-production-settings` - checks to see if the current environment is configured efficiently for production. This should always be run in the `prod` environment:

  ```
  php app/console doctrine:ensure-production-settings --env=prod
  ```

  Listing 8-48

- `doctrine:mapping:import` - allows Doctrine to introspect an existing database and create mapping information. For more information, see *How to generate Entities from an Existing Database*.
- `doctrine:mapping:info` - tells you all of the entities that Doctrine is aware of and whether or not there are any basic errors with the mapping.
- `doctrine:query:dql` and `doctrine:query:sql` - allow you to execute DQL or SQL queries directly from the command line.

To be able to load data fixtures to your database, you will need to have the `DoctrineFixturesBundle` bundle installed. To learn how to do it, read the "*DoctrineFixturesBundle*" entry of the documentation.

Summary

With Doctrine, you can focus on your objects and how they're useful in your application and worry about database persistence second. This is because Doctrine allows you to use any PHP object to hold your data and relies on mapping metadata information to map an object's data to a particular database table.

And even though Doctrine revolves around a simple concept, it's incredibly powerful, allowing you to create complex queries and subscribe to events that allow you to take different actions as objects go through their persistence lifecycle.

For more information about Doctrine, see the *Doctrine* section of the *cookbook*, which includes the following articles:

- *DoctrineFixturesBundle*
- *Doctrine Extensions: Timestampable, Sluggable, Translatable, etc.*

Chapter 9
Databases and Propel

Let's face it, one of the most common and challenging tasks for any application involves persisting and reading information to and from a database. Symfony2 does not come integrated with any ORMs but the Propel integration is easy. To get started, read *Working With Symfony2*[1].

A Simple Example: A Product

In this section, you'll configure your database, create a `Product` object, persist it to the database and fetch it back out.

Code along with the example

If you want to follow along with the example in this chapter, create an `AcmeStoreBundle` via: `php app/console generate:bundle --namespace=Acme/StoreBundle`.

Configuring the Database

Before you can start, you'll need to configure your database connection information. By convention, this information is usually configured in an `app/config/parameters.ini` file:

Listing 9-1
```ini
;app/config/parameters.ini
[parameters]
    database_driver   = mysql
    database_host     = localhost
    database_name     = test_project
    database_user     = root
    database_password = password
    database_charset  = UTF8
```

1. http://www.propelorm.org/cookbook/symfony2/working-with-symfony2.html#installation

Defining the configuration via `parameters.ini` is just a convention. The parameters defined in that file are referenced by the main configuration file when setting up Propel:

Listing 9-2

```
propel:
    dbal:
        driver:     %database_driver%
        user:       %database_user%
        password:   %database_password%
        dsn:
%database_driver%:host=%database_host%;dbname=%database_name%;charset=%database_charset%
```

Now that Propel knows about your database, Symfony2 can create the database for you:

Listing 9-3

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

In this example, you have one configured connection, named `default`. If you want to configure more than one connection, read the PropelBundle configuration section.

Creating a Model Class

In the Propel world, ActiveRecord classes are known as **models** because classes generated by Propel contain some business logic.

For people who use Symfony2 with Doctrine2, **models** are equivalent to **entities**.

Suppose you're building an application where products need to be displayed. First, create a `schema.xml` file inside the `Resources/config` directory of your `AcmeStoreBundle`:

Listing 9-4

```
<?xml version="1.0" encoding="UTF-8"?>
<database name="default" namespace="Acme\StoreBundle\Model" defaultIdMethod="native">
    <table name="product">
        <column name="id" type="integer" required="true" primaryKey="true"
autoIncrement="true" />
        <column name="name" type="varchar" primaryString="true" size="100" />
        <column name="price" type="decimal" />
        <column name="description" type="longvarchar" />
    </table>
</database>
```

Building the Model

After creating your `schema.xml`, generate your model from it by running:

Listing 9-5

```
php app/console propel:model:build
```

This generates each model class to quickly develop your application in the `Model/` directory the `AcmeStoreBundle` bundle.

Creating the Database Tables/Schema

Now you have a usable `Product` class and all you need to persist it. Of course, you don't yet have the corresponding `product` table in your database. Fortunately, Propel can automatically create all the database tables needed for every known model in your application. To do this, run:

Listing 9-6
```
php app/console propel:sql:build

php app/console propel:sql:insert --force
```

Your database now has a fully-functional `product` table with columns that match the schema you've specified.

You can run the last three commands combined by using the following command: `php app/console propel:build --insert-sql`.

Persisting Objects to the Database

Now that you have a `Product` object and corresponding `product` table, you're ready to persist data to the database. From inside a controller, this is pretty easy. Add the following method to the `DefaultController` of the bundle:

Listing 9-7
```
// src/Acme/StoreBundle/Controller/DefaultController.php
use Acme\StoreBundle\Model\Product;
use Symfony\Component\HttpFoundation\Response;
// ...

public function createAction()
{
    $product = new Product();
    $product->setName('A Foo Bar');
    $product->setPrice(19.99);
    $product->setDescription('Lorem ipsum dolor');

    $product->save();

    return new Response('Created product id '.$product->getId());
}
```

In this piece of code, you instantiate and work with the `$product` object. When you call the `save()` method on it, you persist it to the database. No need to use other services, the object knows how to persist itself.

If you're following along with this example, you'll need to create a *route* that points to this action to see it in action.

Fetching Objects from the Database

Fetching an object back from the database is even easier. For example, suppose you've configured a route to display a specific `Product` based on its `id` value:

Listing 9-8
```
use Acme\StoreBundle\Model\ProductQuery;
```

```
public function showAction($id)
{
    $product = ProductQuery::create()
        ->findPk($id);

    if (!$product) {
        throw $this->createNotFoundException('No product found for id '.$id);
    }

    // do something, like pass the $product object into a template
}
```

Updating an Object

Once you've fetched an object from Propel, updating it is easy. Suppose you have a route that maps a product id to an update action in a controller:

Listing 9-9

```
use Acme\StoreBundle\Model\ProductQuery;

public function updateAction($id)
{
    $product = ProductQuery::create()
        ->findPk($id);

    if (!$product) {
        throw $this->createNotFoundException('No product found for id '.$id);
    }

    $product->setName('New product name!');
    $product->save();

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

Updating an object involves just three steps:

1. fetching the object from Propel;
2. modifying the object;
3. saving it.

Deleting an Object

Deleting an object is very similar, but requires a call to the `delete()` method on the object:

Listing 9-10

```
$product->delete();
```

Querying for Objects

Propel provides generated `Query` classes to run both basic and complex queries without any work:

Listing 9-11

```
\Acme\StoreBundle\Model\ProductQuery::create()->findPk($id);

\Acme\StoreBundle\Model\ProductQuery::create()
    ->filterByName('Foo')
    ->findOne();
```

Imagine that you want to query for products which cost more than 19.99, ordered from cheapest to most expensive. From inside a controller, do the following:

Listing 9-12
```
$products = \Acme\StoreBundle\Model\ProductQuery::create()
    ->filterByPrice(array('min' => 19.99))
    ->orderByPrice()
    ->find();
```

In one line, you get your products in a powerful oriented object way. No need to waste your time with SQL or whatever, Symfony2 offers fully object oriented programming and Propel respects the same philosophy by providing an awesome abstraction layer.

If you want to reuse some queries, you can add your own methods to the `ProductQuery` class:

Listing 9-13
```
// src/Acme/StoreBundle/Model/ProductQuery.php

class ProductQuery extends BaseProductQuery
{
    public function filterByExpensivePrice()
    {
        return $this
            ->filterByPrice(array('min' => 1000))
    }
}
```

But note that Propel generates a lot of methods for you and a simple `findAllOrderedByName()` can be written without any effort:

Listing 9-14
```
\Acme\StoreBundle\Model\ProductQuery::create()
    ->orderByName()
    ->find();
```

Relationships/Associations

Suppose that the products in your application all belong to exactly one "category". In this case, you'll need a `Category` object and a way to relate a `Product` object to a `Category` object.

Start by adding the `category` definition in your `schema.xml`:

Listing 9-15
```
<database name="default" namespace="Acme\StoreBundle\Model" defaultIdMethod="native">
    <table name="product">
        <column name="id" type="integer" required="true" primaryKey="true"
autoIncrement="true" />
        <column name="name" type="varchar" primaryString="true" size="100" />
        <column name="price" type="decimal" />
        <column name="description" type="longvarchar" />

        <column name="category_id" type="integer" />
        <foreign-key foreignTable="category">
            <reference local="category_id" foreign="id" />
        </foreign-key>
    </table>

    <table name="category">
        <column name="id" type="integer" required="true" primaryKey="true"
autoIncrement="true" />
        <column name="name" type="varchar" primaryString="true" size="100" />
   </table>
</database>
```

Create the classes:

```
php app/console propel:model:build
```

Listing 9-16

Assuming you have products in your database, you don't want lose them. Thanks to migrations, Propel will be able to update your database without losing existing data.

```
php app/console propel:migration:generate-diff
```

Listing 9-17

```
php app/console propel:migration:migrate
```

Your database has been updated, you can continue to write your application.

Saving Related Objects

Now, let's see the code in action. Imagine you're inside a controller:

Listing 9-18

```
// ...
use Acme\StoreBundle\Model\Category;
use Acme\StoreBundle\Model\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);
        // relate this product to the category
        $product->setCategory($category);

        // save the whole
        $product->save();

        return new Response(
            'Created product id: '.$product->getId().' and category id: '.$category->getId()
        );
    }
}
```

Now, a single row is added to both the `category` and product tables. The `product.category_id` column for the new product is set to whatever the id is of the new category. Propel manages the persistence of this relationship for you.

Fetching Related Objects

When you need to fetch associated objects, your workflow looks just like it did before. First, fetch a `$product` object and then access its related `Category`:

Listing 9-19

```
// ...
use Acme\StoreBundle\Model\ProductQuery;

public function showAction($id)
{
    $product = ProductQuery::create()
        ->joinWithCategory()
```

```
        ->findPk($id);

    $categoryName = $product->getCategory()->getName();

    // ...
}
```

Note, in the above example, only one query was made.

More information on Associations

You will find more information on relations by reading the dedicated chapter on *Relationships*[2].

Lifecycle Callbacks

Sometimes, you need to perform an action right before or after an object is inserted, updated, or deleted. These types of actions are known as "lifecycle" callbacks or "hooks", as they're callback methods that you need to execute during different stages of the lifecycle of an object (e.g. the object is inserted, updated, deleted, etc).

To add a hook, just add a new method to the object class:

Listing 9-20

```
// src/Acme/StoreBundle/Model/Product.php

// ...

class Product extends BaseProduct
{
    public function preInsert(\PropelPDO $con = null)
    {
        // do something before the object is inserted
    }
}
```

Propel provides the following hooks:

- `preInsert()` code executed before insertion of a new object
- `postInsert()` code executed after insertion of a new object
- `preUpdate()` code executed before update of an existing object
- `postUpdate()` code executed after update of an existing object
- `preSave()` code executed before saving an object (new or existing)
- `postSave()` code executed after saving an object (new or existing)
- `preDelete()` code executed before deleting an object
- `postDelete()` code executed after deleting an object

Behaviors

All bundled behaviors in Propel are working with Symfony2. To get more information about how to use Propel behaviors, look at the Behaviors reference section.

2. http://www.propelorm.org/documentation/04-relationships.html

Commands

You should read the dedicated section for *Propel commands in Symfony2*[3].

3. http://www.propelorm.org/cookbook/symfony2/working-with-symfony2#commands

Chapter 10
Testing

Whenever you write a new line of code, you also potentially add new bugs. To build better and more reliable applications, you should test your code using both functional and unit tests.

The PHPUnit Testing Framework

Symfony2 integrates with an independent library - called PHPUnit - to give you a rich testing framework. This chapter won't cover PHPUnit itself, but it has its own excellent *documentation*[1].

Symfony2 works with PHPUnit 3.5.11 or later.

Each test - whether it's a unit test or a functional test - is a PHP class that should live in the *Tests/* subdirectory of your bundles. If you follow this rule, then you can run all of your application's tests with the following command:

Listing 10-1
```
# specify the configuration directory on the command line
$ phpunit -c app/
```

The `-c` option tells PHPUnit to look in the `app/` directory for a configuration file. If you're curious about the PHPUnit options, check out the `app/phpunit.xml.dist` file.

Code coverage can be generated with the `--coverage-html` option.

1. `http://www.phpunit.de/manual/3.5/en/`

Unit Tests

A unit test is usually a test against a specific PHP class. If you want to test the overall behavior of your application, see the section about Functional Tests.

Writing Symfony2 unit tests is no different than writing standard PHPUnit unit tests. Suppose, for example, that you have an *incredibly* simple class called `Calculator` in the `Utility/` directory of your bundle:

Listing 10-2

```
// src/Acme/DemoBundle/Utility/Calculator.php
namespace Acme\DemoBundle\Utility;

class Calculator
{
    public function add($a, $b)
    {
        return $a + $b;
    }
}
```

To test this, create a `CalculatorTest` file in the `Tests/Utility` directory of your bundle:

Listing 10-3

```
// 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);
    }
}
```

By convention, the `Tests/` sub-directory should replicate the directory of your bundle. So, if you're testing a class in your bundle's `Utility/` directory, put the test in the `Tests/Utility/` directory.

Just like in your real application - autoloading is automatically enabled via the `bootstrap.php.cache` file (as configured by default in the `phpunit.xml.dist` file).

Running tests for a given file or directory is also very easy:

Listing 10-4

```
# 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

# run all tests for the entire Bundle
$ phpunit -c app src/Acme/DemoBundle/
```

Functional Tests

Functional tests check the integration of the different layers of an application (from the routing to the views). They are no different from unit tests as far as PHPUnit is concerned, but they have a very specific workflow:

- Make a request;
- Test the response;
- Click on a link or submit a form;
- Test the response;
- Rinse and repeat.

Your First Functional Test

Functional tests are simple PHP files that typically live in the `Tests/Controller` directory of your bundle. If you want to test the pages handled by your `DemoController` class, start by creating a new `DemoControllerTest.php` file that extends a special `WebTestCase` class.

For example, the Symfony2 Standard Edition provides a simple functional test for its `DemoController` (*DemoControllerTest*[2]) that reads as follows:

Listing 10-5

```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->assertGreaterThan(0, $crawler->filter('html:contains("Hello
Fabien")')->count());
    }
}
```

To run your functional tests, the `WebTestCase` class bootstraps the kernel of your application. In most cases, this happens automatically. However, if your kernel is in a non-standard directory, you'll need to modify your `phpunit.xml.dist` file to set the `KERNEL_DIR` environment variable to the directory of your kernel:

Listing 10-6

```xml
<phpunit>
    <!-- ... -->
    <php>
        <server name="KERNEL_DIR" value="/path/to/your/app/" />
    </php>
    <!-- ... -->
</phpunit>
```

The `createClient()` method returns a client, which is like a browser that you'll use to crawl your site:

2. https://github.com/symfony/symfony-standard/blob/master/src/Acme/DemoBundle/Tests/Controller/DemoControllerTest.php

```php
$crawler = $client->request('GET', '/demo/hello/Fabien');
```

Listing 10-7

The `request()` method (see *more about the request method*) returns a *`Crawler`*[3] object which can be used to select elements in the Response, click on links, and submit forms.

The Crawler only works when the response is an XML or an HTML document. To get the raw content response, call `$client->getResponse()->getContent()`.

Click on a link by first selecting it with the Crawler using either an XPath expression or a CSS selector, then use the Client to click on it. For example, the following code finds all links with the text `Greet`, then selects the second one, and ultimately clicks on it:

```php
$link = $crawler->filter('a:contains("Greet")')->eq(1)->link();

$crawler = $client->click($link);
```

Listing 10-8

Submitting a form is very similar; select a form button, optionally override some form values, and submit the corresponding form:

```php
$form = $crawler->selectButton('submit')->form();

// set some values
$form['name'] = 'Lucas';
$form['form_name[subject]'] = 'Hey there!';

// submit the form
$crawler = $client->submit($form);
```

Listing 10-9

The form can also handle uploads and contains methods to fill in different types of form fields (e.g. `select()` and `tick()`). For details, see the Forms section below.

Now that you can easily navigate through an application, use assertions to test that it actually does what you expect it to. Use the Crawler to make assertions on the DOM:

```php
// Assert that the response matches a given CSS selector.
$this->assertGreaterThan(0, $crawler->filter('h1')->count());
```

Listing 10-10

Or, test against the Response content directly if you just want to assert that the content contains some text, or if the Response is not an XML/HTML document:

```php
$this->assertRegExp('/Hello Fabien/', $client->getResponse()->getContent());
```

Listing 10-11

3. http://api.symfony.com/2.0/Symfony/Component/DomCrawler/Crawler.html

More about the `request()` method:

The full signature of the `request()` method is:

Listing 10-12
```
request(
    $method,
    $uri,
    array $parameters = array(),
    array $files = array(),
    array $server = array(),
    $content = null,
    $changeHistory = true
)
```

The `server` array is the raw values that you'd expect to normally find in the PHP *$_SERVER*[4] superglobal. For example, to set the *Content-Type* and *Referer* HTTP headers, you'd pass the following:

Listing 10-13
```
$client->request(
    'GET',
    '/demo/hello/Fabien',
    array(),
    array(),
    array(
        'CONTENT_TYPE' => 'application/json',
        'HTTP_REFERER' => '/foo/bar',
    )
);
```

4. http://php.net/manual/en/reserved.variables.server.php

Useful Assertions

To get you started faster, here is a list of the most common and useful test assertions:

Listing 10-14

```
// Assert that there is more than one h2 tag with the class "subtitle"
$this->assertGreaterThan(0, $crawler->filter('h2.subtitle')->count());

// Assert that there are exactly 4 h2 tags on the page
$this->assertCount(4, $crawler->filter('h2')->count());

// Assert that the "Content-Type" header is "application/json"
$this->assertTrue($client->getResponse()->headers->contains('Content-Type', 'application/
json'));

// Assert that the response content matches a regexp.
$this->assertRegExp('/foo/', $client->getResponse()->getContent());

// Assert that the response status code is 2xx
$this->assertTrue($client->getResponse()->isSuccessful());
// Assert that the response status code is 404
$this->assertTrue($client->getResponse()->isNotFound());
// Assert a specific 200 status code
$this->assertEquals(200, $client->getResponse()->getStatusCode());

// Assert that the response is a redirect to /demo/contact
$this->assertTrue($client->getResponse()->isRedirect('/demo/contact'));
// or simply check that the response is a redirect to any URL
$this->assertTrue($client->getResponse()->isRedirect());
```

Working with the Test Client

The Test Client simulates an HTTP client like a browser and makes requests into your Symfony2 application:

Listing 10-15

```
$crawler = $client->request('GET', '/hello/Fabien');
```

The `request()` method takes the HTTP method and a URL as arguments and returns a `Crawler` instance.

Use the Crawler to find DOM elements in the Response. These elements can then be used to click on links and submit forms:

Listing 10-16

```
$link = $crawler->selectLink('Go elsewhere...')->link();
$crawler = $client->click($link);

$form = $crawler->selectButton('validate')->form();
$crawler = $client->submit($form, array('name' => 'Fabien'));
```

The `click()` and `submit()` methods both return a `Crawler` object. These methods are the best way to browse your application as it takes care of a lot of things for you, like detecting the HTTP method from a form and giving you a nice API for uploading files.

You will learn more about the `Link` and `Form` objects in the *Crawler* section below.

The `request` method can also be used to simulate form submissions directly or perform more complex requests:

Listing 10-17
```
// Directly submit a form (but using the Crawler is easier!)
$client->request('POST', '/submit', array('name' => 'Fabien'));

// Form submission with a file upload
use Symfony\Component\HttpFoundation\File\UploadedFile;

$photo = new UploadedFile(
    '/path/to/photo.jpg',
    'photo.jpg',
    'image/jpeg',
    123
);
// or
$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)
);

// Perform a DELETE requests, and pass HTTP headers
$client->request(
    'DELETE',
    '/post/12',
    array(),
    array(),
    array('PHP_AUTH_USER' => 'username', 'PHP_AUTH_PW' => 'pa$$word')
);
```

Last but not least, you can force each request to be executed in its own PHP process to avoid any side-effects when working with several clients in the same script:

Listing 10-18
```
$client->insulate();
```

Browsing

The Client supports many operations that can be done in a real browser:

Listing 10-19
```
$client->back();
$client->forward();
$client->reload();

// Clears all cookies and the history
$client->restart();
```

Accessing Internal Objects

If you use the client to test your application, you might want to access the client's internal objects:

```
$history   = $client->getHistory();
$cookieJar = $client->getCookieJar();
```

Listing 10-20

You can also get the objects related to the latest request:

```
$request  = $client->getRequest();
$response = $client->getResponse();
$crawler  = $client->getCrawler();
```

Listing 10-21

If your requests are not insulated, you can also access the `Container` and the `Kernel`:

```
$container = $client->getContainer();
$kernel    = $client->getKernel();
```

Listing 10-22

Accessing the Container

It's highly recommended that a functional test only tests the Response. But under certain very rare circumstances, you might want to access some internal objects to write assertions. In such cases, you can access the dependency injection container:

```
$container = $client->getContainer();
```

Listing 10-23

Be warned that this does not work if you insulate the client or if you use an HTTP layer. For a list of services available in your application, use the `container:debug` console task.

If the information you need to check is available from the profiler, use it instead.

Accessing the Profiler Data

On each request, the Symfony profiler collects and stores a lot of data about the internal handling of that request. For example, the profiler could be used to verify that a given page executes less than a certain number of database queries when loading.

To get the Profiler for the last request, do the following:

```
$profile = $client->getProfile();
```

Listing 10-24

For specific details on using the profiler inside a test, see the *How to use the Profiler in a Functional Test* cookbook entry.

Redirecting

When a request returns a redirect response, the client does not follow it automatically. You can examine the response and force a redirection afterwards with the `followRedirect()` method:

```
$crawler = $client->followRedirect();
```

Listing 10-25

If you want the client to automatically follow all redirects, you can force him with the `followRedirects()` method:

```
$client->followRedirects();
```

Listing 10-26

The Crawler

A Crawler instance is returned each time you make a request with the Client. It allows you to traverse HTML documents, select nodes, find links and forms.

Traversing

Like jQuery, the Crawler has methods to traverse the DOM of an HTML/XML document. For example, the following finds all `input[type=submit]` elements, selects the last one on the page, and then selects its immediate parent element:

Listing 10-27
```
$newCrawler = $crawler->filter('input[type=submit]')
    ->last()
    ->parents()
    ->first()
;
```

Many other methods are also available:

Method	Description
`filter('h1.title')`	Nodes that match the CSS selector
`filterXpath('h1')`	Nodes that match the XPath expression
`eq(1)`	Node for the specified index
`first()`	First node
`last()`	Last node
`siblings()`	Siblings
`nextAll()`	All following siblings
`previousAll()`	All preceding siblings
`parents()`	Returns the parent nodes
`children()`	Returns children nodes
`reduce($lambda)`	Nodes for which the callable does not return false

Since each of these methods returns a new `Crawler` instance, you can narrow down your node selection by chaining the method calls:

Listing 10-28
```
$crawler
    ->filter('h1')
    ->reduce(function ($node, $i)
    {
        if (!$node->getAttribute('class')) {
            return false;
        }
    })
    ->first();
```

Use the `count()` function to get the number of nodes stored in a Crawler: `count($crawler)`

Extracting Information

The Crawler can extract information from the nodes:

Listing 10-29

```
// Returns the attribute value for the first node
$crawler->attr('class');

// Returns the node value for the first node
$crawler->text();

// Extracts an array of attributes for all nodes (_text returns the node value)
// returns an array for each element in crawler, each with the value and href
$info = $crawler->extract(array('_text', 'href'));

// Executes a lambda for each node and return an array of results
$data = $crawler->each(function ($node, $i)
{
    return $node->attr('href');
});
```

Links

To select links, you can use the traversing methods above or the convenient `selectLink()` shortcut:

Listing 10-30

```
$crawler->selectLink('Click here');
```

This selects all links that contain the given text, or clickable images for which the `alt` attribute contains the given text. Like the other filtering methods, this returns another `Crawler` object.

Once you've selected a link, you have access to a special `Link` object, which has helpful methods specific to links (such as `getMethod()` and `getUri()`). To click on the link, use the Client's `click()` method and pass it a `Link` object:

Listing 10-31

```
$link = $crawler->selectLink('Click here')->link();

$client->click($link);
```

Forms

Just like links, you select forms with the `selectButton()` method:

Listing 10-32

```
$buttonCrawlerNode = $crawler->selectButton('submit');
```

Notice that we select form buttons and not forms as a form can have several buttons; if you use the traversing API, keep in mind that you must look for a button.

The `selectButton()` method can select `button` tags and submit `input` tags. It uses several different parts of the buttons to find them:

- The `value` attribute value;
- The `id` or `alt` attribute value for images;
- The `id` or `name` attribute value for `button` tags.

Once you have a Crawler representing a button, call the `form()` method to get a `Form` instance for the form wrapping the button node:

Listing 10-33
```
$form = $buttonCrawlerNode->form();
```

When calling the `form()` method, you can also pass an array of field values that overrides the default ones:

Listing 10-34
```
$form = $buttonCrawlerNode->form(array(
    'name'              => 'Fabien',
    'my_form[subject]'  => 'Symfony rocks!',
));
```

And if you want to simulate a specific HTTP method for the form, pass it as a second argument:

Listing 10-35
```
$form = $buttonCrawlerNode->form(array(), 'DELETE');
```

The Client can submit `Form` instances:

Listing 10-36
```
$client->submit($form);
```

The field values can also be passed as a second argument of the `submit()` method:

Listing 10-37
```
$client->submit($form, array(
    'name'              => 'Fabien',
    'my_form[subject]'  => 'Symfony rocks!',
));
```

For more complex situations, use the `Form` instance as an array to set the value of each field individually:

Listing 10-38
```
// Change the value of a field
$form['name'] = 'Fabien';
$form['my_form[subject]'] = 'Symfony rocks!';
```

There is also a nice API to manipulate the values of the fields according to their type:

Listing 10-39
```
// Select an option or a radio
$form['country']->select('France');

// Tick a checkbox
$form['like_symfony']->tick();

// Upload a file
$form['photo']->upload('/path/to/lucas.jpg');
```

You can get the values that will be submitted by calling the `getValues()` method on the `Form` object. The uploaded files are available in a separate array returned by `getFiles()`. The `getPhpValues()` and `getPhpFiles()` methods also return the submitted values, but in the PHP format (it converts the keys with square brackets notation - e.g. `my_form[subject]` - to PHP arrays).

Testing Configuration

The Client used by functional tests creates a Kernel that runs in a special `test` environment. Since Symfony loads the `app/config/config_test.yml` in the `test` environment, you can tweak any of your application's settings specifically for testing.

For example, by default, the swiftmailer is configured to *not* actually deliver emails in the `test` environment. You can see this under the `swiftmailer` configuration option:

```
# app/config/config_test.yml
# ...

swiftmailer:
    disable_delivery: true
```

Listing 10-40

You can also use a different environment entirely, or override the default debug mode (`true`) by passing each as options to the `createClient()` method:

```
$client = static::createClient(array(
    'environment' => 'my_test_env',
    'debug'       => false,
));
```

Listing 10-41

If your application behaves according to some HTTP headers, pass them as the second argument of `createClient()`:

```
$client = static::createClient(array(), array(
    'HTTP_HOST'       => 'en.example.com',
    'HTTP_USER_AGENT' => 'MySuperBrowser/1.0',
));
```

Listing 10-42

You can also override HTTP headers on a per request basis:

```
$client->request('GET', '/', array(), array(), array(
    'HTTP_HOST'       => 'en.example.com',
    'HTTP_USER_AGENT' => 'MySuperBrowser/1.0',
));
```

Listing 10-43

The test client is available as a service in the container in the `test` environment (or wherever the *framework.test* option is enabled). This means you can override the service entirely if you need to.

PHPUnit Configuration

Each application has its own PHPUnit configuration, stored in the `phpunit.xml.dist` file. You can edit this file to change the defaults or create a `phpunit.xml` file to tweak the configuration for your local machine.

Store the `phpunit.xml.dist` file in your code repository, and ignore the `phpunit.xml` file.

By default, only the tests stored in "standard" bundles are run by the `phpunit` command (standard being tests in the `src/*/Bundle/Tests` or `src/*/Bundle/*Bundle/Tests` directories) But you can easily add more directories. For instance, the following configuration adds the tests from the installed third-party bundles:

```
<!-- hello/phpunit.xml.dist -->
<testsuites>
    <testsuite name="Project Test Suite">
        <directory>../src/*/*Bundle/Tests</directory>
        <directory>../src/Acme/Bundle/*Bundle/Tests</directory>
    </testsuite>
</testsuites>
```

Listing 10-44

To include other directories in the code coverage, also edit the `<filter>` section:

Listing 10-45
```
<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>
```

Learn more from the Cookbook

- *How to simulate HTTP Authentication in a Functional Test*
- *How to test the Interaction of several Clients*
- *How to use the Profiler in a Functional Test*

Chapter 11
Validation

Validation is a very common task in web applications. Data entered in forms needs to be validated. Data also needs to be validated before it is written into a database or passed to a web service.

Symfony2 ships with a *Validator*[1] component that makes this task easy and transparent. This component is based on the *JSR303 Bean Validation specification*[2]. What? A Java specification in PHP? You heard right, but it's not as bad as it sounds. Let's look at how it can be used in PHP.

The Basics of Validation

The best way to understand validation is to see it in action. To start, suppose you've created a plain-old-PHP object that you need to use somewhere in your application:

Listing 11-1

```
// src/Acme/BlogBundle/Entity/Author.php
namespace Acme\BlogBundle\Entity;

class Author
{
    public $name;
}
```

So far, this is just an ordinary class that serves some purpose inside your application. The goal of validation is to tell you whether or not the data of an object is valid. For this to work, you'll configure a list of rules (called *constraints*) that the object must follow in order to be valid. These rules can be specified via a number of different formats (YAML, XML, annotations, or PHP).

For example, to guarantee that the `$name` property is not empty, add the following:

Listing 11-2

```
# src/Acme/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\Author:
    properties:
        name:
            - NotBlank: ~
```

1. https://github.com/symfony/Validator
2. http://jcp.org/en/jsr/detail?id=303

Protected and private properties can also be validated, as well as "getter" methods (see *validator-constraint-targets*).

Using the `validator` Service

Next, to actually validate an `Author` object, use the `validate` method on the `validator` service (class *`Validator`*[3]). The job of the `validator` is easy: to read the constraints (i.e. rules) of a class and verify whether or not the data on the object satisfies those constraints. If validation fails, an array of errors is returned. Take this simple example from inside a controller:

Listing 11-3
```php
use Symfony\Component\HttpFoundation\Response;
use Acme\BlogBundle\Entity\Author;
// ...

public function indexAction()
{
    $author = new Author();
    // ... do something to the $author object

    $validator = $this->get('validator');
    $errors = $validator->validate($author);

    if (count($errors) > 0) {
        return new Response(print_r($errors, true));
    } else {
        return new Response('The author is valid! Yes!');
    }
}
```

If the `$name` property is empty, you will see the following error message:

Listing 11-4
```
Acme\BlogBundle\Author.name:
    This value should not be blank
```

If you insert a value into the `name` property, the happy success message will appear.

Most of the time, you won't interact directly with the `validator` service or need to worry about printing out the errors. Most of the time, you'll use validation indirectly when handling submitted form data. For more information, see the *Validation and Forms*.

You could also pass the collection of errors into a template.

Listing 11-5
```php
if (count($errors) > 0) {
    return $this->render('AcmeBlogBundle:Author:validate.html.twig', array(
        'errors' => $errors,
    ));
} else {
    // ...
}
```

Inside the template, you can output the list of errors exactly as needed:

Listing 11-6

3. http://api.symfony.com/2.0/Symfony/Component/Validator/Validator.html

```
{# 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>
```

Each validation error (called a "constraint violation"), is represented by a *ConstraintViolation*[4] object.

Validation and Forms

The `validator` service can be used at any time to validate any object. In reality, however, you'll usually work with the `validator` indirectly when working with forms. Symfony's form library uses the `validator` service internally to validate the underlying object after values have been submitted and bound. The constraint violations on the object are converted into `FieldError` objects that can easily be displayed with your form. The typical form submission workflow looks like the following from inside a controller:

Listing 11-7

```
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()) {
            // the validation passed, do something with the $author object

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

    return $this->render('BlogBundle:Author:form.html.twig', array(
        'form' => $form->createView(),
    ));
}
```

This example uses an `AuthorType` form class, which is not shown here.

For more information, see the *Forms* chapter.

4. http://api.symfony.com/2.0/Symfony/Component/Validator/ConstraintViolation.html

Configuration

The Symfony2 validator is enabled by default, but you must explicitly enable annotations if you're using the annotation method to specify your constraints:

Listing 11-8
```
# app/config/config.yml
framework:
    validation: { enable_annotations: true }
```

Constraints

The `validator` is designed to validate objects against *constraints* (i.e. rules). In order to validate an object, simply map one or more constraints to its class and then pass it to the `validator` service.

Behind the scenes, a constraint is simply a PHP object that makes an assertive statement. In real life, a constraint could be: "The cake must not be burned". In Symfony2, constraints are similar: they are assertions that a condition is true. Given a value, a constraint will tell you whether or not that value adheres to the rules of the constraint.

Supported Constraints

Symfony2 packages a large number of the most commonly-needed constraints:

Basic Constraints

These are the basic constraints: use them to assert very basic things about the value of properties or the return value of methods on your object.

- *NotBlank*
- *Blank*
- *NotNull*
- *Null*
- *True*
- *False*
- *Type*

String Constraints

- *Email*
- *MinLength*
- *MaxLength*
- *Url*
- *Regex*
- *Ip*

Number Constraints

- *Max*
- *Min*

Date Constraints

- *Date*

- *DateTime*
- *Time*

Collection Constraints

- *Choice*
- *Collection*
- *UniqueEntity*
- *Language*
- *Locale*
- *Country*

File Constraints

- *File*
- *Image*

Other Constraints

- *Callback*
- *All*
- *Valid*

You can also create your own custom constraints. This topic is covered in the "*How to create a Custom Validation Constraint*" article of the cookbook.

Constraint Configuration

Some constraints, like *NotBlank*, are simple whereas others, like the *Choice* constraint, have several configuration options available. Suppose that the `Author` class has another property, `gender` that can be set to either "male" or "female":

Listing 11-9

```
# src/Acme/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\Author:
    properties:
        gender:
            - Choice: { choices: [male, female], message: Choose a valid gender. }
```

The options of a constraint can always be passed in as an array. Some constraints, however, also allow you to pass the value of one, "*default*", option in place of the array. In the case of the `Choice` constraint, the `choices` options can be specified in this way.

Listing 11-10

```
# src/Acme/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\Author:
    properties:
        gender:
            - Choice: [male, female]
```

This is purely meant to make the configuration of the most common option of a constraint shorter and quicker.

If you're ever unsure of how to specify an option, either check the API documentation for the constraint or play it safe by always passing in an array of options (the first method shown above).

Translation Constraint Messages

For information on translating the constraint messages, see *Translating Constraint Messages*.

Constraint Targets

Constraints can be applied to a class property (e.g. `name`) or a public getter method (e.g. `getFullName`). The first is the most common and easy to use, but the second allows you to specify more complex validation rules.

Properties

Validating class properties is the most basic validation technique. Symfony2 allows you to validate private, protected or public properties. The next listing shows you how to configure the `$firstName` property of an `Author` class to have at least 3 characters.

Listing 11-11
```yaml
# src/Acme/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\Author:
    properties:
        firstName:
            - NotBlank: ~
            - MinLength: 3
```

Getters

Constraints can also be applied to the return value of a method. Symfony2 allows you to add a constraint to any public method whose name starts with "get" or "is". In this guide, both of these types of methods are referred to as "getters".

The benefit of this technique is that it allows you to validate your object dynamically. For example, suppose you want to make sure that a password field doesn't match the first name of the user (for security reasons). You can do this by creating an `isPasswordLegal` method, and then asserting that this method must return `true`:

Listing 11-12
```yaml
# src/Acme/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\Author:
    getters:
        passwordLegal:
            - "True": { message: "The password cannot match your first name" }
```

Now, create the `isPasswordLegal()` method, and include the logic you need:

Listing 11-13
```php
public function isPasswordLegal()
{
    return ($this->firstName != $this->password);
}
```

The keen-eyed among you will have noticed that the prefix of the getter ("get" or "is") is omitted in the mapping. This allows you to move the constraint to a property with the same name later (or vice versa) without changing your validation logic.

Classes

Some constraints apply to the entire class being validated. For example, the *Callback* constraint is a generic constraint that's applied to the class itself. When that class is validated, methods specified by that constraint are simply executed so that each can provide more custom validation.

Validation Groups

So far, you've been able to add constraints to a class and ask whether or not that class passes all of the defined constraints. In some cases, however, you'll need to validate an object against only *some* of the constraints on that class. To do this, you can organize each constraint into one or more "validation groups", and then apply validation against just one group of constraints.

For example, suppose you have a `User` class, which is used both when a user registers and when a user updates his/her contact information later:

Listing 11-14

```
# 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
```

With this configuration, there are two validation groups:

- `Default` - contains the constraints not assigned to any other group;
- `registration` - contains the constraints on the `email` and `password` fields only.

To tell the validator to use a specific group, pass one or more group names as the second argument to the `validate()` method:

Listing 11-15

```
$errors = $validator->validate($author, array('registration'));
```

Of course, you'll usually work with validation indirectly through the form library. For information on how to use validation groups inside forms, see *Validation Groups*.

Validating Values and Arrays

So far, you've seen how you can validate entire objects. But sometimes, you just want to validate a simple value - like to verify that a string is a valid email address. This is actually pretty easy to do. From inside a controller, it looks like this:

Listing 11-16

```
// add this to the top of your class
use Symfony\Component\Validator\Constraints\Email;

public function addEmailAction($email)
{
    $emailConstraint = new Email();
    // all constraint "options" can be set this way
    $emailConstraint->message = 'Invalid email address';

    // use the validator to validate the value
```

```
    $errorList = $this->get('validator')->validateValue($email, $emailConstraint);

    if (count($errorList) == 0) {
        // this IS a valid email address, do something
    } else {
        // this is *not* a valid email address
        $errorMessage = $errorList[0]->getMessage()

        // do something with the error
    }

    // ...
}
```

By calling `validateValue` on the validator, you can pass in a raw value and the constraint object that you want to validate that value against. A full list of the available constraints - as well as the full class name for each constraint - is available in the *constraints reference* section .

The `validateValue` method returns a *`ConstraintViolationList`*[5] object, which acts just like an array of errors. Each error in the collection is a *`ConstraintViolation`*[6] object, which holds the error message on its *getMessage* method.

Final Thoughts

The Symfony2 `validator` is a powerful tool that can be leveraged to guarantee that the data of any object is "valid". The power behind validation lies in "constraints", which are rules that you can apply to properties or getter methods of your object. And while you'll most commonly use the validation framework indirectly when using forms, remember that it can be used anywhere to validate any object.

Learn more from the Cookbook

- *How to create a Custom Validation Constraint*

5. http://api.symfony.com/2.0/Symfony/Component/Validator/ConstraintViolationList.html
6. http://api.symfony.com/2.0/Symfony/Component/Validator/ConstraintViolation.html

Chapter 12
Forms

Dealing with HTML forms is one of the most common - and challenging - tasks for a web developer. Symfony2 integrates a Form component that makes dealing with forms easy. In this chapter, you'll build a complex form from the ground-up, learning the most important features of the form library along the way.

The Symfony form component is a standalone library that can be used outside of Symfony2 projects. For more information, see the *Symfony2 Form Component*[1] on Github.

Creating a Simple Form

Suppose you're building a simple todo list application that will need to display "tasks". Because your users will need to edit and create tasks, you're going to need to build a form. But before you begin, first focus on the generic `Task` class that represents and stores the data for a single task:

Listing 12-1

```
// 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;
```

1. https://github.com/symfony/Form

```
    }

    public function getDueDate()
    {
        return $this->dueDate;
    }
    public function setDueDate(\DateTime $dueDate = null)
    {
        $this->dueDate = $dueDate;
    }
}
```

If you're coding along with this example, create the `AcmeTaskBundle` first by running the following command (and accepting all of the default options):

Listing 12-2
```
php app/console generate:bundle --namespace=Acme/TaskBundle
```

This class is a "plain-old-PHP-object" because, so far, it has nothing to do with Symfony or any other library. It's quite simply a normal PHP object that directly solves a problem inside *your* application (i.e. the need to represent a task in your application). Of course, by the end of this chapter, you'll be able to submit data to a `Task` instance (via an HTML form), validate its data, and persist it to the database.

Building the Form

Now that you've created a `Task` class, the next step is to create and render the actual HTML form. In Symfony2, this is done by building a form object and then rendering it in a template. For now, this can all be done from inside a controller:

Listing 12-3
```
// src/Acme/TaskBundle/Controller/DefaultController.php
namespace Acme\TaskBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Acme\TaskBundle\Entity\Task;
use Symfony\Component\HttpFoundation\Request;

class DefaultController extends Controller
{
    public function newAction(Request $request)
    {
        // create a task and give it some dummy data for this example
        $task = new Task();
        $task->setTask('Write a blog post');
        $task->setDueDate(new \DateTime('tomorrow'));

        $form = $this->createFormBuilder($task)
            ->add('task', 'text')
            ->add('dueDate', 'date')
            ->getForm();

        return $this->render('AcmeTaskBundle:Default:new.html.twig', array(
            'form' => $form->createView(),
        ));
    }
}
```

This example shows you how to build your form directly in the controller. Later, in the "*Creating Form Classes*" section, you'll learn how to build your form in a standalone class, which is recommended as your form becomes reusable.

Creating a form requires relatively little code because Symfony2 form objects are built with a "form builder". The form builder's purpose is to allow you to write simple form "recipes", and have it do all the heavy-lifting of actually building the form.

In this example, you've added two fields to your form - `task` and `dueDate` - corresponding to the `task` and `dueDate` properties of the `Task` class. You've also assigned each a "type" (e.g. `text`, `date`), which, among other things, determines which HTML form tag(s) is rendered for that field.

Symfony2 comes with many built-in types that will be discussed shortly (see *Built-in Field Types*).

Rendering the Form

Now that the form has been created, the next step is to render it. This is done by passing a special form "view" object to your template (notice the `$form->createView()` in the controller above) and using a set of form helper functions:

Listing 12-4

```
{# 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>
```

This example assumes that you've created a route called `task_new` that points to the `AcmeTaskBundle:Default:new` controller that was created earlier.

That's it! By printing `form_widget(form)`, each field in the form is rendered, along with a label and error message (if there is one). As easy as this is, it's not very flexible (yet). Usually, you'll want to render each form field individually so you can control how the form looks. You'll learn how to do that in the "*Rendering a Form in a Template*" section.

Before moving on, notice how the rendered `task` input field has the value of the `task` property from the `$task` object (i.e. "Write a blog post"). This is the first job of a form: to take data from an object and translate it into a format that's suitable for being rendered in an HTML form.

The form system is smart enough to access the value of the protected `task` property via the `getTask()` and `setTask()` methods on the `Task` class. Unless a property is public, it *must* have a "getter" and "setter" method so that the form component can get and put data onto the property. For a Boolean property, you can use an "isser" method (e.g. `isPublished()`) instead of a getter (e.g. `getPublished()`).

Handling Form Submissions

The second job of a form is to translate user-submitted data back to the properties of an object. To make this happen, the submitted data from the user must be bound to the form. Add the following functionality to your controller:

Listing 12-5

```
// ...

public function newAction(Request $request)
{
    // just setup a fresh $task object (remove the dummy data)
    $task = new Task();

    $form = $this->createFormBuilder($task)
        ->add('task', 'text')
        ->add('dueDate', 'date')
        ->getForm();

    if ($request->getMethod() == 'POST') {
        $form->bindRequest($request);

        if ($form->isValid()) {
            // perform some action, such as saving the task to the database

            return $this->redirect($this->generateUrl('task_success'));
        }
    }

    // ...
}
```

Now, when submitting the form, the controller binds the submitted data to the form, which translates that data back to the `task` and `dueDate` properties of the `$task` object. This all happens via the `bindRequest()` method.

As soon as `bindRequest()` is called, the submitted data is transferred to the underlying object immediately. This happens regardless of whether or not the underlying data is actually valid.

This controller follows a common pattern for handling forms, and has three possible paths:

1. When initially loading the page in a browser, the request method is `GET` and the form is simply created and rendered;
2. When the user submits the form (i.e. the method is `POST`) with invalid data (validation is covered in the next section), the form is bound and then rendered, this time displaying all validation errors;
3. When the user submits the form with valid data, the form is bound and you have the opportunity to perform some actions using the `$task` object (e.g. persisting it to the database) before redirecting the user to some other page (e.g. a "thank you" or "success" page).

Redirecting a user after a successful form submission prevents the user from being able to hit "refresh" and re-post the data.

Form Validation

In the previous section, you learned how a form can be submitted with valid or invalid data. In Symfony2, validation is applied to the underlying object (e.g. `Task`). In other words, the question isn't whether the "form" is valid, but whether or not the `$task` object is valid after the form has applied the submitted data to it. Calling `$form->isValid()` is a shortcut that asks the `$task` object whether or not it has valid data.

Validation is done by adding a set of rules (called constraints) to a class. To see this in action, add validation constraints so that the `task` field cannot be empty and the `dueDate` field cannot be empty and must be a valid DateTime object.

Listing 12-6

```
# Acme/TaskBundle/Resources/config/validation.yml
Acme\TaskBundle\Entity\Task:
    properties:
        task:
            - NotBlank: ~
        dueDate:
            - NotBlank: ~
            - Type: \DateTime
```

That's it! If you re-submit the form with invalid data, you'll see the corresponding errors printed out with the form.

HTML5 Validation

As of HTML5, many browsers can natively enforce certain validation constraints on the client side. The most common validation is activated by rendering a `required` attribute on fields that are required. For browsers that support HTML5, this will result in a native browser message being displayed if the user tries to submit the form with that field blank.

Generated forms take full advantage of this new feature by adding sensible HTML attributes that trigger the validation. The client-side validation, however, can be disabled by adding the `novalidate` attribute to the `form` tag or `formnovalidate` to the submit tag. This is especially useful when you want to test your server-side validation constraints, but are being prevented by your browser from, for example, submitting blank fields.

Validation is a very powerful feature of Symfony2 and has its own *dedicated chapter*.

Validation Groups

If you're not using *validation groups*, then you can skip this section.

If your object takes advantage of *validation groups*, you'll need to specify which validation group(s) your form should use:

Listing 12-7

```
$form = $this->createFormBuilder($users, array(
    'validation_groups' => array('registration'),
```

```
))->add(...)
;
```

If you're creating *form classes* (a good practice), then you'll need to add the following to the `getDefaultOptions()` method:

Listing 12-8
```
public function getDefaultOptions(array $options)
{
    return array(
        'validation_groups' => array('registration')
    );
}
```

In both of these cases, *only* the `registration` validation group will be used to validate the underlying object.

Built-in Field Types

Symfony comes standard with a large group of field types that cover all of the common form fields and data types you'll encounter:

Text Fields

- *text*
- *textarea*
- *email*
- *integer*
- *money*
- *number*
- *password*
- *percent*
- *search*
- *url*

Choice Fields

- *choice*
- *entity*
- *country*
- *language*
- *locale*
- *timezone*

Date and Time Fields

- *date*
- *datetime*
- *time*
- *birthday*

Other Fields

- *checkbox*
- *file*

- *radio*

Field Groups

- *collection*
- *repeated*

Hidden Fields

- *hidden*
- *csrf*

Base Fields

- *field*
- *form*

You can also create your own custom field types. This topic is covered in the "*How to Create a Custom Form Field Type*" article of the cookbook.

Field Type Options

Each field type has a number of options that can be used to configure it. For example, the `dueDate` field is currently being rendered as 3 select boxes. However, the *date field* can be configured to be rendered as a single text box (where the user would enter the date as a string in the box):

```
->add('dueDate', 'date', array('widget' => 'single_text'))
```

Listing 12-9

Each field type has a number of different options that can be passed to it. Many of these are specific to the field type and details can be found in the documentation for each type.

The `required` option

The most common option is the `required` option, which can be applied to any field. By default, the `required` option is set to `true`, meaning that HTML5-ready browsers will apply client-side validation if the field is left blank. If you don't want this behavior, either set the `required` option on your field to `false` or *disable HTML5 validation*.

Also note that setting the `required` option to `true` will **not** result in server-side validation to be applied. In other words, if a user submits a blank value for the field (either with an old browser or web service, for example), it will be accepted as a valid value unless you use Symfony's `NotBlank` or `NotNull` validation constraint.

In other words, the `required` option is "nice", but true server-side validation should *always* be used.

The `label` option

The label for the form field can be set using the `label` option, which can be applied to any field:

Listing 12-10
```
->add('dueDate', 'date', array(
    'widget' => 'single_text',
    'label'  => 'Due Date',
))
```

The label for a field can also be set in the template rendering the form, see below.

Field Type Guessing

Now that you've added validation metadata to the `Task` class, Symfony already knows a bit about your fields. If you allow it, Symfony can "guess" the type of your field and set it up for you. In this example, Symfony can guess from the validation rules that both the `task` field is a normal `text` field and the `dueDate` field is a `date` field:

Listing 12-11
```
public function newAction()
{
    $task = new Task();

    $form = $this->createFormBuilder($task)
        ->add('task')
        ->add('dueDate', null, array('widget' => 'single_text'))
        ->getForm();
}
```

The "guessing" is activated when you omit the second argument to the `add()` method (or if you pass `null` to it). If you pass an options array as the third argument (done for `dueDate` above), these options are applied to the guessed field.

If your form uses a specific validation group, the field type guesser will still consider *all* validation constraints when guessing your field types (including constraints that are not part of the validation group(s) being used).

Field Type Options Guessing

In addition to guessing the "type" for a field, Symfony can also try to guess the correct values of a number of field options.

When these options are set, the field will be rendered with special HTML attributes that provide for HTML5 client-side validation. However, it doesn't generate the equivalent server-side constraints (e.g. `Assert\MaxLength`). And though you'll need to manually add your server-side validation, these field type options can then be guessed from that information.

- `required`: The `required` option can be guessed based off of the validation rules (i.e. is the field `NotBlank` or `NotNull`) or the Doctrine metadata (i.e. is the field `nullable`). This is very useful, as your client-side validation will automatically match your validation rules.

- `max_length`: If the field is some sort of text field, then the `max_length` option can be guessed from the validation constraints (if `MaxLength` or `Max` is used) or from the Doctrine metadata (via the field's length).

These field options are *only* guessed if you're using Symfony to guess the field type (i.e. omit or pass `null` as the second argument to `add()`).

If you'd like to change one of the guessed values, you can override it by passing the option in the options field array:

```
->add('task', null, array('max_length' => 4))
```

Listing 12-12

Rendering a Form in a Template

So far, you've seen how an entire form can be rendered with just one line of code. Of course, you'll usually need much more flexibility when rendering:

```
{# 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>
```

Listing 12-13

Let's take a look at each part:

- `form_enctype(form)` - If at least one field is a file upload field, this renders the obligatory `enctype="multipart/form-data"`;
- `form_errors(form)` - Renders any errors global to the whole form (field-specific errors are displayed next to each field);
- `form_row(form.dueDate)` - Renders the label, any errors, and the HTML form widget for the given field (e.g. `dueDate`) inside, by default, a `div` element;
- `form_rest(form)` - Renders any fields that have not yet been rendered. It's usually a good idea to place a call to this helper at the bottom of each form (in case you forgot to output a field or don't want to bother manually rendering hidden fields). This helper is also useful for taking advantage of the automatic *CSRF Protection*.

The majority of the work is done by the `form_row` helper, which renders the label, errors and HTML form widget of each field inside a `div` tag by default. In the *Form Theming* section, you'll learn how the `form_row` output can be customized on many different levels.

You can access the current data of your form via `form.vars.value`:

```
{{ form.vars.value.task }}
```

Listing 12-14

Rendering each Field by Hand

The `form_row` helper is great because you can very quickly render each field of your form (and the markup used for the "row" can be customized as well). But since life isn't always so simple, you can also render each field entirely by hand. The end-product of the following is the same as when you used the `form_row` helper:

Listing 12-15
```
{{ 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) }}
```

If the auto-generated label for a field isn't quite right, you can explicitly specify it:

Listing 12-16
```
{{ form_label(form.task, 'Task Description') }}
```

Some field types have additional rendering options that can be passed to the widget. These options are documented with each type, but one common options is `attr`, which allows you to modify attributes on the form element. The following would add the `task_field` class to the rendered input text field:

Listing 12-17
```
{{ form_widget(form.task, { 'attr': {'class': 'task_field'} }) }}
```

If you need to render form fields "by hand" then you can access individual values for fields such as the `id`, `name` and `label`. For example to get the `id`:

Listing 12-18
```
{{ form.task.vars.id }}
```

To get the value used for the form field's name attribute you need to use the `full_name` value:

Listing 12-19
```
{{ form.task.vars.full_name }}
```

Twig Template Function Reference

If you're using Twig, a full reference of the form rendering functions is available in the *reference manual*. Read this to know everything about the helpers available and the options that can be used with each.

Creating Form Classes

As you've seen, a form can be created and used directly in a controller. However, a better practice is to build the form in a separate, standalone PHP class, which can then be reused anywhere in your application. Create a new class that will house the logic for building the task form:

Listing 12-20
```
// src/Acme/TaskBundle/Form/Type/TaskType.php

namespace Acme\TaskBundle\Form\Type;
```

```php
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';
    }
}
```

This new class contains all the directions needed to create the task form (note that the `getName()` method should return a unique identifier for this form "type"). It can be used to quickly build a form object in the controller:

Listing 12-21

```php
// src/Acme/TaskBundle/Controller/DefaultController.php

// add this new use statement at the top of the class
use Acme\TaskBundle\Form\Type\TaskType;

public function newAction()
{
    $task = // ...
    $form = $this->createForm(new TaskType(), $task);

    // ...
}
```

Placing the form logic into its own class means that the form can be easily reused elsewhere in your project. This is the best way to create forms, but the choice is ultimately up to you.

Setting the `data_class`

Every form needs to know the name of the class that holds the underlying data (e.g. `Acme\TaskBundle\Entity\Task`). Usually, this is just guessed based off of the object passed to the second argument to `createForm` (i.e. `$task`). Later, when you begin embedding forms, this will no longer be sufficient. So, while not always necessary, it's generally a good idea to explicitly specify the `data_class` option by adding the following to your form type class:

Listing 12-22

```php
public function getDefaultOptions(array $options)
{
    return array(
        'data_class' => 'Acme\TaskBundle\Entity\Task',
    );
}
```

When mapping forms to objects, all fields are mapped. Any fields on the form that do not exist on the mapped object will cause an exception to be thrown.

In cases where you need extra fields in the form (for example: a "do you agree with these terms" checkbox) that will not be mapped to the underlying object, you need to set the property_path option to `false`:

Listing 12-23
```
public function buildForm(FormBuilder $builder, array $options)
{
    $builder->add('task');
    $builder->add('dueDate', null, array('property_path' => false));
}
```

Additionally, if there are any fields on the form that aren't included in the submitted data, those fields will be explicitly set to `null`.

Forms and Doctrine

The goal of a form is to translate data from an object (e.g. `Task`) to an HTML form and then translate user-submitted data back to the original object. As such, the topic of persisting the `Task` object to the database is entirely unrelated to the topic of forms. But, if you've configured the `Task` class to be persisted via Doctrine (i.e. you've added *mapping metadata* for it), then persisting it after a form submission can be done when the form is valid:

Listing 12-24
```
if ($form->isValid()) {
    $em = $this->getDoctrine()->getEntityManager();
    $em->persist($task);
    $em->flush();

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

If, for some reason, you don't have access to your original `$task` object, you can fetch it from the form:

Listing 12-25
```
$task = $form->getData();
```

For more information, see the *Doctrine ORM chapter*.

The key thing to understand is that when the form is bound, the submitted data is transferred to the underlying object immediately. If you want to persist that data, you simply need to persist the object itself (which already contains the submitted data).

Embedded Forms

Often, you'll want to build a form that will include fields from many different objects. For example, a registration form may contain data belonging to a `User` object as well as many `Address` objects. Fortunately, this is easy and natural with the form component.

Embedding a Single Object

Suppose that each `Task` belongs to a simple `Category` object. Start, of course, by creating the `Category` object:

Listing 12-26
```
// src/Acme/TaskBundle/Entity/Category.php
namespace Acme\TaskBundle\Entity;

use Symfony\Component\Validator\Constraints as Assert;
```

```
class Category
{
    /**
     * @Assert\NotBlank()
     */
    public $name;
}
```

Next, add a new `category` property to the `Task` class:

Listing 12-27

```
// ...

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;
    }
}
```

Now that your application has been updated to reflect the new requirements, create a form class so that a `Category` object can be modified by the user:

Listing 12-28

```
// 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';
```

```
    }
}
```

The end goal is to allow the `Category` of a `Task` to be modified right inside the task form itself. To accomplish this, add a `category` field to the `TaskType` object whose type is an instance of the new `CategoryType` class:

Listing 12-29

```
public function buildForm(FormBuilder $builder, array $options)
{
    // ...

    $builder->add('category', new CategoryType());
}
```

The fields from `CategoryType` can now be rendered alongside those from the `TaskType` class. Render the `Category` fields in the same way as the original `Task` fields:

Listing 12-30

```
{# ... #}

<h3>Category</h3>
<div class="category">
    {{ form_row(form.category.name) }}
</div>

{{ form_rest(form) }}
{# ... #}
```

When the user submits the form, the submitted data for the `Category` fields are used to construct an instance of `Category`, which is then set on the `category` field of the `Task` instance.

The `Category` instance is accessible naturally via `$task->getCategory()` and can be persisted to the database or used however you need.

Embedding a Collection of Forms

You can also embed a collection of forms into one form (imagine a `Category` form with many `Product` sub-forms). This is done by using the `collection` field type.

For more information see the "*How to Embed a Collection of Forms*" cookbook entry and the *collection* field type reference.

Form Theming

Every part of how a form is rendered can be customized. You're free to change how each form "row" renders, change the markup used to render errors, or even customize how a `textarea` tag should be rendered. Nothing is off-limits, and different customizations can be used in different places.

Symfony uses templates to render each and every part of a form, such as `label` tags, `input` tags, error messages and everything else.

In Twig, each form "fragment" is represented by a Twig block. To customize any part of how a form renders, you just need to override the appropriate block.

In PHP, each form "fragment" is rendered via an individual template file. To customize any part of how a form renders, you just need to override the existing template by creating a new one.

To understand how this works, let's customize the `form_row` fragment and add a class attribute to the `div` element that surrounds each row. To do this, create a new template file that will store the new markup:

Listing 12-31

```
{# 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 %}
```

The `field_row` form fragment is used when rendering most fields via the `form_row` function. To tell the form component to use your new `field_row` fragment defined above, add the following to the top of the template that renders the form:

Listing 12-32

```
{# src/Acme/TaskBundle/Resources/views/Default/new.html.twig #}

{% form_theme form 'AcmeTaskBundle:Form:fields.html.twig' %}

{% form_theme form 'AcmeTaskBundle:Form:fields.html.twig'
'AcmeTaskBundle:Form:fields2.html.twig' %}

<form ...>
```

The `form_theme` tag (in Twig) "imports" the fragments defined in the given template and uses them when rendering the form. In other words, when the `form_row` function is called later in this template, it will use the `field_row` block from your custom theme (instead of the default `field_row` block that ships with Symfony).

Your custom theme does not have to override all the blocks. When rendering a block which is not overridden in your custom theme, the theming engine will fall back to the global theme (defined at the bundle level).

If several custom themes are provided they will be searched in the listed order before falling back to the global theme.

To customize any portion of a form, you just need to override the appropriate fragment. Knowing exactly which block or file to override is the subject of the next section.

For a more extensive discussion, see *How to customize Form Rendering*.

Form Fragment Naming

In Symfony, every part of a form that is rendered - HTML form elements, errors, labels, etc - is defined in a base theme, which is a collection of blocks in Twig and a collection of template files in PHP.

In Twig, every block needed is defined in a single template file (*form_div_layout.html.twig*[2]) that lives inside the *Twig Bridge*[3]. Inside this file, you can see every block needed to render a form and every default field type.

In PHP, the fragments are individual template files. By default they are located in the *Resources/views/Form* directory of the framework bundle (*view on GitHub*[4]).

Each fragment name follows the same basic pattern and is broken up into two pieces, separated by a single underscore character (_). A few examples are:

- `field_row` - used by `form_row` to render most fields;

2. https://github.com/symfony/symfony/blob/master/src/Symfony/Bridge/Twig/Resources/views/Form/form_div_layout.html.twig
3. https://github.com/symfony/symfony/tree/master/src/Symfony/Bridge/Twig
4. https://github.com/symfony/symfony/tree/master/src/Symfony/Bundle/FrameworkBundle/Resources/views/Form

- `textarea_widget` - used by `form_widget` to render a `textarea` field type;
- `field_errors` - used by `form_errors` to render errors for a field;

Each fragment follows the same basic pattern: `type_part`. The `type` portion corresponds to the field *type* being rendered (e.g. `textarea`, `checkbox`, `date`, etc) whereas the `part` portion corresponds to *what* is being rendered (e.g. `label`, `widget`, `errors`, etc). By default, there are 4 possible *parts* of a form that can be rendered:

`label`	(e.g. `field_label`)	renders the field's label
`widget`	(e.g. `field_widget`)	renders the field's HTML representation
`errors`	(e.g. `field_errors`)	renders the field's errors
`row`	(e.g. `field_row`)	renders the field's entire row (label, widget & errors)

There are actually 3 other *parts* - `rows`, `rest`, and `enctype` - but you should rarely if ever need to worry about overriding them.

By knowing the field type (e.g. `textarea`) and which part you want to customize (e.g. `widget`), you can construct the fragment name that needs to be overridden (e.g. `textarea_widget`).

Template Fragment Inheritance

In some cases, the fragment you want to customize will appear to be missing. For example, there is no `textarea_errors` fragment in the default themes provided with Symfony. So how are the errors for a textarea field rendered?

The answer is: via the `field_errors` fragment. When Symfony renders the errors for a textarea type, it looks first for a `textarea_errors` fragment before falling back to the `field_errors` fragment. Each field type has a *parent* type (the parent type of `textarea` is `field`), and Symfony uses the fragment for the parent type if the base fragment doesn't exist.

So, to override the errors for *only* `textarea` fields, copy the `field_errors` fragment, rename it to `textarea_errors` and customize it. To override the default error rendering for *all* fields, copy and customize the `field_errors` fragment directly.

The "parent" type of each field type is available in the *form type reference* for each field type.

Global Form Theming

In the above example, you used the `form_theme` helper (in Twig) to "import" the custom form fragments into *just* that form. You can also tell Symfony to import form customizations across your entire project.

Twig

To automatically include the customized blocks from the `fields.html.twig` template created earlier in *all* templates, modify your application configuration file:

Listing 12-33

```
# app/config/config.yml

twig:
    form:
```

```
        resources:
            - 'AcmeTaskBundle:Form:fields.html.twig'
    # ...
```

Any blocks inside the `fields.html.twig` template are now used globally to define form output.

Customizing Form Output all in a Single File with Twig

In Twig, you can also customize a form block right inside the template where that customization is needed:

Listing 12-34

```
{% 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 %}
```

The `{% form_theme form _self %}` tag allows form blocks to be customized directly inside the template that will use those customizations. Use this method to quickly make form output customizations that will only ever be needed in a single template.

PHP

To automatically include the customized templates from the `Acme/TaskBundle/Resources/views/Form` directory created earlier in *all* templates, modify your application configuration file:

Listing 12-35

```
# app/config/config.yml

framework:
    templating:
        form:
            resources:
                - 'AcmeTaskBundle:Form'
# ...
```

Any fragments inside the `Acme/TaskBundle/Resources/views/Form` directory are now used globally to define form output.

CSRF Protection

CSRF - or *Cross-site request forgery*[5] - is a method by which a malicious user attempts to make your legitimate users unknowingly submit data that they don't intend to submit. Fortunately, CSRF attacks can be prevented by using a CSRF token inside your forms.

5. http://en.wikipedia.org/wiki/Cross-site_request_forgery

The good news is that, by default, Symfony embeds and validates CSRF tokens automatically for you. This means that you can take advantage of the CSRF protection without doing anything. In fact, every form in this chapter has taken advantage of the CSRF protection!

CSRF protection works by adding a hidden field to your form - called `_token` by default - that contains a value that only you and your user knows. This ensures that the user - not some other entity - is submitting the given data. Symfony automatically validates the presence and accuracy of this token.

The `_token` field is a hidden field and will be automatically rendered if you include the `form_rest()` function in your template, which ensures that all un-rendered fields are output.

The CSRF token can be customized on a form-by-form basis. For example:

Listing 12-36
```
class TaskType extends AbstractType
{
    // ...

    public function getDefaultOptions(array $options)
    {
        return array(
            'data_class'      => 'Acme\TaskBundle\Entity\Task',
            'csrf_protection' => true,
            'csrf_field_name' => '_token',
            // a unique key to help generate the secret token
            'intention'       => 'task_item',
        );
    }

    // ...
}
```

To disable CSRF protection, set the `csrf_protection` option to false. Customizations can also be made globally in your project. For more information, see the *form configuration reference* section.

The `intention` option is optional but greatly enhances the security of the generated token by making it different for each form.

Using a Form without a Class

In most cases, a form is tied to an object, and the fields of the form get and store their data on the properties of that object. This is exactly what you've seen so far in this chapter with the *Task* class.

But sometimes, you may just want to use a form without a class, and get back an array of the submitted data. This is actually really easy:

Listing 12-37
```
// make sure you've imported the Request namespace above the class
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
}
```

By default, a form actually assumes that you want to work with arrays of data, instead of an object. There are exactly two ways that you can change this behavior and tie the form to an object instead:

1. Pass an object when creating the form (as the first argument to `createFormBuilder` or the second argument to `createForm`);
2. Declare the `data_class` option on your form.

If you *don't* do either of these, then the form will return the data as an array. In this example, since `$defaultData` is not an object (and no `data_class` option is set), `$form->getData()` ultimately returns an array.

You can also access POST values (in this case "name") directly through the request object, like so:

```
$this->get('request')->request->get('name');
```

Listing 12-38

Be advised, however, that in most cases using the getData() method is a better choice, since it returns the data (usually an object) after it's been transformed by the form framework.

Adding Validation

The only missing piece is validation. Usually, when you call `$form->isValid()`, the object is validated by reading the constraints that you applied to that class. But without a class, how can you add constraints to the data of your form?

The answer is to setup the constraints yourself, and pass them into your form. The overall approach is covered a bit more in the *validation chapter*, but here's a short example:

```
// import the namespaces above your controller class
use Symfony\Component\Validator\Constraints\Email;
use Symfony\Component\Validator\Constraints\MinLength;
use Symfony\Component\Validator\Constraints\Collection;

$collectionConstraint = new Collection(array(
    'name' => new MinLength(5),
    'email' => new Email(array('message' => 'Invalid email address')),
));

// create a form, no default values, pass in the constraint option
$form = $this->createFormBuilder(null, array(
    'validation_constraint' => $collectionConstraint,
))->add('email', 'email')
    // ...
;
```

Listing 12-39

Now, when you call *$form->bindRequest($request)*, the constraints setup here are run against your form's data. If you're using a form class, override the `getDefaultOptions` method to specify the option:

```
namespace Acme\TaskBundle\Form\Type;
```

Listing 12-40

```php
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')),
        ));

        return array('validation_constraint' => $collectionConstraint);
    }
}
```

Now, you have the flexibility to create forms - with validation - that return an array of data, instead of an object. In most cases, it's better - and certainly more robust - to bind your form to an object. But for simple forms, this is a great approach.

Final Thoughts

You now know all of the building blocks necessary to build complex and functional forms for your application. When building forms, keep in mind that the first goal of a form is to translate data from an object (`Task`) to an HTML form so that the user can modify that data. The second goal of a form is to take the data submitted by the user and to re-apply it to the object.

There's still much more to learn about the powerful world of forms, such as how to handle *file uploads with Doctrine* or how to create a form where a dynamic number of sub-forms can be added (e.g. a todo list where you can keep adding more fields via Javascript before submitting). See the cookbook for these topics. Also, be sure to lean on the *field type reference documentation*, which includes examples of how to use each field type and its options.

Learn more from the Cookbook

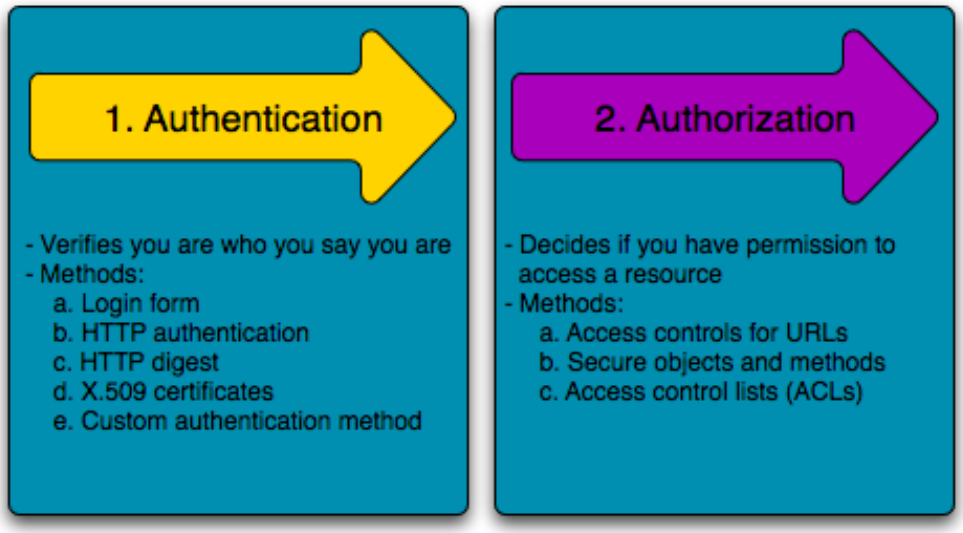
- *How to handle File Uploads with Doctrine*
- *File Field Reference*
- *Creating Custom Field Types*
- *How to customize Form Rendering*
- *How to Dynamically Generate Forms Using Form Events*
- *Using Data Transformers*

Chapter 13
Security

Security is a two-step process whose goal is to prevent a user from accessing a resource that he/she should not have access to.

In the first step of the process, the security system identifies who the user is by requiring the user to submit some sort of identification. This is called **authentication**, and it means that the system is trying to find out who you are.

Once the system knows who you are, the next step is to determine if you should have access to a given resource. This part of the process is called **authorization**, and it means that the system is checking to see if you have privileges to perform a certain action.



Since the best way to learn is to see an example, let's dive right in.

Symfony's *security component*[1] is available as a standalone PHP library for use inside any PHP project.

Basic Example: HTTP Authentication

The security component can be configured via your application configuration. In fact, most standard security setups are just a matter of using the right configuration. The following configuration tells Symfony to secure any URL matching `/admin/*` and to ask the user for credentials using basic HTTP authentication (i.e. the old-school username/password box):

Listing 13-1

```
# 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
```

A standard Symfony distribution separates the security configuration into a separate file (e.g. `app/config/security.yml`). If you don't have a separate security file, you can put the configuration directly into your main config file (e.g. `app/config/config.yml`).

The end result of this configuration is a fully-functional security system that looks like the following:

- There are two users in the system (`ryan` and `admin`);
- Users authenticate themselves via the basic HTTP authentication prompt;
- Any URL matching `/admin/*` is secured, and only the `admin` user can access it;
- All URLs *not* matching `/admin/*` are accessible by all users (and the user is never prompted to login).

Let's look briefly at how security works and how each part of the configuration comes into play.

How Security Works: Authentication and Authorization

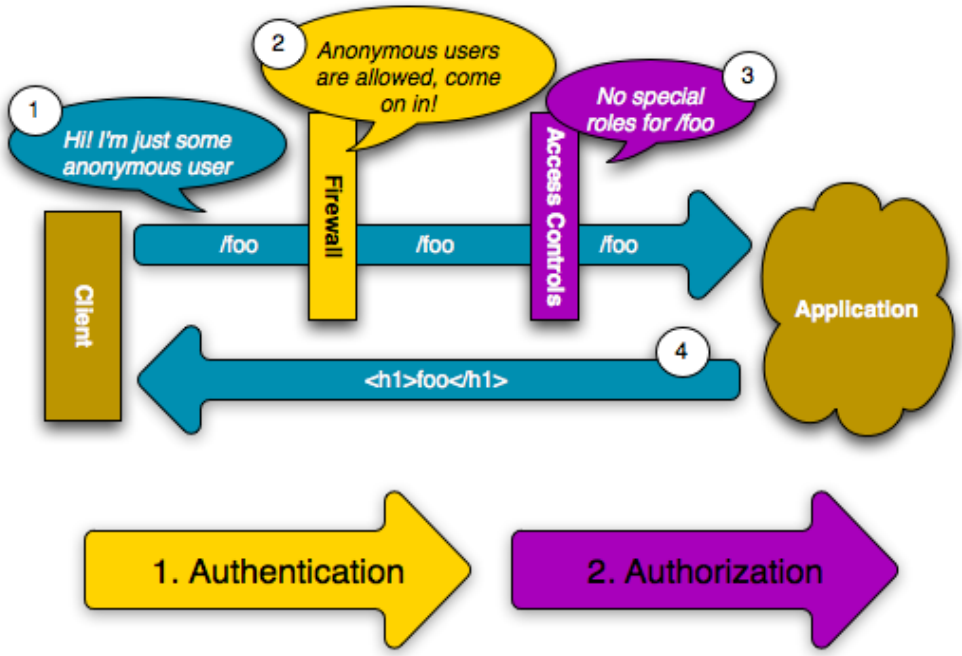
Symfony's security system works by determining who a user is (i.e. authentication) and then checking to see if that user should have access to a specific resource or URL.

Firewalls (Authentication)

When a user makes a request to a URL that's protected by a firewall, the security system is activated. The job of the firewall is to determine whether or not the user needs to be authenticated, and if he does, to send a response back to the user initiating the authentication process.

1. `https://github.com/symfony/Security`

A firewall is activated when the URL of an incoming request matches the configured firewall's regular expression `pattern` config value. In this example, the `pattern` (`^/`) will match *every* incoming request. The fact that the firewall is activated does *not* mean, however, that the HTTP authentication username and password box is displayed for every URL. For example, any user can access `/foo` without being prompted to authenticate.

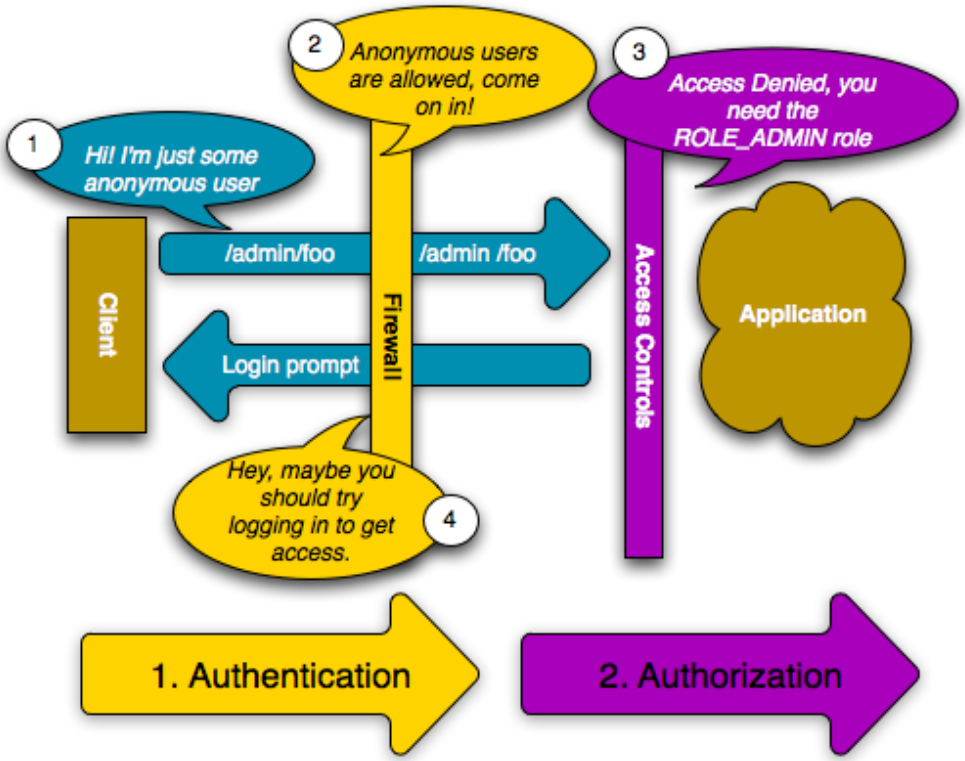


This works first because the firewall allows *anonymous users* via the `anonymous` configuration parameter. In other words, the firewall doesn't require the user to fully authenticate immediately. And because no special `role` is needed to access `/foo` (under the `access_control` section), the request can be fulfilled without ever asking the user to authenticate.

If you remove the `anonymous` key, the firewall will *always* make a user fully authenticate immediately.

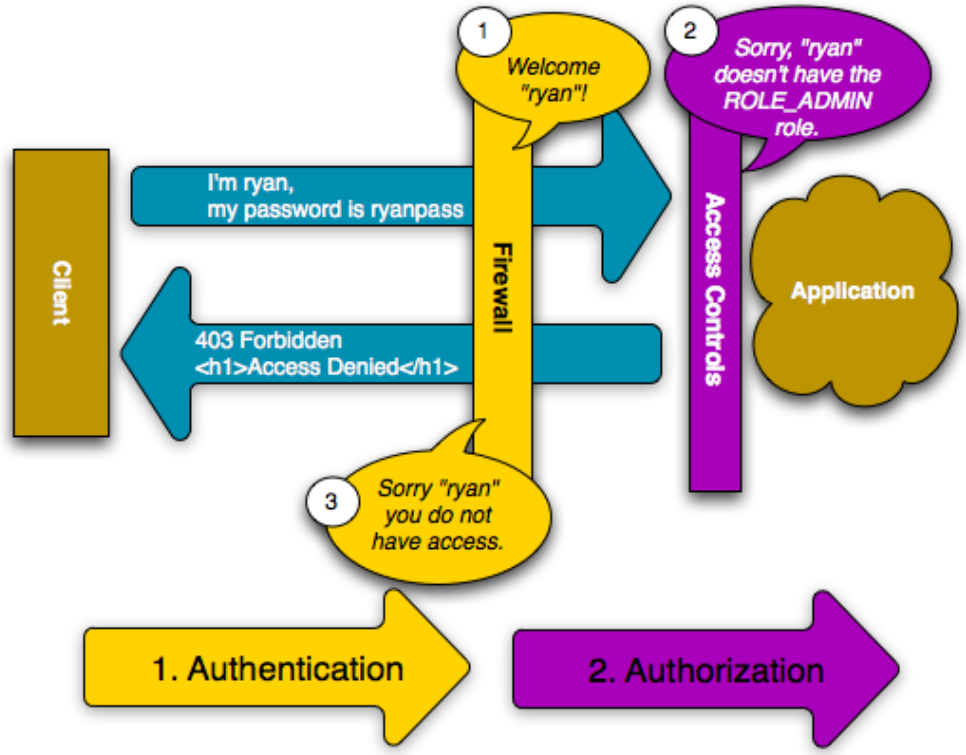
Access Controls (Authorization)

If a user requests `/admin/foo`, however, the process behaves differently. This is because of the `access_control` configuration section that says that any URL matching the regular expression pattern `^/admin` (i.e. `/admin` or anything matching `/admin/*`) requires the `ROLE_ADMIN` role. Roles are the basis for most authorization: a user can access `/admin/foo` only if it has the `ROLE_ADMIN` role.



Like before, when the user originally makes the request, the firewall doesn't ask for any identification. However, as soon as the access control layer denies the user access (because the anonymous user doesn't have the `ROLE_ADMIN` role), the firewall jumps into action and initiates the authentication process. The authentication process depends on the authentication mechanism you're using. For example, if you're using the form login authentication method, the user will be redirected to the login page. If you're using HTTP authentication, the user will be sent an HTTP 401 response so that the user sees the username and password box.

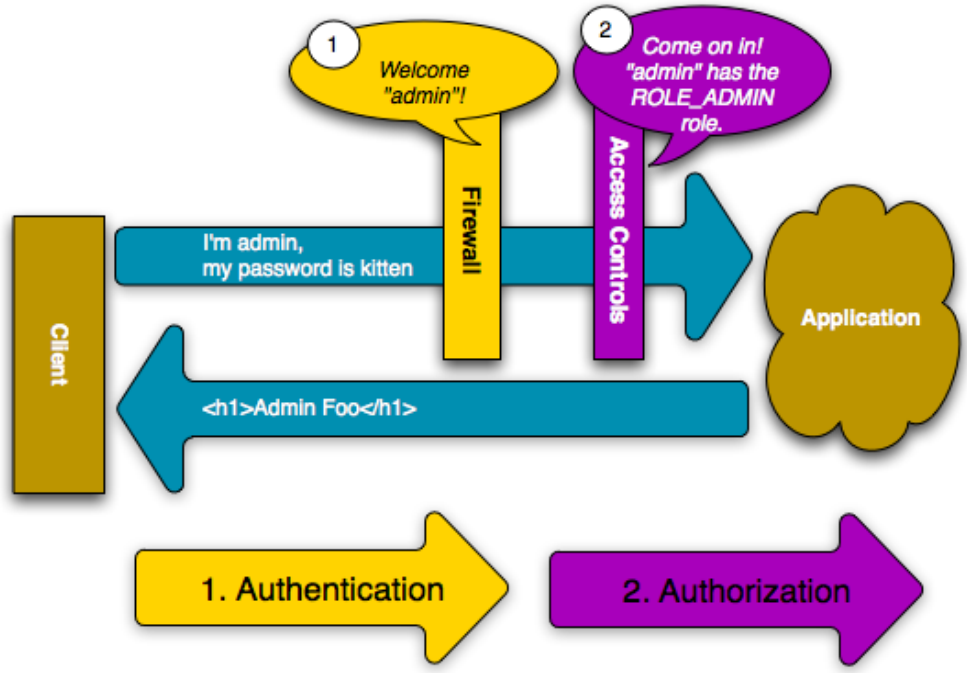
The user now has the opportunity to submit its credentials back to the application. If the credentials are valid, the original request can be re-tried.



In this example, the user `ryan` successfully authenticates with the firewall. But since `ryan` doesn't have the `ROLE_ADMIN` role, he's still denied access to `/admin/foo`. Ultimately, this means that the user will see some sort of message indicating that access has been denied.

When Symfony denies the user access, the user sees an error screen and receives a 403 HTTP status code (`Forbidden`). You can customize the access denied error screen by following the directions in the *Error Pages* cookbook entry to customize the 403 error page.

Finally, if the `admin` user requests `/admin/foo`, a similar process takes place, except now, after being authenticated, the access control layer will let the request pass through:



The request flow when a user requests a protected resource is straightforward, but incredibly flexible. As you'll see later, authentication can be handled in any number of ways, including via a form login, X.509 certificate, or by authenticating the user via Twitter. Regardless of the authentication method, the request flow is always the same:

1. A user accesses a protected resource;
2. The application redirects the user to the login form;
3. The user submits its credentials (e.g. username/password);
4. The firewall authenticates the user;
5. The authenticated user re-tries the original request.

The *exact* process actually depends a little bit on which authentication mechanism you're using. For example, when using form login, the user submits its credentials to one URL that processes the form (e.g. `/login_check`) and then is redirected back to the originally requested URL (e.g. `/admin/foo`). But with HTTP authentication, the user submits its credentials directly to the original URL (e.g. `/admin/foo`) and then the page is returned to the user in that same request (i.e. no redirect).

These types of idiosyncrasies shouldn't cause you any problems, but they're good to keep in mind.

You'll also learn later how *anything* can be secured in Symfony2, including specific controllers, objects, or even PHP methods.

Using a Traditional Login Form

In this section, you'll learn how to create a basic login form that continues to use the hard-coded users that are defined in the `security.yml` file.

> To load users from the database, please read *How to load Security Users from the Database (the Entity Provider)*. By reading that article and this section, you can create a full login form system that loads users from the database.

So far, you've seen how to blanket your application beneath a firewall and then protect access to certain areas with roles. By using HTTP Authentication, you can effortlessly tap into the native username/password box offered by all browsers. However, Symfony supports many authentication mechanisms out of the box. For details on all of them, see the *Security Configuration Reference*.

In this section, you'll enhance this process by allowing the user to authenticate via a traditional HTML login form.

First, enable form login under your firewall:

Listing 13-2

```yaml
# app/config/security.yml
security:
    firewalls:
        secured_area:
            pattern:    ^/
            anonymous: ~
            form_login:
                login_path:  /login
                check_path:  /login_check
```

If you don't need to customize your `login_path` or `check_path` values (the values used here are the default values), you can shorten your configuration:

Listing 13-3

```yaml
form_login: ~
```

Now, when the security system initiates the authentication process, it will redirect the user to the login form (`/login` by default). Implementing this login form visually is your job. First, create two routes: one that will display the login form (i.e. `/login`) and one that will handle the login form submission (i.e. `/login_check`):

Listing 13-4

```yaml
# app/config/routing.yml
login:
    pattern:   /login
    defaults:  { _controller: AcmeSecurityBundle:Security:login }
login_check:
    pattern:   /login_check
```

You will *not* need to implement a controller for the `/login_check` URL as the firewall will automatically catch and process any form submitted to this URL. It's optional, but helpful, to create a route so that you can use it to generate the form submission URL in the login template below.

Notice that the name of the `login` route isn't important. What's important is that the URL of the route (`/login`) matches the `login_path` config value, as that's where the security system will redirect users that need to login.

Next, create the controller that will display the login form:

Listing 13-5

```php
// src/Acme/SecurityBundle/Controller/SecurityController.php;
namespace Acme\SecurityBundle\Controller;
```

```php
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\Security\Core\SecurityContext;

class SecurityController extends Controller
{
    public function loginAction()
    {
        $request = $this->getRequest();
        $session = $request->getSession();

        // get the login error if there is one
        if ($request->attributes->has(SecurityContext::AUTHENTICATION_ERROR)) {
            $error = $request->attributes->get(SecurityContext::AUTHENTICATION_ERROR);
        } else {
            $error = $session->get(SecurityContext::AUTHENTICATION_ERROR);
            $session->remove(SecurityContext::AUTHENTICATION_ERROR);
        }

        return $this->render('AcmeSecurityBundle:Security:login.html.twig', array(
            // last username entered by the user
            'last_username' => $session->get(SecurityContext::LAST_USERNAME),
            'error'         => $error,
        ));
    }
}
```

Don't let this controller confuse you. As you'll see in a moment, when the user submits the form, the security system automatically handles the form submission for you. If the user had submitted an invalid username or password, this controller reads the form submission error from the security system so that it can be displayed back to the user.

In other words, your job is to display the login form and any login errors that may have occurred, but the security system itself takes care of checking the submitted username and password and authenticating the user.

Finally, create the corresponding template:

Listing 13-6

```twig
{# src/Acme/SecurityBundle/Resources/views/Security/login.html.twig #}
{% if error %}
    <div>{{ error.message }}</div>
{% endif %}

<form action="{{ path('login_check') }}" method="post">
    <label for="username">Username:</label>
    <input type="text" id="username" name="_username" value="{{ last_username }}" />

    <label for="password">Password:</label>
    <input type="password" id="password" name="_password" />

    {#
        If you want to control the URL the user is redirected to on success (more details
below)
        <input type="hidden" name="_target_path" value="/account" />
    #}

    <button type="submit">login</button>
</form>
```

The `error` variable passed into the template is an instance of *`AuthenticationException`*[2]. It may contain more information - or even sensitive information - about the authentication failure, so use it wisely!

The form has very few requirements. First, by submitting the form to `/login_check` (via the `login_check` route), the security system will intercept the form submission and process the form for you automatically. Second, the security system expects the submitted fields to be called `_username` and `_password` (these field names can be *configured*).

And that's it! When you submit the form, the security system will automatically check the user's credentials and either authenticate the user or send the user back to the login form where the error can be displayed.

Let's review the whole process:

1. The user tries to access a resource that is protected;
2. The firewall initiates the authentication process by redirecting the user to the login form (`/login`);
3. The `/login` page renders login form via the route and controller created in this example;
4. The user submits the login form to `/login_check`;
5. The security system intercepts the request, checks the user's submitted credentials, authenticates the user if they are correct, and sends the user back to the login form if they are not.

By default, if the submitted credentials are correct, the user will be redirected to the original page that was requested (e.g. `/admin/foo`). If the user originally went straight to the login page, he'll be redirected to the homepage. This can be highly customized, allowing you to, for example, redirect the user to a specific URL.

For more details on this and how to customize the form login process in general, see *How to customize your Form Login*.

2. http://api.symfony.com/2.0/Symfony/Component/Security/Core/Exception/AuthenticationException.html

Avoid Common Pitfalls

When setting up your login form, watch out for a few common pitfalls.

1. Create the correct routes

First, be sure that you've defined the `/login` and `/login_check` routes correctly and that they correspond to the `login_path` and `check_path` config values. A misconfiguration here can mean that you're redirected to a 404 page instead of the login page, or that submitting the login form does nothing (you just see the login form over and over again).

2. Be sure the login page isn't secure

Also, be sure that the login page does *not* require any roles to be viewed. For example, the following configuration - which requires the `ROLE_ADMIN` role for all URLs (including the `/login` URL), will cause a redirect loop:

Listing 13-7

```
access_control:
    - { path: ^/, roles: ROLE_ADMIN }
```

Removing the access control on the `/login` URL fixes the problem:

Listing 13-8

```
access_control:
    - { path: ^/login, roles: IS_AUTHENTICATED_ANONYMOUSLY }
    - { path: ^/, roles: ROLE_ADMIN }
```

Also, if your firewall does *not* allow for anonymous users, you'll need to create a special firewall that allows anonymous users for the login page:

Listing 13-9

```
firewalls:
    login_firewall:
        pattern:    ^/login$
        anonymous:  ~
    secured_area:
        pattern:    ^/
        form_login: ~
```

3. Be sure ``/login_check`` is behind a firewall

Next, make sure that your `check_path` URL (e.g. `/login_check`) is behind the firewall you're using for your form login (in this example, the single firewall matches *all* URLs, including `/login_check`). If `/login_check` doesn't match any firewall, you'll receive a `Unable to find the controller for path "/login_check"` exception.

4. Multiple firewalls don't share security context

If you're using multiple firewalls and you authenticate against one firewall, you will *not* be authenticated against any other firewalls automatically. Different firewalls are like different security systems. That's why, for most applications, having one main firewall is enough.

Authorization

The first step in security is always authentication: the process of verifying who the user is. With Symfony, authentication can be done in any way - via a form login, basic HTTP Authentication, or even via Facebook.

Once the user has been authenticated, authorization begins. Authorization provides a standard and powerful way to decide if a user can access any resource (a URL, a model object, a method call, ...). This works by assigning specific roles to each user, and then requiring different roles for different resources.

The process of authorization has two different sides:

1. The user has a specific set of roles;

2. A resource requires a specific role in order to be accessed.

In this section, you'll focus on how to secure different resources (e.g. URLs, method calls, etc) with different roles. Later, you'll learn more about how roles are created and assigned to users.

Securing Specific URL Patterns

The most basic way to secure part of your application is to secure an entire URL pattern. You've seen this already in the first example of this chapter, where anything matching the regular expression pattern `^/admin` requires the `ROLE_ADMIN` role.

You can define as many URL patterns as you need - each is a regular expression.

Listing 13-10

```
# app/config/security.yml
security:
    # ...
    access_control:
        - { path: ^/admin/users, roles: ROLE_SUPER_ADMIN }
        - { path: ^/admin, roles: ROLE_ADMIN }
```

Prepending the path with ^ ensures that only URLs *beginning* with the pattern are matched. For example, a path of simply `/admin` (without the ^) would correctly match `/admin/foo` but would also match URLs like `/foo/admin`.

For each incoming request, Symfony2 tries to find a matching access control rule (the first one wins). If the user isn't authenticated yet, the authentication process is initiated (i.e. the user is given a chance to login). However, if the user *is* authenticated but doesn't have the required role, an *`AccessDeniedException`*[3] exception is thrown, which you can handle and turn into a nice "access denied" error page for the user. See *How to customize Error Pages* for more information.

Since Symfony uses the first access control rule it matches, a URL like `/admin/users/new` will match the first rule and require only the `ROLE_SUPER_ADMIN` role. Any URL like `/admin/blog` will match the second rule and require `ROLE_ADMIN`.

Securing by IP

Certain situations may arise when you may need to restrict access to a given route based on IP. This is particularly relevant in the case of *Edge Side Includes* (ESI), for example, which utilize a route named "_internal". When ESI is used, the _internal route is required by the gateway cache to enable different caching options for subsections within a given page. This route comes with the ^/_internal prefix by default in the standard edition (assuming you've uncommented those lines from the routing file).

Here is an example of how you might secure this route from outside access:

Listing 13-11

```
# app/config/security.yml
security:
    # ...
    access_control:
        - { path: ^/_internal, roles: IS_AUTHENTICATED_ANONYMOUSLY, ip: 127.0.0.1 }
```

Securing by Channel

Much like securing based on IP, requiring the use of SSL is as simple as adding a new access_control entry:

3. http://api.symfony.com/2.0/Symfony/Component/Security/Core/Exception/AccessDeniedException.html

Listing 13-12
```
# app/config/security.yml
security:
    # ...
    access_control:
        - { path: ^/cart/checkout, roles: IS_AUTHENTICATED_ANONYMOUSLY, requires_channel:
https }
```

Securing a Controller

Protecting your application based on URL patterns is easy, but may not be fine-grained enough in certain cases. When necessary, you can easily force authorization from inside a controller:

Listing 13-13
```
use Symfony\Component\Security\Core\Exception\AccessDeniedException;
// ...

public function helloAction($name)
{
    if (false === $this->get('security.context')->isGranted('ROLE_ADMIN')) {
        throw new AccessDeniedException();
    }

    // ...
}
```

You can also choose to install and use the optional `JMSSecurityExtraBundle`, which can secure your controller using annotations:

Listing 13-14
```
use JMS\SecurityExtraBundle\Annotation\Secure;

/**
 * @Secure(roles="ROLE_ADMIN")
 */
public function helloAction($name)
{
    // ...
}
```

For more information, see the *JMSSecurityExtraBundle*[4] documentation. If you're using Symfony's Standard Distribution, this bundle is available by default. If not, you can easily download and install it.

Securing other Services

In fact, anything in Symfony can be protected using a strategy similar to the one seen in the previous section. For example, suppose you have a service (i.e. a PHP class) whose job is to send emails from one user to another. You can restrict use of this class - no matter where it's being used from - to users that have a specific role.

For more information on how you can use the security component to secure different services and methods in your application, see *How to secure any Service or Method in your Application*.

Access Control Lists (ACLs): Securing Individual Database Objects

Imagine you are designing a blog system where your users can comment on your posts. Now, you want a user to be able to edit his own comments, but not those of other users. Also, as the admin user, you yourself want to be able to edit *all* comments.

4. `https://github.com/schmittjoh/JMSSecurityExtraBundle`

The security component comes with an optional access control list (ACL) system that you can use when you need to control access to individual instances of an object in your system. *Without* ACL, you can secure your system so that only certain users can edit blog comments in general. But *with* ACL, you can restrict or allow access on a comment-by-comment basis.

For more information, see the cookbook article: *Access Control Lists (ACLs)*.

Users

In the previous sections, you learned how you can protect different resources by requiring a set of *roles* for a resource. In this section we'll explore the other side of authorization: users.

Where do Users come from? (*User Providers*)

During authentication, the user submits a set of credentials (usually a username and password). The job of the authentication system is to match those credentials against some pool of users. So where does this list of users come from?

In Symfony2, users can come from anywhere - a configuration file, a database table, a web service, or anything else you can dream up. Anything that provides one or more users to the authentication system is known as a "user provider". Symfony2 comes standard with the two most common user providers: one that loads users from a configuration file and one that loads users from a database table.

Specifying Users in a Configuration File

The easiest way to specify your users is directly in a configuration file. In fact, you've seen this already in the example in this chapter.

Listing 13-15

```yaml
# app/config/security.yml
security:
    # ...
    providers:
        default_provider:
            users:
                ryan:  { password: ryanpass, roles: 'ROLE_USER' }
                admin: { password: kitten, roles: 'ROLE_ADMIN' }
```

This user provider is called the "in-memory" user provider, since the users aren't stored anywhere in a database. The actual user object is provided by Symfony (*User*[5]).

Any user provider can load users directly from configuration by specifying the `users` configuration parameter and listing the users beneath it.

If your username is completely numeric (e.g. `77`) or contains a dash (e.g. `user-name`), you should use that alternative syntax when specifying users in YAML:

Listing 13-16

```yaml
users:
    - { name: 77, password: pass, roles: 'ROLE_USER' }
    - { name: user-name, password: pass, roles: 'ROLE_USER' }
```

5. http://api.symfony.com/2.0/Symfony/Component/Security/Core/User/User.html

For smaller sites, this method is quick and easy to setup. For more complex systems, you'll want to load your users from the database.

Loading Users from the Database

If you'd like to load your users via the Doctrine ORM, you can easily do this by creating a `User` class and configuring the `entity` provider.

With this approach, you'll first create your own `User` class, which will be stored in the database.

Listing 13-17
```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;

    // ...
}
```

As far as the security system is concerned, the only requirement for your custom user class is that it implements the *`UserInterface`*[6] interface. This means that your concept of a "user" can be anything, as long as it implements this interface.

The user object will be serialized and saved in the session during requests, therefore it is recommended that you *implement the Serializable interface*[7] in your user object. This is especially important if your `User` class has a parent class with private properties.

Next, configure an `entity` user provider, and point it to your `User` class:

Listing 13-18
```yaml
# app/config/security.yml
security:
    providers:
        main:
            entity: { class: Acme\UserBundle\Entity\User, property: username }
```

With the introduction of this new provider, the authentication system will attempt to load a `User` object from the database by using the `username` field of that class.

This example is just meant to show you the basic idea behind the `entity` provider. For a full working example, see *How to load Security Users from the Database (the Entity Provider)*.

For more information on creating your own custom provider (e.g. if you needed to load users via a web service), see *How to create a custom User Provider*.

---

6. http://api.symfony.com/2.0/Symfony/Component/Security/Core/User/UserInterface.html
7. http://php.net/manual/en/class.serializable.php

Encoding the User's Password

So far, for simplicity, all the examples have stored the users' passwords in plain text (whether those users are stored in a configuration file or in a database somewhere). Of course, in a real application, you'll want to encode your users' passwords for security reasons. This is easily accomplished by mapping your User class to one of several built-in "encoders". For example, to store your users in memory, but obscure their passwords via `sha1`, do the following:

Listing 13-19

```
# app/config/security.yml
security:
    # ...
    providers:
        in_memory:
            users:
                ryan:  { password: bb87a29949f3a1ee0559f8a57357487151281386, roles:
'ROLE_USER' }
                admin: { password: 74913f5cd5f61ec0bcfdb775414c2fb3d161b620, roles:
'ROLE_ADMIN' }

    encoders:
        Symfony\Component\Security\Core\User\User:
            algorithm:   sha1
            iterations: 1
            encode_as_base64: false
```

By setting the `iterations` to `1` and the `encode_as_base64` to false, the password is simply run through the `sha1` algorithm one time and without any extra encoding. You can now calculate the hashed password either programmatically (e.g. `hash('sha1', 'ryanpass')`) or via some online tool like *functions-online.com*[8]

If you're creating your users dynamically (and storing them in a database), you can use even tougher hashing algorithms and then rely on an actual password encoder object to help you encode passwords. For example, suppose your User object is `Acme\UserBundle\Entity\User` (like in the above example). First, configure the encoder for that user:

Listing 13-20

```
# app/config/security.yml
security:
    # ...

    encoders:
        Acme\UserBundle\Entity\User: sha512
```

In this case, you're using the stronger `sha512` algorithm. Also, since you've simply specified the algorithm (`sha512`) as a string, the system will default to hashing your password 5000 times in a row and then encoding it as base64. In other words, the password has been greatly obfuscated so that the hashed password can't be decoded (i.e. you can't determine the password from the hashed password).

If you have some sort of registration form for users, you'll need to be able to determine the hashed password so that you can set it on your user. No matter what algorithm you configure for your user object, the hashed password can always be determined in the following way from a controller:

Listing 13-21

```
$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);
```

8. http://www.functions-online.com/sha1.html

Retrieving the User Object

After authentication, the `User` object of the current user can be accessed via the `security.context` service. From inside a controller, this will look like:

Listing 13-22
```php
public function indexAction()
{
    $user = $this->get('security.context')->getToken()->getUser();
}
```

Anonymous users are technically authenticated, meaning that the `isAuthenticated()` method of an anonymous user object will return true. To check if your user is actually authenticated, check for the `IS_AUTHENTICATED_FULLY` role.

In a Twig Template this object can be accessed via the `app.user` key, which calls the *`GlobalVariables::getUser()`*[9] method:

Listing 13-23
```twig
<p>Username: {{ app.user.username }}</p>
```

Using Multiple User Providers

Each authentication mechanism (e.g. HTTP Authentication, form login, etc) uses exactly one user provider, and will use the first declared user provider by default. But what if you want to specify a few users via configuration and the rest of your users in the database? This is possible by creating a new provider that chains the two together:

Listing 13-24
```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 }
```

Now, all authentication mechanisms will use the `chain_provider`, since it's the first specified. The `chain_provider` will, in turn, try to load the user from both the `in_memory` and `user_db` providers.

If you have no reasons to separate your `in_memory` users from your `user_db` users, you can accomplish this even more easily by combining the two sources into a single provider:

Listing 13-25
```yaml
# app/config/security.yml
security:
    providers:
        main_provider:
            users:
                foo: { password: test }
            entity: { class: Acme\UserBundle\Entity\User, property: username }
```

You can also configure the firewall or individual authentication mechanisms to use a specific provider. Again, unless a provider is specified explicitly, the first provider is always used:

9. http://api.symfony.com/2.0/Symfony/Bundle/FrameworkBundle/Templating/GlobalVariables.html#getUser()

Listing 13-26

```yaml
# app/config/security.yml
security:
    firewalls:
        secured_area:
            # ...
            provider: user_db
            http_basic:
                realm: "Secured Demo Area"
                provider: in_memory
            form_login: ~
```

In this example, if a user tries to login via HTTP authentication, the authentication system will use the `in_memory` user provider. But if the user tries to login via the form login, the `user_db` provider will be used (since it's the default for the firewall as a whole).

For more information about user provider and firewall configuration, see the *Security Configuration Reference*.

Roles

The idea of a "role" is key to the authorization process. Each user is assigned a set of roles and then each resource requires one or more roles. If the user has the required roles, access is granted. Otherwise access is denied.

Roles are pretty simple, and are basically strings that you can invent and use as needed (though roles are objects internally). For example, if you need to start limiting access to the blog admin section of your website, you could protect that section using a `ROLE_BLOG_ADMIN` role. This role doesn't need to be defined anywhere - you can just start using it.

All roles **must** begin with the `ROLE_` prefix to be managed by Symfony2. If you define your own roles with a dedicated `Role` class (more advanced), don't use the `ROLE_` prefix.

Hierarchical Roles

Instead of associating many roles to users, you can define role inheritance rules by creating a role hierarchy:

Listing 13-27

```yaml
# app/config/security.yml
security:
    role_hierarchy:
        ROLE_ADMIN:       ROLE_USER
        ROLE_SUPER_ADMIN: [ROLE_ADMIN, ROLE_ALLOWED_TO_SWITCH]
```

In the above configuration, users with `ROLE_ADMIN` role will also have the `ROLE_USER` role. The `ROLE_SUPER_ADMIN` role has `ROLE_ADMIN`, `ROLE_ALLOWED_TO_SWITCH` and `ROLE_USER` (inherited from `ROLE_ADMIN`).

Logging Out

Usually, you'll also want your users to be able to log out. Fortunately, the firewall can handle this automatically for you when you activate the `logout` config parameter:

Listing 13-28

```yaml
# app/config/security.yml
security:
    firewalls:
        secured_area:
            # ...
            logout:
                path:   /logout
                target: /
    # ...
```

Once this is configured under your firewall, sending a user to `/logout` (or whatever you configure the `path` to be), will un-authenticate the current user. The user will then be sent to the homepage (the value defined by the `target` parameter). Both the `path` and `target` config parameters default to what's specified here. In other words, unless you need to customize them, you can omit them entirely and shorten your configuration:

Listing 13-29
```yaml
logout: ~
```

Note that you will *not* need to implement a controller for the `/logout` URL as the firewall takes care of everything. You may, however, want to create a route so that you can use it to generate the URL:

Listing 13-30
```yaml
# app/config/routing.yml
logout:
    pattern:   /logout
```

Once the user has been logged out, he will be redirected to whatever path is defined by the `target` parameter above (e.g. the `homepage`). For more information on configuring the logout, see the *Security Configuration Reference*.

Access Control in Templates

If you want to check if the current user has a role inside a template, use the built-in helper function:

Listing 13-31
```twig
{% if is_granted('ROLE_ADMIN') %}
    <a href="...">Delete</a>
{% endif %}
```

If you use this function and are *not* at a URL where there is a firewall active, an exception will be thrown. Again, it's almost always a good idea to have a main firewall that covers all URLs (as has been shown in this chapter).

Access Control in Controllers

If you want to check if the current user has a role in your controller, use the `isGranted` method of the security context:

Listing 13-32
```php
public function indexAction()
{
    // show different content to admin users
    if ($this->get('security.context')->isGranted('ROLE_ADMIN')) {
        // load admin content here
    }
    // load other regular content here
}
```

A firewall must be active or an exception will be thrown when the `isGranted` method is called. See the note above about templates for more details.

Impersonating a User

Sometimes, it's useful to be able to switch from one user to another without having to logout and login again (for instance when you are debugging or trying to understand a bug a user sees that you can't reproduce). This can be easily done by activating the `switch_user` firewall listener:

Listing 13-33

```yaml
# app/config/security.yml
security:
    firewalls:
        main:
            # ...
            switch_user: true
```

To switch to another user, just add a query string with the `_switch_user` parameter and the username as the value to the current URL:

http://example.com/somewhere?_switch_user=thomas[10]

To switch back to the original user, use the special `_exit` username:

http://example.com/somewhere?_switch_user=_exit[11]

Of course, this feature needs to be made available to a small group of users. By default, access is restricted to users having the `ROLE_ALLOWED_TO_SWITCH` role. The name of this role can be modified via the `role` setting. For extra security, you can also change the query parameter name via the `parameter` setting:

Listing 13-34

```yaml
# app/config/security.yml
security:
    firewalls:
        main:
            // ...
            switch_user: { role: ROLE_ADMIN, parameter: _want_to_be_this_user }
```

Stateless Authentication

By default, Symfony2 relies on a cookie (the Session) to persist the security context of the user. But if you use certificates or HTTP authentication for instance, persistence is not needed as credentials are available for each request. In that case, and if you don't need to store anything else between requests, you can activate the stateless authentication (which means that no cookie will be ever created by Symfony2):

Listing 13-35

```yaml
# app/config/security.yml
security:
    firewalls:
        main:
            http_basic: ~
            stateless:  true
```

10. `http://example.com/somewhere?_switch_user=thomas`
11. `http://example.com/somewhere?_switch_user=_exit`

If you use a form login, Symfony2 will create a cookie even if you set `stateless` to `true`.

Final Words

Security can be a deep and complex issue to solve correctly in your application. Fortunately, Symfony's security component follows a well-proven security model based around *authentication* and *authorization*. Authentication, which always happens first, is handled by a firewall whose job is to determine the identity of the user through several different methods (e.g. HTTP authentication, login form, etc). In the cookbook, you'll find examples of other methods for handling authentication, including how to implement a "remember me" cookie functionality.

Once a user is authenticated, the authorization layer can determine whether or not the user should have access to a specific resource. Most commonly, *roles* are applied to URLs, classes or methods and if the current user doesn't have that role, access is denied. The authorization layer, however, is much deeper, and follows a system of "voting" so that multiple parties can determine if the current user should have access to a given resource. Find out more about this and other topics in the cookbook.

Learn more from the Cookbook

- *Forcing HTTP/HTTPS*
- *Blacklist users by IP address with a custom voter*
- *Access Control Lists (ACLs)*
- *How to add "Remember Me" Login Functionality*

Chapter 14
HTTP Cache

The nature of rich web applications means that they're dynamic. No matter how efficient your application, each request will always contain more overhead than serving a static file.

And for most Web applications, that's fine. Symfony2 is lightning fast, and unless you're doing some serious heavy-lifting, each request will come back quickly without putting too much stress on your server.

But as your site grows, that overhead can become a problem. The processing that's normally performed on every request should be done only once. This is exactly what caching aims to accomplish.

Caching on the Shoulders of Giants

The most effective way to improve performance of an application is to cache the full output of a page and then bypass the application entirely on each subsequent request. Of course, this isn't always possible for highly dynamic websites, or is it? In this chapter, we'll show you how the Symfony2 cache system works and why we think this is the best possible approach.

The Symfony2 cache system is different because it relies on the simplicity and power of the HTTP cache as defined in the *HTTP specification*. Instead of reinventing a caching methodology, Symfony2 embraces the standard that defines basic communication on the Web. Once you understand the fundamental HTTP validation and expiration caching models, you'll be ready to master the Symfony2 cache system.

For the purposes of learning how to cache with Symfony2, we'll cover the subject in four steps:

- **Step 1**: A *gateway cache*, or reverse proxy, is an independent layer that sits in front of your application. The reverse proxy caches responses as they're returned from your application and answers requests with cached responses before they hit your application. Symfony2 provides its own reverse proxy, but any reverse proxy can be used.
- **Step 2**: *HTTP cache* headers are used to communicate with the gateway cache and any other caches between your application and the client. Symfony2 provides sensible defaults and a powerful interface for interacting with the cache headers.
- **Step 3**: HTTP *expiration and validation* are the two models used for determining whether cached content is *fresh* (can be reused from the cache) or *stale* (should be regenerated by the application).

- **Step 4**: *Edge Side Includes* (ESI) allow HTTP cache to be used to cache page fragments (even nested fragments) independently. With ESI, you can even cache an entire page for 60 minutes, but an embedded sidebar for only 5 minutes.

Since caching with HTTP isn't unique to Symfony, many articles already exist on the topic. If you're new to HTTP caching, we *highly* recommend Ryan Tomayko's article *Things Caches Do*[1]. Another in-depth resource is Mark Nottingham's *Cache Tutorial*[2].

Caching with a Gateway Cache

When caching with HTTP, the *cache* is separated from your application entirely and sits between your application and the client making the request.

The job of the cache is to accept requests from the client and pass them back to your application. The cache will also receive responses back from your application and forward them on to the client. The cache is the "middle-man" of the request-response communication between the client and your application.

Along the way, the cache will store each response that is deemed "cacheable" (See *Introduction to HTTP Caching*). If the same resource is requested again, the cache sends the cached response to the client, ignoring your application entirely.

This type of cache is known as a HTTP gateway cache and many exist such as *Varnish*[3], *Squid in reverse proxy mode*[4], and the Symfony2 reverse proxy.

Types of Caches

But a gateway cache isn't the only type of cache. In fact, the HTTP cache headers sent by your application are consumed and interpreted by up to three different types of caches:

- *Browser caches*: Every browser comes with its own local cache that is mainly useful for when you hit "back" or for images and other assets. The browser cache is a *private* cache as cached resources aren't shared with anyone else.
- *Proxy caches*: A proxy is a *shared* cache as many people can be behind a single one. It's usually installed by large corporations and ISPs to reduce latency and network traffic.
- *Gateway caches*: Like a proxy, it's also a *shared* cache but on the server side. Installed by network administrators, it makes websites more scalable, reliable and performant.

Gateway caches are sometimes referred to as reverse proxy caches, surrogate caches, or even HTTP accelerators.

The significance of *private* versus *shared* caches will become more obvious as we talk about caching responses containing content that is specific to exactly one user (e.g. account information).

Each response from your application will likely go through one or both of the first two cache types. These caches are outside of your control but follow the HTTP cache directions set in the response.

1. http://tomayko.com/writings/things-caches-do
2. http://www.mnot.net/cache_docs/
3. http://www.varnish-cache.org/
4. http://wiki.squid-cache.org/SquidFaq/ReverseProxy

Symfony2 Reverse Proxy

Symfony2 comes with a reverse proxy (also called a gateway cache) written in PHP. Enable it and cacheable responses from your application will start to be cached right away. Installing it is just as easy. Each new Symfony2 application comes with a pre-configured caching kernel (`AppCache`) that wraps the default one (`AppKernel`). The caching Kernel *is* the reverse proxy.

To enable caching, modify the code of a front controller to use the caching kernel:

Listing 14-1

```
// 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();
```

The caching kernel will immediately act as a reverse proxy - caching responses from your application and returning them to the client.

The cache kernel has a special `getLog()` method that returns a string representation of what happened in the cache layer. In the development environment, use it to debug and validate your cache strategy:

Listing 14-2

```
error_log($kernel->getLog());
```

The `AppCache` object has a sensible default configuration, but it can be finely tuned via a set of options you can set by overriding the `getOptions()` method:

Listing 14-3

```
// app/AppCache.php

use Symfony\Bundle\FrameworkBundle\HttpCache\HttpCache;

class AppCache extends HttpCache
{
    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,
        );
    }
}
```

Unless overridden in `getOptions()`, the `debug` option will be set to automatically be the debug value of the wrapped `AppKernel`.

Here is a list of the main options:

- `default_ttl`: The number of seconds that a cache entry should be considered fresh when no explicit freshness information is provided in a response. Explicit `Cache-Control` or `Expires` headers override this value (default: `0`);
- `private_headers`: Set of request headers that trigger "private" `Cache-Control` behavior on responses that don't explicitly state whether the response is `public` or `private` via a `Cache-Control` directive. (default: `Authorization` and `Cookie`);
- `allow_reload`: Specifies whether the client can force a cache reload by including a `Cache-Control` "no-cache" directive in the request. Set it to `true` for compliance with RFC 2616 (default: `false`);
- `allow_revalidate`: Specifies whether the client can force a cache revalidate by including a `Cache-Control` "max-age=0" directive in the request. Set it to `true` for compliance with RFC 2616 (default: false);
- `stale_while_revalidate`: Specifies the default number of seconds (the granularity is the second as the Response TTL precision is a second) during which the cache can immediately return a stale response while it revalidates it in the background (default: `2`); this setting is overridden by the `stale-while-revalidate` HTTP `Cache-Control` extension (see RFC 5861);
- `stale_if_error`: Specifies the default number of seconds (the granularity is the second) during which the cache can serve a stale response when an error is encountered (default: `60`). This setting is overridden by the `stale-if-error` HTTP `Cache-Control` extension (see RFC 5861).

If `debug` is `true`, Symfony2 automatically adds a `X-Symfony-Cache` header to the response containing useful information about cache hits and misses.

Changing from one Reverse Proxy to Another

The Symfony2 reverse proxy is a great tool to use when developing your website or when you deploy your website to a shared host where you cannot install anything beyond PHP code. But being written in PHP, it cannot be as fast as a proxy written in C. That's why we highly recommend you to use Varnish or Squid on your production servers if possible. The good news is that the switch from one proxy server to another is easy and transparent as no code modification is needed in your application. Start easy with the Symfony2 reverse proxy and upgrade later to Varnish when your traffic increases.

For more information on using Varnish with Symfony2, see the *How to use Varnish* cookbook chapter.

The performance of the Symfony2 reverse proxy is independent of the complexity of the application. That's because the application kernel is only booted when the request needs to be forwarded to it.

Introduction to HTTP Caching

To take advantage of the available cache layers, your application must be able to communicate which responses are cacheable and the rules that govern when/how that cache should become stale. This is done by setting HTTP cache headers on the response.

Keep in mind that "HTTP" is nothing more than the language (a simple text language) that web clients (e.g. browsers) and web servers use to communicate with each other. When we talk about HTTP caching, we're talking about the part of that language that allows clients and servers to exchange information related to caching.

HTTP specifies four response cache headers that we're concerned with:

- `Cache-Control`
- `Expires`
- `ETag`
- `Last-Modified`

The most important and versatile header is the `Cache-Control` header, which is actually a collection of various cache information.

Each of the headers will be explained in full detail in the *HTTP Expiration and Validation* section.

The Cache-Control Header

The `Cache-Control` header is unique in that it contains not one, but various pieces of information about the cacheability of a response. Each piece of information is separated by a comma:

Cache-Control: private, max-age=0, must-revalidate

Cache-Control: max-age=3600, must-revalidate

Symfony provides an abstraction around the `Cache-Control` header to make its creation more manageable:

Listing 14-4

```
$response = new Response();

// mark the response as either public or private
$response->setPublic();
$response->setPrivate();

// set the private or shared max age
$response->setMaxAge(600);
$response->setSharedMaxAge(600);

// set a custom Cache-Control directive
$response->headers->addCacheControlDirective('must-revalidate', true);
```

Public vs Private Responses

Both gateway and proxy caches are considered "shared" caches as the cached content is shared by more than one user. If a user-specific response were ever mistakenly stored by a shared cache, it might be

returned later to any number of different users. Imagine if your account information were cached and then returned to every subsequent user who asked for their account page!

To handle this situation, every response may be set to be public or private:

- *public*: Indicates that the response may be cached by both private and shared caches;
- *private*: Indicates that all or part of the response message is intended for a single user and must not be cached by a shared cache.

Symfony conservatively defaults each response to be private. To take advantage of shared caches (like the Symfony2 reverse proxy), the response will need to be explicitly set as public.

Safe Methods

HTTP caching only works for "safe" HTTP methods (like GET and HEAD). Being safe means that you never change the application's state on the server when serving the request (you can of course log information, cache data, etc). This has two very reasonable consequences:

- You should *never* change the state of your application when responding to a GET or HEAD request. Even if you don't use a gateway cache, the presence of proxy caches mean that any GET or HEAD request may or may not actually hit your server.
- Don't expect PUT, POST or DELETE methods to cache. These methods are meant to be used when mutating the state of your application (e.g. deleting a blog post). Caching them would prevent certain requests from hitting and mutating your application.

Caching Rules and Defaults

HTTP 1.1 allows caching anything by default unless there is an explicit `Cache-Control` header. In practice, most caches do nothing when requests have a cookie, an authorization header, use a non-safe method (i.e. PUT, POST, DELETE), or when responses have a redirect status code.

Symfony2 automatically sets a sensible and conservative `Cache-Control` header when none is set by the developer by following these rules:

- If no cache header is defined (`Cache-Control`, `Expires`, `ETag` or `Last-Modified`), `Cache-Control` is set to `no-cache`, meaning that the response will not be cached;
- If `Cache-Control` is empty (but one of the other cache headers is present), its value is set to `private, must-revalidate`;
- But if at least one `Cache-Control` directive is set, and no 'public' or `private` directives have been explicitly added, Symfony2 adds the `private` directive automatically (except when `s-maxage` is set).

HTTP Expiration and Validation

The HTTP specification defines two caching models:

- With the *expiration model*[5], you simply specify how long a response should be considered "fresh" by including a `Cache-Control` and/or an `Expires` header. Caches that understand expiration will not make the same request until the cached version reaches its expiration time and becomes "stale".
- When pages are really dynamic (i.e. their representation changes often), the *validation model*[6] is often necessary. With this model, the cache stores the response, but asks the server on each request whether or not the cached response is still valid. The application uses a unique

5. http://tools.ietf.org/html/rfc2616#section-13.2
6. http://tools.ietf.org/html/rfc2616#section-13.3

response identifier (the `Etag` header) and/or a timestamp (the `Last-Modified` header) to check if the page has changed since being cached.

The goal of both models is to never generate the same response twice by relying on a cache to store and return "fresh" responses.

Reading the HTTP Specification

The HTTP specification defines a simple but powerful language in which clients and servers can communicate. As a web developer, the request-response model of the specification dominates our work. Unfortunately, the actual specification document - *RFC 2616*[7] - can be difficult to read.

There is an on-going effort (*HTTP Bis*[8]) to rewrite the RFC 2616. It does not describe a new version of HTTP, but mostly clarifies the original HTTP specification. The organization is also improved as the specification is split into seven parts; everything related to HTTP caching can be found in two dedicated parts (*P4 - Conditional Requests*[9] and P6 - Caching: Browser and intermediary caches).

As a web developer, we strongly urge you to read the specification. Its clarity and power - even more than ten years after its creation - is invaluable. Don't be put-off by the appearance of the spec - its contents are much more beautiful than its cover.

Expiration

The expiration model is the more efficient and straightforward of the two caching models and should be used whenever possible. When a response is cached with an expiration, the cache will store the response and return it directly without hitting the application until it expires.

The expiration model can be accomplished using one of two, nearly identical, HTTP headers: `Expires` or `Cache-Control`.

Expiration with the `Expires` Header

According to the HTTP specification, "the `Expires` header field gives the date/time after which the response is considered stale." The `Expires` header can be set with the `setExpires()` `Response` method. It takes a `DateTime` instance as an argument:

Listing 14-5

```
$date = new DateTime();
$date->modify('+600 seconds');

$response->setExpires($date);
```

The resulting HTTP header will look like this:

Listing 14-6

```
Expires: Thu, 01 Mar 2011 16:00:00 GMT
```

The `setExpires()` method automatically converts the date to the GMT timezone as required by the specification.

Note that in HTTP versions before 1.1 the origin server wasn't required to send the `Date` header. Consequently the cache (e.g. the browser) might need to rely onto his local clock to evaluate the `Expires` header making the lifetime calculation vulnerable to clock skew. Another limitation of the `Expires`

7. http://tools.ietf.org/html/rfc2616
8. http://tools.ietf.org/wg/httpbis/
9. http://tools.ietf.org/html/draft-ietf-httpbis-p4-conditional-12

header is that the specification states that "HTTP/1.1 servers should not send `Expires` dates more than one year in the future."

Expiration with the `Cache-Control` Header

Because of the `Expires` header limitations, most of the time, you should use the `Cache-Control` header instead. Recall that the `Cache-Control` header is used to specify many different cache directives. For expiration, there are two directives, `max-age` and `s-maxage`. The first one is used by all caches, whereas the second one is only taken into account by shared caches:

Listing 14-7
```
// Sets the number of seconds after which the response
// should no longer be considered fresh
$response->setMaxAge(600);

// Same as above but only for shared caches
$response->setSharedMaxAge(600);
```

The `Cache-Control` header would take on the following format (it may have additional directives):

Listing 14-8
```
Cache-Control: max-age=600, s-maxage=600
```

Validation

When a resource needs to be updated as soon as a change is made to the underlying data, the expiration model falls short. With the expiration model, the application won't be asked to return the updated response until the cache finally becomes stale.

The validation model addresses this issue. Under this model, the cache continues to store responses. The difference is that, for each request, the cache asks the application whether or not the cached response is still valid. If the cache *is* still valid, your application should return a 304 status code and no content. This tells the cache that it's ok to return the cached response.

Under this model, you mainly save bandwidth as the representation is not sent twice to the same client (a 304 response is sent instead). But if you design your application carefully, you might be able to get the bare minimum data needed to send a 304 response and save CPU also (see below for an implementation example).

The 304 status code means "Not Modified". It's important because with this status code do *not* contain the actual content being requested. Instead, the response is simply a light-weight set of directions that tell cache that it should use its stored version.

Like with expiration, there are two different HTTP headers that can be used to implement the validation model: `ETag` and `Last-Modified`.

Validation with the `ETag` Header

The `ETag` header is a string header (called the "entity-tag") that uniquely identifies one representation of the target resource. It's entirely generated and set by your application so that you can tell, for example, if the `/about` resource that's stored by the cache is up-to-date with what your application would return. An `ETag` is like a fingerprint and is used to quickly compare if two different versions of a resource are equivalent. Like fingerprints, each `ETag` must be unique across all representations of the same resource.

Let's walk through a simple implementation that generates the ETag as the md5 of the content:

Listing 14-9
```
public function indexAction()
{
    $response = $this->render('MyBundle:Main:index.html.twig');
```

```
    $response->setETag(md5($response->getContent()));
    $response->isNotModified($this->getRequest());

    return $response;
}
```

The `Response::isNotModified()` method compares the `ETag` sent with the `Request` with the one set on the `Response`. If the two match, the method automatically sets the `Response` status code to 304.

This algorithm is simple enough and very generic, but you need to create the whole `Response` before being able to compute the ETag, which is sub-optimal. In other words, it saves on bandwidth, but not CPU cycles.

In the *Optimizing your Code with Validation* section, we'll show how validation can be used more intelligently to determine the validity of a cache without doing so much work.

Symfony2 also supports weak ETags by passing `true` as the second argument to the *`setETag()`*[10] method.

Validation with the `Last-Modified` Header

The `Last-Modified` header is the second form of validation. According to the HTTP specification, "The `Last-Modified` header field indicates the date and time at which the origin server believes the representation was last modified." In other words, the application decides whether or not the cached content has been updated based on whether or not it's been updated since the response was cached.

For instance, you can use the latest update date for all the objects needed to compute the resource representation as the value for the `Last-Modified` header value:

Listing 14-10

```
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;
}
```

The `Response::isNotModified()` method compares the `If-Modified-Since` header sent by the request with the `Last-Modified` header set on the response. If they are equivalent, the `Response` will be set to a 304 status code.

The `If-Modified-Since` request header equals the `Last-Modified` header of the last response sent to the client for the particular resource. This is how the client and server communicate with each other and decide whether or not the resource has been updated since it was cached.

10. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/Response.html#setETag()

Optimizing your Code with Validation

The main goal of any caching strategy is to lighten the load on the application. Put another way, the less you do in your application to return a 304 response, the better. The `Response::isNotModified()` method does exactly that by exposing a simple and efficient pattern:

Listing 14-11
```
public function showAction($articleSlug)
{
    // Get the minimum information to compute
    // the ETag or the Last-Modified value
    // (based on the Request, data is retrieved from
    // a database or a key-value store for instance)
    $article = // ...

    // create a Response with a ETag and/or a Last-Modified header
    $response = new Response();
    $response->setETag($article->computeETag());
    $response->setLastModified($article->getPublishedAt());

    // Check that the Response is not modified for the given Request
    if ($response->isNotModified($this->getRequest())) {
        // return the 304 Response immediately
        return $response;
    } else {
        // do more work here - like retrieving more data
        $comments = // ...

        // or render a template with the $response you've already started
        return $this->render(
            'MyBundle:MyController:article.html.twig',
            array('article' => $article, 'comments' => $comments),
            $response
        );
    }
}
```

When the `Response` is not modified, the `isNotModified()` automatically sets the response status code to `304`, removes the content, and removes some headers that must not be present for `304` responses (see *`setNotModified()`*[11]).

Varying the Response

So far, we've assumed that each URI has exactly one representation of the target resource. By default, HTTP caching is done by using the URI of the resource as the cache key. If two people request the same URI of a cacheable resource, the second person will receive the cached version.

Sometimes this isn't enough and different versions of the same URI need to be cached based on one or more request header values. For instance, if you compress pages when the client supports it, any given URI has two representations: one when the client supports compression, and one when it does not. This determination is done by the value of the `Accept-Encoding` request header.

In this case, we need the cache to store both a compressed and uncompressed version of the response for the particular URI and return them based on the request's `Accept-Encoding` value. This is done by using the `Vary` response header, which is a comma-separated list of different headers whose values trigger a different representation of the requested resource:

Listing 14-12
```
Vary: Accept-Encoding, User-Agent
```

11. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/Response.html#setNotModified()

This particular `Vary` header would cache different versions of each resource based on the URI and the value of the `Accept-Encoding` and `User-Agent` request header.

The `Response` object offers a clean interface for managing the `Vary` header:

Listing 14-13

```
// set one vary header
$response->setVary('Accept-Encoding');

// set multiple vary headers
$response->setVary(array('Accept-Encoding', 'User-Agent'));
```

The `setVary()` method takes a header name or an array of header names for which the response varies.

Expiration and Validation

You can of course use both validation and expiration within the same `Response`. As expiration wins over validation, you can easily benefit from the best of both worlds. In other words, by using both expiration and validation, you can instruct the cache to serve the cached content, while checking back at some interval (the expiration) to verify that the content is still valid.

More Response Methods

The Response class provides many more methods related to the cache. Here are the most useful ones:

Listing 14-14

```
// Marks the Response stale
$response->expire();

// Force the response to return a proper 304 response with no content
$response->setNotModified();
```

Additionally, most cache-related HTTP headers can be set via the single `setCache()` method:

Listing 14-15

```
// Set cache settings in one call
$response->setCache(array(
    'etag'          => $etag,
    'last_modified' => $date,
    'max_age'       => 10,
    's_maxage'      => 10,
    'public'        => true,
    // 'private'    => true,
));
```

Using Edge Side Includes

Gateway caches are a great way to make your website perform better. But they have one limitation: they can only cache whole pages. If you can't cache whole pages or if parts of a page has "more" dynamic parts, you are out of luck. Fortunately, Symfony2 provides a solution for these cases, based on a technology called *ESI*[12], or Edge Side Includes. Akamaï wrote this specification almost 10 years ago, and it allows specific parts of a page to have a different caching strategy than the main page.

The ESI specification describes tags you can embed in your pages to communicate with the gateway cache. Only one tag is implemented in Symfony2, `include`, as this is the only useful one outside of Akamaï context:

12. http://www.w3.org/TR/esi-lang

Listing 14-16
```
<html>
    <body>
        Some content

        <!-- Embed the content of another page here -->
        <esi:include src="http://..." />

        More content
    </body>
</html>
```

Notice from the example that each ESI tag has a fully-qualified URL. An ESI tag represents a page fragment that can be fetched via the given URL.

When a request is handled, the gateway cache fetches the entire page from its cache or requests it from the backend application. If the response contains one or more ESI tags, these are processed in the same way. In other words, the gateway cache either retrieves the included page fragment from its cache or requests the page fragment from the backend application again. When all the ESI tags have been resolved, the gateway cache merges each into the main page and sends the final content to the client.

All of this happens transparently at the gateway cache level (i.e. outside of your application). As you'll see, if you choose to take advantage of ESI tags, Symfony2 makes the process of including them almost effortless.

Using ESI in Symfony2

First, to use ESI, be sure to enable it in your application configuration:

Listing 14-17
```
# app/config/config.yml
framework:
    # ...
    esi: { enabled: true }
```

Now, suppose we have a page that is relatively static, except for a news ticker at the bottom of the content. With ESI, we can cache the news ticker independent of the rest of the page.

Listing 14-18
```
public function indexAction()
{
    $response = $this->render('MyBundle:MyController:index.html.twig');
    $response->setSharedMaxAge(600);

    return $response;
}
```

In this example, we've given the full-page cache a lifetime of ten minutes. Next, let's include the news ticker in the template by embedding an action. This is done via the `render` helper (See *Embedding Controllers* for more details).

As the embedded content comes from another page (or controller for that matter), Symfony2 uses the standard `render` helper to configure ESI tags:

Listing 14-19
```
{% render '...:news' with {}, {'standalone': true} %}
```

By setting `standalone` to `true`, you tell Symfony2 that the action should be rendered as an ESI tag. You might be wondering why you would want to use a helper instead of just writing the ESI tag yourself. That's because using a helper makes your application work even if there is no gateway cache installed. Let's see how it works.

When standalone is `false` (the default), Symfony2 merges the included page content within the main one before sending the response to the client. But when standalone is `true`, *and* if Symfony2 detects that it's talking to a gateway cache that supports ESI, it generates an ESI include tag. But if there is no gateway cache or if it does not support ESI, Symfony2 will just merge the included page content within the main one as it would have done were standalone set to `false`.

Symfony2 detects if a gateway cache supports ESI via another Akamaï specification that is supported out of the box by the Symfony2 reverse proxy.

The embedded action can now specify its own caching rules, entirely independent of the master page.

Listing 14-20

```
public function newsAction()
{
  // ...

  $response->setSharedMaxAge(60);
}
```

With ESI, the full page cache will be valid for 600 seconds, but the news component cache will only last for 60 seconds.

A requirement of ESI, however, is that the embedded action be accessible via a URL so the gateway cache can fetch it independently of the rest of the page. Of course, an action can't be accessed via a URL unless it has a route that points to it. Symfony2 takes care of this via a generic route and controller. For the ESI include tag to work properly, you must define the `_internal` route:

Listing 14-21

```
# app/config/routing.yml
_internal:
    resource: "@FrameworkBundle/Resources/config/routing/internal.xml"
    prefix:   /_internal
```

Since this route allows all actions to be accessed via a URL, you might want to protect it by using the Symfony2 firewall feature (by allowing access to your reverse proxy's IP range). See the *Securing by IP* section of the *Security Chapter* for more information on how to do this.

One great advantage of this caching strategy is that you can make your application as dynamic as needed and at the same time, hit the application as little as possible.

Once you start using ESI, remember to always use the `s-maxage` directive instead of `max-age`. As the browser only ever receives the aggregated resource, it is not aware of the sub-components, and so it will obey the `max-age` directive and cache the entire page. And you don't want that.

The `render` helper supports two other useful options:

- `alt`: used as the `alt` attribute on the ESI tag, which allows you to specify an alternative URL to be used if the `src` cannot be found;
- `ignore_errors`: if set to true, an `onerror` attribute will be added to the ESI with a value of `continue` indicating that, in the event of a failure, the gateway cache will simply remove the ESI tag silently.

Cache Invalidation

> "There are only two hard things in Computer Science: cache invalidation and naming things."
> --Phil Karlton

You should never need to invalidate cached data because invalidation is already taken into account natively in the HTTP cache models. If you use validation, you never need to invalidate anything by definition; and if you use expiration and need to invalidate a resource, it means that you set the expires date too far away in the future.

Since invalidation is a topic specific to each type of reverse proxy, if you don't worry about invalidation, you can switch between reverse proxies without changing anything in your application code.

Actually, all reverse proxies provide ways to purge cached data, but you should avoid them as much as possible. The most standard way is to purge the cache for a given URL by requesting it with the special `PURGE` HTTP method.

Here is how you can configure the Symfony2 reverse proxy to support the `PURGE` HTTP method:

Listing 14-22

```php
// app/AppCache.php

use Symfony\Bundle\FrameworkBundle\HttpCache\HttpCache;

class AppCache extends HttpCache
{
    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;
    }
}
```

You must protect the `PURGE` HTTP method somehow to avoid random people purging your cached data.

Summary

Symfony2 was designed to follow the proven rules of the road: HTTP. Caching is no exception. Mastering the Symfony2 cache system means becoming familiar with the HTTP cache models and using them effectively. This means that, instead of relying only on Symfony2 documentation and code examples, you have access to a world of knowledge related to HTTP caching and gateway caches such as Varnish.

Learn more from the Cookbook

- *How to use Varnish to speed up my Website*

Chapter 15
Translations

The term "internationalization" (often abbreviated *i18n*[1]) refers to the process of abstracting strings and other locale-specific pieces out of your application and into a layer where they can be translated and converted based on the user's locale (i.e. language and country). For text, this means wrapping each with a function capable of translating the text (or "message") into the language of the user:

Listing 15-1
```
// text will *always* print out in English
echo 'Hello World';

// text can be translated into the end-user's language or default to English
echo $translator->trans('Hello World');
```

The term *locale* refers roughly to the user's language and country. It can be any string that your application uses to manage translations and other format differences (e.g. currency format). We recommended the *ISO639-1*[2] *language* code, an underscore (_), then the *ISO3166 Alpha-2*[3] *country* code (e.g. `fr_FR` for French/France).

In this chapter, we'll learn how to prepare an application to support multiple locales and then how to create translations for multiple locales. Overall, the process has several common steps:

1. Enable and configure Symfony's `Translation` component;
2. Abstract strings (i.e. "messages") by wrapping them in calls to the `Translator`;
3. Create translation resources for each supported locale that translate each message in the application;
4. Determine, set and manage the user's locale in the session.

Configuration

Translations are handled by a `Translator` *service* that uses the user's locale to lookup and return translated messages. Before using it, enable the `Translator` in your configuration:

1. http://en.wikipedia.org/wiki/Internationalization_and_localization
2. http://en.wikipedia.org/wiki/List_of_ISO_639-1_codes
3. http://en.wikipedia.org/wiki/ISO_3166-1#Current_codes

```
# app/config/config.yml
framework:
    translator: { fallback: en }
```

Listing 15-2

The `fallback` option defines the fallback locale when a translation does not exist in the user's locale.

When a translation does not exist for a locale, the translator first tries to find the translation for the language (`fr` if the locale is `fr_FR` for instance). If this also fails, it looks for a translation using the fallback locale.

The locale used in translations is the one stored in the user session.

Basic Translation

Translation of text is done through the `translator` service (*`Translator`*[4]). To translate a block of text (called a *message*), use the *`trans()`*[5] method. Suppose, for example, that we're translating a simple message from inside a controller:

```
public function indexAction()
{
    $t = $this->get('translator')->trans('Symfony2 is great');

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

Listing 15-3

When this code is executed, Symfony2 will attempt to translate the message "Symfony2 is great" based on the `locale` of the user. For this to work, we need to tell Symfony2 how to translate the message via a "translation resource", which is a collection of message translations for a given locale. This "dictionary" of translations can be created in several different formats, XLIFF being the recommended format:

```
<!-- 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>
```

Listing 15-4

Now, if the language of the user's locale is French (e.g. `fr_FR` or `fr_BE`), the message will be translated into `J'aime Symfony2`.

The Translation Process

To actually translate the message, Symfony2 uses a simple process:

- The `locale` of the current user, which is stored in the session, is determined;
- A catalog of translated messages is loaded from translation resources defined for the `locale` (e.g. `fr_FR`). Messages from the fallback locale are also loaded and added to the catalog if

4. http://api.symfony.com/2.0/Symfony/Component/Translation/Translator.html
5. http://api.symfony.com/2.0/Symfony/Component/Translation/Translator.html#trans()

they don't already exist. The end result is a large "dictionary" of translations. See Message Catalogues for more details;
- If the message is located in the catalog, the translation is returned. If not, the translator returns the original message.

When using the `trans()` method, Symfony2 looks for the exact string inside the appropriate message catalog and returns it (if it exists).

Message Placeholders

Sometimes, a message containing a variable needs to be translated:

Listing 15-5
```
public function indexAction($name)
{
    $t = $this->get('translator')->trans('Hello '.$name);

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

However, creating a translation for this string is impossible since the translator will try to look up the exact message, including the variable portions (e.g. "Hello Ryan" or "Hello Fabien"). Instead of writing a translation for every possible iteration of the `$name` variable, we can replace the variable with a "placeholder":

Listing 15-6
```
public function indexAction($name)
{
    $t = $this->get('translator')->trans('Hello %name%', array('%name%' => $name));

    new Response($t);
}
```

Symfony2 will now look for a translation of the raw message (`Hello %name%`) and *then* replace the placeholders with their values. Creating a translation is done just as before:

Listing 15-7
```
<!-- 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>
```

The placeholders can take on any form as the full message is reconstructed using the PHP *strtr function*[6]. However, the `%var%` notation is required when translating in Twig templates, and is overall a sensible convention to follow.

As we've seen, creating a translation is a two-step process:

1. Abstract the message that needs to be translated by processing it through the `Translator`.
2. Create a translation for the message in each locale that you choose to support.

6. http://www.php.net/manual/en/function.strtr.php

The second step is done by creating message catalogues that define the translations for any number of different locales.

Message Catalogues

When a message is translated, Symfony2 compiles a message catalogue for the user's locale and looks in it for a translation of the message. A message catalogue is like a dictionary of translations for a specific locale. For example, the catalogue for the `fr_FR` locale might contain the following translation:

    Symfony2 is Great => J'aime Symfony2

It's the responsibility of the developer (or translator) of an internationalized application to create these translations. Translations are stored on the filesystem and discovered by Symfony, thanks to some conventions.

Each time you create a *new* translation resource (or install a bundle that includes a translation resource), be sure to clear your cache so that Symfony can discover the new translation resource:

```
php app/console cache:clear
```

Listing 15-8

Translation Locations and Naming Conventions

Symfony2 looks for message files (i.e. translations) in two locations:

- For messages found in a bundle, the corresponding message files should live in the `Resources/translations/` directory of the bundle;
- To override any bundle translations, place message files in the `app/Resources/translations` directory.

The filename of the translations is also important as Symfony2 uses a convention to determine details about the translations. Each message file must be named according to the following pattern: `domain.locale.loader`:

- **domain**: An optional way to organize messages into groups (e.g. `admin`, `navigation` or the default `messages`) - see Using Message Domains;
- **locale**: The locale that the translations are for (e.g. `en_GB`, `en`, etc);
- **loader**: How Symfony2 should load and parse the file (e.g. `xliff`, `php` or `yml`).

The loader can be the name of any registered loader. By default, Symfony provides the following loaders:

- `xliff`: XLIFF file;
- `php`: PHP file;
- `yml`: YAML file.

The choice of which loader to use is entirely up to you and is a matter of taste.

You can also store translations in a database, or any other storage by providing a custom class implementing the *`LoaderInterface`*[7] interface.

7. http://api.symfony.com/2.0/Symfony/Component/Translation/Loader/LoaderInterface.html

Creating Translations

The act of creating translation files is an important part of "localization" (often abbreviated *L10n*[8]). Translation files consist of a series of id-translation pairs for the given domain and locale. The source is the identifier for the individual translation, and can be the message in the main locale (e.g. "Symfony is great") of your application or a unique identifier (e.g. "symfony2.great" - see the sidebar below):

Listing 15-9

```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>
```

Symfony2 will discover these files and use them when translating either "Symfony2 is great" or "symfony2.great" into a French language locale (e.g. `fr_FR` or `fr_BE`).

8. http://en.wikipedia.org/wiki/Internationalization_and_localization

Using Real or Keyword Messages

This example illustrates the two different philosophies when creating messages to be translated:

```
$t = $translator->trans('Symfony2 is great');

$t = $translator->trans('symfony2.great');
```

Listing 15-10

In the first method, messages are written in the language of the default locale (English in this case). That message is then used as the "id" when creating translations.

In the second method, messages are actually "keywords" that convey the idea of the message. The keyword message is then used as the "id" for any translations. In this case, translations must be made for the default locale (i.e. to translate `symfony2.great` to `Symfony2 is great`).

The second method is handy because the message key won't need to be changed in every translation file if we decide that the message should actually read "Symfony2 is really great" in the default locale.

The choice of which method to use is entirely up to you, but the "keyword" format is often recommended.

Additionally, the `php` and `yaml` file formats support nested ids to avoid repeating yourself if you use keywords instead of real text for your ids:

```
symfony2:
    is:
        great: Symfony2 is great
        amazing: Symfony2 is amazing
    has:
        bundles: Symfony2 has bundles
user:
    login: Login
```

Listing 15-11

The multiple levels are flattened into single id/translation pairs by adding a dot (.) between every level, therefore the above examples are equivalent to the following:

```
symfony2.is.great: Symfony2 is great
symfony2.is.amazing: Symfony2 is amazing
symfony2.has.bundles: Symfony2 has bundles
user.login: Login
```

Listing 15-12

Using Message Domains

As we've seen, message files are organized into the different locales that they translate. The message files can also be organized further into "domains". When creating message files, the domain is the first portion of the filename. The default domain is `messages`. For example, suppose that, for organization, translations were split into three different domains: `messages`, `admin` and `navigation`. The French translation would have the following message files:

- `messages.fr.xliff`
- `admin.fr.xliff`
- `navigation.fr.xliff`

When translating strings that are not in the default domain (`messages`), you must specify the domain as the third argument of `trans()`:

```
$this->get('translator')->trans('Symfony2 is great', array(), 'admin');
```

Listing 15-13

Symfony2 will now look for the message in the `admin` domain of the user's locale.

Handling the User's Locale

The locale of the current user is stored in the session and is accessible via the `session` service:

Listing 15-14
```
$locale = $this->get('session')->getLocale();

$this->get('session')->setLocale('en_US');
```

Fallback and Default Locale

If the locale hasn't been set explicitly in the session, the `fallback_locale` configuration parameter will be used by the `Translator`. The parameter defaults to `en` (see Configuration).

Alternatively, you can guarantee that a locale is set on the user's session by defining a `default_locale` for the session service:

Listing 15-15
```
# app/config/config.yml
framework:
    session: { default_locale: en }
```

The Locale and the URL

Since the locale of the user is stored in the session, it may be tempting to use the same URL to display a resource in many different languages based on the user's locale. For example, `http://www.example.com/contact` could show content in English for one user and French for another user. Unfortunately, this violates a fundamental rule of the Web: that a particular URL returns the same resource regardless of the user. To further muddy the problem, which version of the content would be indexed by search engines?

A better policy is to include the locale in the URL. This is fully-supported by the routing system using the special `_locale` parameter:

Listing 15-16
```
contact:
    pattern:   /{_locale}/contact
    defaults:  { _controller: AcmeDemoBundle:Contact:index, _locale: en }
    requirements:
        _locale: en|fr|de
```

When using the special *_locale* parameter in a route, the matched locale will *automatically be set on the user's session*. In other words, if a user visits the URI `/fr/contact`, the locale `fr` will automatically be set as the locale for the user's session.

You can now use the user's locale to create routes to other translated pages in your application.

Pluralization

Message pluralization is a tough topic as the rules can be quite complex. For instance, here is the mathematic representation of the Russian pluralization rules:

Listing 15-17
```
(($number % 10 == 1) && ($number % 100 != 11)) ? 0 : ((($number % 10 >= 2) && ($number % 10 <=
4) && (($number % 100 < 10) || ($number % 100 >= 20))) ? 1 : 2);
```

As you can see, in Russian, you can have three different plural forms, each given an index of 0, 1 or 2. For each form, the plural is different, and so the translation is also different.

When a translation has different forms due to pluralization, you can provide all the forms as a string separated by a pipe (|):

```
'There is one apple|There are %count% apples'
```

Listing 15-18

To translate pluralized messages, use the *transChoice()*[9] method:

```
$t = $this->get('translator')->transChoice(
    'There is one apple|There are %count% apples',
    10,
    array('%count%' => 10)
);
```

Listing 15-19

The second argument (`10` in this example), is the *number* of objects being described and is used to determine which translation to use and also to populate the `%count%` placeholder.

Based on the given number, the translator chooses the right plural form. In English, most words have a singular form when there is exactly one object and a plural form for all other numbers (0, 2, 3...). So, if `count` is `1`, the translator will use the first string (`There is one apple`) as the translation. Otherwise it will use `There are %count% apples`.

Here is the French translation:

```
'Il y a %count% pomme|Il y a %count% pommes'
```

Listing 15-20

Even if the string looks similar (it is made of two sub-strings separated by a pipe), the French rules are different: the first form (no plural) is used when `count` is `0` or `1`. So, the translator will automatically use the first string (`Il y a %count% pomme`) when `count` is `0` or `1`.

Each locale has its own set of rules, with some having as many as six different plural forms with complex rules behind which numbers map to which plural form. The rules are quite simple for English and French, but for Russian, you'd may want a hint to know which rule matches which string. To help translators, you can optionally "tag" each string:

```
'one: There is one apple|some: There are %count% apples'

'none_or_one: Il y a %count% pomme|some: Il y a %count% pommes'
```

Listing 15-21

The tags are really only hints for translators and don't affect the logic used to determine which plural form to use. The tags can be any descriptive string that ends with a colon (`:`). The tags also do not need to be the same in the original message as in the translated one.

Explicit Interval Pluralization

The easiest way to pluralize a message is to let Symfony2 use internal logic to choose which string to use based on a given number. Sometimes, you'll need more control or want a different translation for specific cases (for `0`, or when the count is negative, for example). For such cases, you can use explicit math intervals:

```
'{0} There are no apples|{1} There is one apple|]1,19] There are %count% apples|[20,Inf] There are many apples'
```

Listing 15-22

The intervals follow the *ISO 31-11*[10] notation. The above string specifies four different intervals: exactly `0`, exactly `1`, `2-19`, and `20` and higher.

You can also mix explicit math rules and standard rules. In this case, if the count is not matched by a specific interval, the standard rules take effect after removing the explicit rules:

9. http://api.symfony.com/2.0/Symfony/Component/Translation/Translator.html#transChoice()
10. http://en.wikipedia.org/wiki/Interval_%28mathematics%29#The_ISO_notation

Listing 15-23
```
'{0} There are no apples|[20,Inf] There are many apples|There is one apple|a_few: There are %count% apples'
```

For example, for `1` apple, the standard rule `There is one apple` will be used. For `2-19` apples, the second standard rule `There are %count% apples` will be selected.

An *`Interval`*[11] can represent a finite set of numbers:

Listing 15-24
```
{1,2,3,4}
```

Or numbers between two other numbers:

Listing 15-25
```
[1, +Inf[
]-1,2[
```

The left delimiter can be `[` (inclusive) or `]` (exclusive). The right delimiter can be `[` (exclusive) or `]` (inclusive). Beside numbers, you can use `-Inf` and `+Inf` for the infinite.

Translations in Templates

Most of the time, translation occurs in templates. Symfony2 provides native support for both Twig and PHP templates.

Twig Templates

Symfony2 provides specialized Twig tags (`trans` and `transchoice`) to help with message translation of *static blocks of text*:

Listing 15-26
```
{% trans %}Hello %name%{% endtrans %}

{% transchoice count %}
    {0} There are no apples|{1} There is one apple|]1,Inf] There are %count% apples
{% endtranschoice %}
```

The `transchoice` tag automatically gets the `%count%` variable from the current context and passes it to the translator. This mechanism only works when you use a placeholder following the `%var%` pattern.

If you need to use the percent character (`%`) in a string, escape it by doubling it: `{% trans %}Percent: %percent%%%{% endtrans %}`

You can also specify the message domain and pass some additional variables:

Listing 15-27
```
{% trans with {'%name%': 'Fabien'} from "app" %}Hello %name%{% endtrans %}

{% trans with {'%name%': 'Fabien'} from "app" into "fr" %}Hello %name%{% endtrans %}

{% transchoice count with {'%name%': 'Fabien'} from "app" %}
    {0} There is no apples|{1} There is one apple|]1,Inf] There are %count% apples
{% endtranschoice %}
```

The `trans` and `transchoice` filters can be used to translate *variable texts* and complex expressions:

Listing 15-28
```
{{ message|trans }}
```

11. http://api.symfony.com/2.0/Symfony/Component/Translation/Interval.html

```
{{ message|transchoice(5) }}

{{ message|trans({'%name%': 'Fabien'}, "app") }}

{{ message|transchoice(5, {'%name%': 'Fabien'}, 'app') }}
```

Using the translation tags or filters have the same effect, but with one subtle difference: automatic output escaping is only applied to variables translated using a filter. In other words, if you need to be sure that your translated variable is *not* output escaped, you must apply the raw filter after the translation filter:

Listing 15-29

```
{# text translated between tags is never escaped #}
{% trans %}
    <h3>foo</h3>
{% endtrans %}

{% set message = '<h3>foo</h3>' %}

{# a variable translated via a filter is escaped by default #}
{{ message|trans|raw }}

{# but static strings are never escaped #}
{{ '<h3>foo</h3>'|trans }}
```

PHP Templates

The translator service is accessible in PHP templates through the `translator` helper:

Listing 15-30

```
<?php echo $view['translator']->trans('Symfony2 is great') ?>

<?php echo $view['translator']->transChoice(
    '{0} There is no apples|{1} There is one apple|]1,Inf[ There are %count% apples',
    10,
    array('%count%' => 10)
) ?>
```

Forcing the Translator Locale

When translating a message, Symfony2 uses the locale from the user's session or the `fallback` locale if necessary. You can also manually specify the locale to use for translation:

Listing 15-31

```
$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',
);
```

Translating Database Content

The translation of database content should be handled by Doctrine through the *Translatable Extension*[12]. For more information, see the documentation for that library.

Translating Constraint Messages

The best way to understand constraint translation is to see it in action. To start, suppose you've created a plain-old-PHP object that you need to use somewhere in your application:

Listing 15-32
```php
// src/Acme/BlogBundle/Entity/Author.php
namespace Acme\BlogBundle\Entity;

class Author
{
    public $name;
}
```

Add constraints though any of the supported methods. Set the message option to the translation source text. For example, to guarantee that the $name property is not empty, add the following:

Listing 15-33
```yaml
# src/Acme/BlogBundle/Resources/config/validation.yml
Acme\BlogBundle\Entity\Author:
    properties:
        name:
            - NotBlank: { message: "author.name.not_blank" }
```

Create a translation file under the `validators` catalog for the constraint messages, typically in the `Resources/translations/` directory of the bundle. See Message Catalogues for more details.

Listing 15-34
```xml
<!-- validators.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>author.name.not_blank</source>
                <target>Please enter an author name.</target>
            </trans-unit>
        </body>
    </file>
</xliff>
```

Summary

With the Symfony2 Translation component, creating an internationalized application no longer needs to be a painful process and boils down to just a few basic steps:

- Abstract messages in your application by wrapping each in either the *trans()*[13] or *transChoice()*[14] methods;

12. https://github.com/l3pp4rd/DoctrineExtensions
13. http://api.symfony.com/2.0/Symfony/Component/Translation/Translator.html#trans()
14. http://api.symfony.com/2.0/Symfony/Component/Translation/Translator.html#transChoice()

- Translate each message into multiple locales by creating translation message files. Symfony2 discovers and processes each file because its name follows a specific convention;
- Manage the user's locale, which is stored in the session.

Chapter 16
Service Container

A modern PHP application is full of objects. One object may facilitate the delivery of email messages while another may allow you to persist information into a database. In your application, you may create an object that manages your product inventory, or another object that processes data from a third-party API. The point is that a modern application does many things and is organized into many objects that handle each task.

In this chapter, we'll talk about a special PHP object in Symfony2 that helps you instantiate, organize and retrieve the many objects of your application. This object, called a service container, will allow you to standardize and centralize the way objects are constructed in your application. The container makes your life easier, is super fast, and emphasizes an architecture that promotes reusable and decoupled code. And since all core Symfony2 classes use the container, you'll learn how to extend, configure and use any object in Symfony2. In large part, the service container is the biggest contributor to the speed and extensibility of Symfony2.

Finally, configuring and using the service container is easy. By the end of this chapter, you'll be comfortable creating your own objects via the container and customizing objects from any third-party bundle. You'll begin writing code that is more reusable, testable and decoupled, simply because the service container makes writing good code so easy.

What is a Service?

Put simply, a *Service* is any PHP object that performs some sort of "global" task. It's a purposefully-generic name used in computer science to describe an object that's created for a specific purpose (e.g. delivering emails). Each service is used throughout your application whenever you need the specific functionality it provides. You don't have to do anything special to make a service: simply write a PHP class with some code that accomplishes a specific task. Congratulations, you've just created a service!

As a rule, a PHP object is a service if it is used globally in your application. A single `Mailer` service is used globally to send email messages whereas the many `Message` objects that it delivers are *not* services. Similarly, a `Product` object is not a service, but an object that persists `Product` objects to a database *is* a service.

So what's the big deal then? The advantage of thinking about "services" is that you begin to think about separating each piece of functionality in your application into a series of services. Since each service does just one job, you can easily access each service and use its functionality wherever you need it. Each service can also be more easily tested and configured since it's separated from the other functionality in your application. This idea is called *service-oriented architecture*[1] and is not unique to Symfony2 or even PHP. Structuring your application around a set of independent service classes is a well-known and trusted object-oriented best-practice. These skills are key to being a good developer in almost any language.

What is a Service Container?

A *Service Container* (or *dependency injection container*) is simply a PHP object that manages the instantiation of services (i.e. objects). For example, suppose we have a simple PHP class that delivers email messages. Without a service container, we must manually create the object whenever we need it:

Listing 16-1

```
use Acme\HelloBundle\Mailer;

$mailer = new Mailer('sendmail');
$mailer->send('ryan@foobar.net', ... );
```

This is easy enough. The imaginary `Mailer` class allows us to configure the method used to deliver the email messages (e.g. `sendmail`, `smtp`, etc). But what if we wanted to use the mailer service somewhere else? We certainly don't want to repeat the mailer configuration *every* time we need to use the `Mailer` object. What if we needed to change the `transport` from `sendmail` to `smtp` everywhere in the application? We'd need to hunt down every place we create a `Mailer` service and change it.

Creating/Configuring Services in the Container

A better answer is to let the service container create the `Mailer` object for you. In order for this to work, we must *teach* the container how to create the `Mailer` service. This is done via configuration, which can be specified in YAML, XML or PHP:

Listing 16-2

```
# app/config/config.yml
services:
    my_mailer:
        class:        Acme\HelloBundle\Mailer
        arguments:    [sendmail]
```

When Symfony2 initializes, it builds the service container using the application configuration (`app/config/config.yml` by default). The exact file that's loaded is dictated by the `AppKernel::registerContainerConfiguration()` method, which loads an environment-specific configuration file (e.g. `config_dev.yml` for the `dev` environment or `config_prod.yml` for `prod`).

An instance of the `Acme\HelloBundle\Mailer` object is now available via the service container. The container is available in any traditional Symfony2 controller where you can access the services of the container via the `get()` shortcut method:

Listing 16-3

```
class HelloController extends Controller
{
    // ...

    public function sendEmailAction()
```

1. http://wikipedia.org/wiki/Service-oriented_architecture

```
    {
        // ...
        $mailer = $this->get('my_mailer');
        $mailer->send('ryan@foobar.net', ... );
    }
}
```

When we ask for the `my_mailer` service from the container, the container constructs the object and returns it. This is another major advantage of using the service container. Namely, a service is *never* constructed until it's needed. If you define a service and never use it on a request, the service is never created. This saves memory and increases the speed of your application. This also means that there's very little or no performance hit for defining lots of services. Services that are never used are never constructed.

As an added bonus, the `Mailer` service is only created once and the same instance is returned each time you ask for the service. This is almost always the behavior you'll need (it's more flexible and powerful), but we'll learn later how you can configure a service that has multiple instances.

Service Parameters

The creation of new services (i.e. objects) via the container is pretty straightforward. Parameters make defining services more organized and flexible:

Listing 16-4
```
# 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%]
```

The end result is exactly the same as before - the difference is only in *how* we defined the service. By surrounding the `my_mailer.class` and `my_mailer.transport` strings in percent (%) signs, the container knows to look for parameters with those names. When the container is built, it looks up the value of each parameter and uses it in the service definition.

The percent sign inside a parameter or argument, as part of the string, must be escaped with another percent sign:

Listing 16-5
```
<argument type="string">http://symfony.com/?foo=%%s&bar=%%d</argument>
```

The purpose of parameters is to feed information into services. Of course there was nothing wrong with defining the service without using any parameters. Parameters, however, have several advantages:

- separation and organization of all service "options" under a single `parameters` key;
- parameter values can be used in multiple service definitions;
- when creating a service in a bundle (we'll show this shortly), using parameters allows the service to be easily customized in your application.

The choice of using or not using parameters is up to you. High-quality third-party bundles will *always* use parameters as they make the service stored in the container more configurable. For the services in your application, however, you may not need the flexibility of parameters.

Array Parameters

Parameters do not need to be flat strings, they can also be arrays. For the XML format, you need to use the type="collection" attribute for all parameters that are arrays.

Listing 16-6

```yaml
# app/config/config.yml
parameters:
    my_mailer.gateways:
        - mail1
        - mail2
        - mail3
    my_multilang.language_fallback:
        en:
            - en
            - fr
        fr:
            - fr
            - en
```

Importing other Container Configuration Resources

In this section, we'll refer to service configuration files as *resources*. This is to highlight that fact that, while most configuration resources will be files (e.g. YAML, XML, PHP), Symfony2 is so flexible that configuration could be loaded from anywhere (e.g. a database or even via an external web service).

The service container is built using a single configuration resource (`app/config/config.yml` by default). All other service configuration (including the core Symfony2 and third-party bundle configuration) must be imported from inside this file in one way or another. This gives you absolute flexibility over the services in your application.

External service configuration can be imported in two different ways. First, we'll talk about the method that you'll use most commonly in your application: the `imports` directive. In the following section, we'll introduce the second method, which is the flexible and preferred method for importing service configuration from third-party bundles.

Importing Configuration with `imports`

So far, we've placed our `my_mailer` service container definition directly in the application configuration file (e.g. `app/config/config.yml`). Of course, since the `Mailer` class itself lives inside the `AcmeHelloBundle`, it makes more sense to put the `my_mailer` container definition inside the bundle as well.

First, move the `my_mailer` container definition into a new container resource file inside `AcmeHelloBundle`. If the `Resources` or `Resources/config` directories don't exist, create them.

Listing 16-7

```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%]
```

The definition itself hasn't changed, only its location. Of course the service container doesn't know about the new resource file. Fortunately, we can easily import the resource file using the `imports` key in the application configuration.

Listing 16-8
```
# app/config/config.yml
imports:
    - { resource: @AcmeHelloBundle/Resources/config/services.yml }
```

The `imports` directive allows your application to include service container configuration resources from any other location (most commonly from bundles). The `resource` location, for files, is the absolute path to the resource file. The special `@AcmeHello` syntax resolves the directory path of the `AcmeHelloBundle` bundle. This helps you specify the path to the resource without worrying later if you move the `AcmeHelloBundle` to a different directory.

Importing Configuration via Container Extensions

When developing in Symfony2, you'll most commonly use the `imports` directive to import container configuration from the bundles you've created specifically for your application. Third-party bundle container configuration, including Symfony2 core services, are usually loaded using another method that's more flexible and easy to configure in your application.

Here's how it works. Internally, each bundle defines its services very much like we've seen so far. Namely, a bundle uses one or more configuration resource files (usually XML) to specify the parameters and services for that bundle. However, instead of importing each of these resources directly from your application configuration using the `imports` directive, you can simply invoke a *service container extension* inside the bundle that does the work for you. A service container extension is a PHP class created by the bundle author to accomplish two things:

- import all service container resources needed to configure the services for the bundle;
- provide semantic, straightforward configuration so that the bundle can be configured without interacting with the flat parameters of the bundle's service container configuration.

In other words, a service container extension configures the services for a bundle on your behalf. And as we'll see in a moment, the extension provides a sensible, high-level interface for configuring the bundle.

Take the `FrameworkBundle` - the core Symfony2 framework bundle - as an example. The presence of the following code in your application configuration invokes the service container extension inside the `FrameworkBundle`:

Listing 16-9
```
# app/config/config.yml
framework:
    secret:          xxxxxxxxxx
    charset:         UTF-8
    form:            true
    csrf_protection: true
    router:        { resource: "%kernel.root_dir%/config/routing.yml" }
    # ...
```

When the configuration is parsed, the container looks for an extension that can handle the `framework` configuration directive. The extension in question, which lives in the `FrameworkBundle`, is invoked and the service configuration for the `FrameworkBundle` is loaded. If you remove the `framework` key from your application configuration file entirely, the core Symfony2 services won't be loaded. The point is that you're in control: the Symfony2 framework doesn't contain any magic or perform any actions that you don't have control over.

Of course you can do much more than simply "activate" the service container extension of the `FrameworkBundle`. Each extension allows you to easily customize the bundle, without worrying about how the internal services are defined.

In this case, the extension allows you to customize the `charset`, `error_handler`, `csrf_protection`, `router` configuration and much more. Internally, the `FrameworkBundle` uses the options specified here to define and configure the services specific to it. The bundle takes care of creating all the necessary `parameters` and `services` for the service container, while still allowing much of the configuration to be easily customized. As an added bonus, most service container extensions are also smart enough to perform validation - notifying you of options that are missing or the wrong data type.

When installing or configuring a bundle, see the bundle's documentation for how the services for the bundle should be installed and configured. The options available for the core bundles can be found inside the *Reference Guide*.

Natively, the service container only recognizes the `parameters`, `services`, and `imports` directives. Any other directives are handled by a service container extension.

If you want to expose user friendly configuration in your own bundles, read the "*How to expose a Semantic Configuration for a Bundle*" cookbook recipe.

Referencing (Injecting) Services

So far, our original `my_mailer` service is simple: it takes just one argument in its constructor, which is easily configurable. As you'll see, the real power of the container is realized when you need to create a service that depends on one or more other services in the container.

Let's start with an example. Suppose we have a new service, `NewsletterManager`, that helps to manage the preparation and delivery of an email message to a collection of addresses. Of course the `my_mailer` service is already really good at delivering email messages, so we'll use it inside `NewsletterManager` to handle the actual delivery of the messages. This pretend class might look something like this:

Listing 16-10

```
namespace Acme\HelloBundle\Newsletter;

use Acme\HelloBundle\Mailer;

class NewsletterManager
{
    protected $mailer;

    public function __construct(Mailer $mailer)
    {
        $this->mailer = $mailer;
    }

    // ...
}
```

Without using the service container, we can create a new `NewsletterManager` fairly easily from inside a controller:

Listing 16-11

```
public function sendNewsletterAction()
{
    $mailer = $this->get('my_mailer');
    $newsletter = new Acme\HelloBundle\Newsletter\NewsletterManager($mailer);
    // ...
}
```

This approach is fine, but what if we decide later that the `NewsletterManager` class needs a second or third constructor argument? What if we decide to refactor our code and rename the class? In both cases,

you'd need to find every place where the `NewsletterManager` is instantiated and modify it. Of course, the service container gives us a much more appealing option:

Listing 16-12
```
# 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]
```

In YAML, the special `@my_mailer` syntax tells the container to look for a service named `my_mailer` and to pass that object into the constructor of `NewsletterManager`. In this case, however, the specified service `my_mailer` must exist. If it does not, an exception will be thrown. You can mark your dependencies as optional - this will be discussed in the next section.

Using references is a very powerful tool that allows you to create independent service classes with well-defined dependencies. In this example, the `newsletter_manager` service needs the `my_mailer` service in order to function. When you define this dependency in the service container, the container takes care of all the work of instantiating the objects.

Optional Dependencies: Setter Injection

Injecting dependencies into the constructor in this manner is an excellent way of ensuring that the dependency is available to use. If you have optional dependencies for a class, then "setter injection" may be a better option. This means injecting the dependency using a method call rather than through the constructor. The class would look like this:

Listing 16-13
```
namespace Acme\HelloBundle\Newsletter;

use Acme\HelloBundle\Mailer;

class NewsletterManager
{
    protected $mailer;

    public function setMailer(Mailer $mailer)
    {
        $this->mailer = $mailer;
    }

    // ...
}
```

Injecting the dependency by the setter method just needs a change of syntax:

Listing 16-14
```
# 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 ] ]
```

The approaches presented in this section are called "constructor injection" and "setter injection". The Symfony2 service container also supports "property injection".

Making References Optional

Sometimes, one of your services may have an optional dependency, meaning that the dependency is not required for your service to work properly. In the example above, the `my_mailer` service *must* exist, otherwise an exception will be thrown. By modifying the `newsletter_manager` service definition, you can make this reference optional. The container will then inject it if it exists and do nothing if it doesn't:

Listing 16-15

```
# src/Acme/HelloBundle/Resources/config/services.yml
parameters:
    # ...

services:
    newsletter_manager:
        class:     %newsletter_manager.class%
        arguments: [@?my_mailer]
```

In YAML, the special `@?` syntax tells the service container that the dependency is optional. Of course, the `NewsletterManager` must also be written to allow for an optional dependency:

Listing 16-16

```
public function __construct(Mailer $mailer = null)
{
    // ...
}
```

Core Symfony and Third-Party Bundle Services

Since Symfony2 and all third-party bundles configure and retrieve their services via the container, you can easily access them or even use them in your own services. To keep things simple, Symfony2 by default does not require that controllers be defined as services. Furthermore Symfony2 injects the entire service container into your controller. For example, to handle the storage of information on a user's session, Symfony2 provides a `session` service, which you can access inside a standard controller as follows:

Listing 16-17

```
public function indexAction($bar)
{
    $session = $this->get('session');
    $session->set('foo', $bar);

    // ...
}
```

In Symfony2, you'll constantly use services provided by the Symfony core or other third-party bundles to perform tasks such as rendering templates (`templating`), sending emails (`mailer`), or accessing information on the request (`request`).

We can take this a step further by using these services inside services that you've created for your application. Let's modify the `NewsletterManager` to use the real Symfony2 `mailer` service (instead of the

pretend `my_mailer`). Let's also pass the templating engine service to the `NewsletterManager` so that it can generate the email content via a template:

Listing 16-18

```
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;
    }

    // ...
}
```

Configuring the service container is easy:

Listing 16-19

```
services:
    newsletter_manager:
        class:     %newsletter_manager.class%
        arguments: [@mailer, @templating]
```

The `newsletter_manager` service now has access to the core `mailer` and `templating` services. This is a common way to create services specific to your application that leverage the power of different services within the framework.

Be sure that `swiftmailer` entry appears in your application configuration. As we mentioned in *Importing Configuration via Container Extensions*, the `swiftmailer` key invokes the service extension from the `SwiftmailerBundle`, which registers the `mailer` service.

Advanced Container Configuration

As we've seen, defining services inside the container is easy, generally involving a `service` configuration key and a few parameters. However, the container has several other tools available that help to *tag* services for special functionality, create more complex services, and perform operations after the container is built.

Marking Services as public / private

When defining services, you'll usually want to be able to access these definitions within your application code. These services are called `public`. For example, the `doctrine` service registered with the container when using the DoctrineBundle is a public service as you can access it via:

Listing 16-20

```
$doctrine = $container->get('doctrine');
```

However, there are use-cases when you don't want a service to be public. This is common when a service is only defined because it could be used as an argument for another service.

If you use a private service as an argument to more than one other service, this will result in two different instances being used as the instantiation of the private service is done inline (e.g. `new PrivateFooBar()`).

Simply said: A service will be private when you do not want to access it directly from your code.

Here is an example:

```
services:
   foo:
     class: Acme\HelloBundle\Foo
     public: false
```

Listing 16-21

Now that the service is private, you *cannot* call:

```
$container->get('foo');
```

Listing 16-22

However, if a service has been marked as private, you can still alias it (see below) to access this service (via the alias).

Services are by default public.

Aliasing

When using core or third party bundles within your application, you may want to use shortcuts to access some services. You can do so by aliasing them and, furthermore, you can even alias non-public services.

```
services:
   foo:
     class: Acme\HelloBundle\Foo
   bar:
     alias: foo
```

Listing 16-23

This means that when using the container directly, you can access the `foo` service by asking for the `bar` service like this:

```
$container->get('bar'); // Would return the foo service
```

Listing 16-24

Requiring files

There might be use cases when you need to include another file just before the service itself gets loaded. To do so, you can use the `file` directive.

```
services:
   foo:
     class: Acme\HelloBundle\Foo\Bar
     file: %kernel.root_dir%/src/path/to/file/foo.php
```

Listing 16-25

Notice that symfony will internally call the PHP function require_once which means that your file will be included only once per request.

Tags (`tags`)

In the same way that a blog post on the Web might be tagged with things such as "Symfony" or "PHP", services configured in your container can also be tagged. In the service container, a tag implies that the service is meant to be used for a specific purpose. Take the following example:

Listing 16-26

```
services:
    foo.twig.extension:
        class: Acme\HelloBundle\Extension\FooExtension
        tags:
            -  { name: twig.extension }
```

The `twig.extension` tag is a special tag that the `TwigBundle` uses during configuration. By giving the service this `twig.extension` tag, the bundle knows that the `foo.twig.extension` service should be registered as a Twig extension with Twig. In other words, Twig finds all services tagged with `twig.extension` and automatically registers them as extensions.

Tags, then, are a way to tell Symfony2 or other third-party bundles that your service should be registered or used in some special way by the bundle.

The following is a list of tags available with the core Symfony2 bundles. Each of these has a different effect on your service and many tags require additional arguments (beyond just the `name` parameter).

- 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

Learn more

- *How to Use a Factory to Create Services*
- *How to Manage Common Dependencies with Parent Services*
- *How to define Controllers as Services*

Chapter 17
Performance

Symfony2 is fast, right out of the box. Of course, if you really need speed, there are many ways that you can make Symfony even faster. In this chapter, you'll explore many of the most common and powerful ways to make your Symfony application even faster.

Use a Byte Code Cache (e.g. APC)

One the best (and easiest) things that you should do to improve your performance is to use a "byte code cache". The idea of a byte code cache is to remove the need to constantly recompile the PHP source code. There are a number of *byte code caches*[1] available, some of which are open source. The most widely used byte code cache is probably *APC*[2]

Using a byte code cache really has no downside, and Symfony2 has been architected to perform really well in this type of environment.

Further Optimizations

Byte code caches usually monitor the source files for changes. This ensures that if the source of a file changes, the byte code is recompiled automatically. This is really convenient, but obviously adds overhead.

For this reason, some byte code caches offer an option to disable these checks. Obviously, when disabling these checks, it will be up to the server admin to ensure that the cache is cleared whenever any source files change. Otherwise, the updates you've made won't be seen.

For example, to disable these checks in APC, simply add `apc.stat=0` to your php.ini configuration.

1. `http://en.wikipedia.org/wiki/List_of_PHP_accelerators`
2. `http://php.net/manual/en/book.apc.php`

Use an Autoloader that caches (e.g. `ApcUniversalClassLoader`)

By default, the Symfony2 standard edition uses the `UniversalClassLoader` in the *autoloader.php*[3] file. This autoloader is easy to use, as it will automatically find any new classes that you've placed in the registered directories.

Unfortunately, this comes at a cost, as the loader iterates over all configured namespaces to find a particular file, making `file_exists` calls until it finally finds the file it's looking for.

The simplest solution is to cache the location of each class after it's located the first time. Symfony comes with a class - `ApcUniversalClassLoader` - loader that extends the `UniversalClassLoader` and stores the class locations in APC.

To use this class loader, simply adapt your `autoloader.php` as follows:

Listing 17-1
```
// app/autoload.php
require __DIR__.'/../vendor/symfony/src/Symfony/Component/ClassLoader/
ApcUniversalClassLoader.php';

use Symfony\Component\ClassLoader\ApcUniversalClassLoader;

$loader = new ApcUniversalClassLoader('some caching unique prefix');
// ...
```

When using the APC autoloader, if you add new classes, they will be found automatically and everything will work the same as before (i.e. no reason to "clear" the cache). However, if you change the location of a particular namespace or prefix, you'll need to flush your APC cache. Otherwise, the autoloader will still be looking at the old location for all classes inside that namespace.

Use Bootstrap Files

To ensure optimal flexibility and code reuse, Symfony2 applications leverage a variety of classes and 3rd party components. But loading all of these classes from separate files on each request can result in some overhead. To reduce this overhead, the Symfony2 Standard Edition provides a script to generate a so-called *bootstrap file*[4], consisting of multiple classes definitions in a single file. By including this file (which contains a copy of many of the core classes), Symfony no longer needs to include any of the source files containing those classes. This will reduce disc IO quite a bit.

If you're using the Symfony2 Standard Edition, then you're probably already using the bootstrap file. To be sure, open your front controller (usually `app.php`) and check to make sure that the following line exists:

Listing 17-2
```
require_once __DIR__.'/../app/bootstrap.php.cache';
```

Note that there are two disadvantages when using a bootstrap file:

- the file needs to be regenerated whenever any of the original sources change (i.e. when you update the Symfony2 source or vendor libraries);
- when debugging, one will need to place break points inside the bootstrap file.

If you're using Symfony2 Standard Edition, the bootstrap file is automatically rebuilt after updating the vendor libraries via the `php bin/vendors install` command.

3. https://github.com/symfony/symfony-standard/blob/master/app/autoload.php
4. https://github.com/sensio/SensioDistributionBundle/blob/2.0/Resources/bin/build_bootstrap.php

Bootstrap Files and Byte Code Caches

Even when using a byte code cache, performance will improve when using a bootstrap file since there will be less files to monitor for changes. Of course if this feature is disabled in the byte code cache (e.g. `apc.stat=0` in APC), there is no longer a reason to use a bootstrap file.

Chapter 18
Internals

Looks like you want to understand how Symfony2 works and how to extend it. That makes me very happy! This section is an in-depth explanation of the Symfony2 internals.

You need to read this section only if you want to understand how Symfony2 works behind the scene, or if you want to extend Symfony2.

Overview

The Symfony2 code is made of several independent layers. Each layer is built on top of the previous one.

Autoloading is not managed by the framework directly; it's done independently with the help of the *`UniversalClassLoader`*[1] class and the `src/autoload.php` file. Read the *dedicated chapter* for more information.

`HttpFoundation` Component

The deepest level is the *`HttpFoundation`*[2] component. HttpFoundation provides the main objects needed to deal with HTTP. It is an Object-Oriented abstraction of some native PHP functions and variables:

- The *`Request`*[3] class abstracts the main PHP global variables like `$_GET`, `$_POST`, `$_COOKIE`, `$_FILES`, and `$_SERVER`;
- The *`Response`*[4] class abstracts some PHP functions like `header()`, `setcookie()`, and `echo`;

1. http://api.symfony.com/2.0/Symfony/Component/ClassLoader/UniversalClassLoader.html
2. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation.html
3. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/Request.html
4. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/Response.html

- The *Session*[5] class and *SessionStorageInterface*[6] interface abstract session management `session_*()` functions.

`HttpKernel` Component

On top of HttpFoundation is the *HttpKernel*[7] component. HttpKernel handles the dynamic part of HTTP; it is a thin wrapper on top of the Request and Response classes to standardize the way requests are handled. It also provides extension points and tools that makes it the ideal starting point to create a Web framework without too much overhead.

It also optionally adds configurability and extensibility, thanks to the Dependency Injection component and a powerful plugin system (bundles).

Read more about Dependency Injection and Bundles.

`FrameworkBundle` Bundle

The *FrameworkBundle*[8] bundle is the bundle that ties the main components and libraries together to make a lightweight and fast MVC framework. It comes with a sensible default configuration and conventions to ease the learning curve.

Kernel

The *HttpKernel*[9] class is the central class of Symfony2 and is responsible for handling client requests. Its main goal is to "convert" a *Request*[10] object to a *Response*[11] object.

Every Symfony2 Kernel implements *HttpKernelInterface*[12]:

```
function handle(Request $request, $type = self::MASTER_REQUEST, $catch = true)
```

Listing 18-1

Controllers

To convert a Request to a Response, the Kernel relies on a "Controller". A Controller can be any valid PHP callable.

The Kernel delegates the selection of what Controller should be executed to an implementation of *ControllerResolverInterface*[13]:

```
public function getController(Request $request);

public function getArguments(Request $request, $controller);
```

Listing 18-2

The *getController()*[14] method returns the Controller (a PHP callable) associated with the given Request. The default implementation (*ControllerResolver*[15]) looks for a `_controller` request attribute

5. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/Session.html
6. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/SessionStorage/SessionStorageInterface.html
7. http://api.symfony.com/2.0/Symfony/Component/HttpKernel.html
8. http://api.symfony.com/2.0/Symfony/Bundle/FrameworkBundle.html
9. http://api.symfony.com/2.0/Symfony/Component/HttpKernel/HttpKernel.html
10. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/Request.html
11. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/Response.html
12. http://api.symfony.com/2.0/Symfony/Component/HttpKernel/HttpKernelInterface.html
13. http://api.symfony.com/2.0/Symfony/Component/HttpKernel/Controller/ControllerResolverInterface.html
14. http://api.symfony.com/2.0/Symfony/Component/HttpKernel/Controller/ControllerResolverInterface.html#getController()
15. http://api.symfony.com/2.0/Symfony/Component/HttpKernel/Controller/ControllerResolver.html

that represents the controller name (a "class::method" string, like `Bundle\BlogBundle\PostController:indexAction`).

The default implementation uses the *`RouterListener`*[16] to define the `_controller` Request attribute (see *kernel.request Event*).

The *`getArguments()`*[17] method returns an array of arguments to pass to the Controller callable. The default implementation automatically resolves the method arguments, based on the Request attributes.

Matching Controller method arguments from Request attributes

For each method argument, Symfony2 tries to get the value of a Request attribute with the same name. If it is not defined, the argument default value is used if defined:

Listing 18-3
```php
// Symfony2 will look for an 'id' attribute (mandatory)
// and an 'admin' one (optional)
public function showAction($id, $admin = true)
{
    // ...
}
```

Handling Requests

The `handle()` method takes a `Request` and *always* returns a `Response`. To convert the `Request`, `handle()` relies on the Resolver and an ordered chain of Event notifications (see the next section for more information about each Event):

1. Before doing anything else, the `kernel.request` event is notified -- if one of the listeners returns a `Response`, it jumps to step 8 directly;
2. The Resolver is called to determine the Controller to execute;
3. Listeners of the `kernel.controller` event can now manipulate the Controller callable the way they want (change it, wrap it, ...);
4. The Kernel checks that the Controller is actually a valid PHP callable;
5. The Resolver is called to determine the arguments to pass to the Controller;
6. The Kernel calls the Controller;
7. If the Controller does not return a `Response`, listeners of the `kernel.view` event can convert the Controller return value to a `Response`;
8. Listeners of the `kernel.response` event can manipulate the `Response` (content and headers);
9. The Response is returned.

If an Exception is thrown during processing, the `kernel.exception` is notified and listeners are given a chance to convert the Exception to a Response. If that works, the `kernel.response` event is notified; if not, the Exception is re-thrown.

If you don't want Exceptions to be caught (for embedded requests for instance), disable the `kernel.exception` event by passing `false` as the third argument to the `handle()` method.

Internal Requests

At any time during the handling of a request (the 'master' one), a sub-request can be handled. You can pass the request type to the `handle()` method (its second argument):

16. http://api.symfony.com/2.0/Symfony/Bundle/FrameworkBundle/EventListener/RouterListener.html
17. http://api.symfony.com/2.0/Symfony/Component/HttpKernel/Controller/ControllerResolverInterface.html#getArguments()

- `HttpKernelInterface::MASTER_REQUEST`;
- `HttpKernelInterface::SUB_REQUEST`.

The type is passed to all events and listeners can act accordingly (some processing must only occur on the master request).

Events

Each event thrown by the Kernel is a subclass of *`KernelEvent`*[18]. This means that each event has access to the same basic information:

- `getRequestType()` - returns the *type* of the request (`HttpKernelInterface::MASTER_REQUEST` or `HttpKernelInterface::SUB_REQUEST`);
- `getKernel()` - returns the Kernel handling the request;
- `getRequest()` - returns the current `Request` being handled.

`getRequestType()`

The `getRequestType()` method allows listeners to know the type of the request. For instance, if a listener must only be active for master requests, add the following code at the beginning of your listener method:

Listing 18-4

```
use Symfony\Component\HttpKernel\HttpKernelInterface;

if (HttpKernelInterface::MASTER_REQUEST !== $event->getRequestType()) {
    // return immediately
    return;
}
```

If you are not yet familiar with the Symfony2 Event Dispatcher, read the *Event Dispatcher Component Documentation* section first.

`kernel.request` Event

Event Class: *`GetResponseEvent`*[19]

The goal of this event is to either return a `Response` object immediately or setup variables so that a Controller can be called after the event. Any listener can return a `Response` object via the `setResponse()` method on the event. In this case, all other listeners won't be called.

This event is used by `FrameworkBundle` to populate the `_controller Request` attribute, via the *`RouterListener`*[20]. RequestListener uses a *`RouterInterface`*[21] object to match the `Request` and determine the Controller name (stored in the `_controller Request` attribute).

`kernel.controller` Event

Event Class: *`FilterControllerEvent`*[22]

This event is not used by `FrameworkBundle`, but can be an entry point used to modify the controller that should be executed:

Listing 18-5

18. http://api.symfony.com/2.0/Symfony/Component/HttpKernel/Event/KernelEvent.html
19. http://api.symfony.com/2.0/Symfony/Component/HttpKernel/Event/GetResponseEvent.html
20. http://api.symfony.com/2.0/Symfony/Bundle/FrameworkBundle/EventListener/RouterListener.html
21. http://api.symfony.com/2.0/Symfony/Component/Routing/RouterInterface.html
22. http://api.symfony.com/2.0/Symfony/Component/HttpKernel/Event/FilterControllerEvent.html

```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);
}
```

`kernel.view` Event

Event Class: *`GetResponseForControllerResultEvent`*[23]

This event is not used by `FrameworkBundle`, but it can be used to implement a view sub-system. This event is called *only* if the Controller does *not* return a `Response` object. The purpose of the event is to allow some other return value to be converted into a `Response`.

The value returned by the Controller is accessible via the `getControllerResult` method:

Listing 18-6
```php
use Symfony\Component\HttpKernel\Event\GetResponseForControllerResultEvent;
use Symfony\Component\HttpFoundation\Response;

public function onKernelView(GetResponseForControllerResultEvent $event)
{
    $val = $event->getControllerResult();
    $response = new Response();
    // some how customize the Response from the return value

    $event->setResponse($response);
}
```

`kernel.response` Event

Event Class: *`FilterResponseEvent`*[24]

The purpose of this event is to allow other systems to modify or replace the `Response` object after its creation:

Listing 18-7
```php
public function onKernelResponse(FilterResponseEvent $event)
{
    $response = $event->getResponse();
    // .. modify the response object
}
```

The `FrameworkBundle` registers several listeners:

- *`ProfilerListener`*[25]: collects data for the current request;
- *`WebDebugToolbarListener`*[26]: injects the Web Debug Toolbar;
- *`ResponseListener`*[27]: fixes the Response `Content-Type` based on the request format;
- *`EsiListener`*[28]: adds a `Surrogate-Control` HTTP header when the Response needs to be parsed for ESI tags.

23. http://api.symfony.com/2.0/Symfony/Component/HttpKernel/Event/GetResponseForControllerResultEvent.html
24. http://api.symfony.com/2.0/Symfony/Component/HttpKernel/Event/FilterResponseEvent.html
25. http://api.symfony.com/2.0/Symfony/Component/HttpKernel/EventListener/ProfilerListener.html
26. http://api.symfony.com/2.0/Symfony/Bundle/WebProfilerBundle/EventListener/WebDebugToolbarListener.html
27. http://api.symfony.com/2.0/Symfony/Component/HttpKernel/EventListener/ResponseListener.html
28. http://api.symfony.com/2.0/Symfony/Component/HttpKernel/EventListener/EsiListener.html

`kernel.exception` Event

Event Class: *`GetResponseForExceptionEvent`*[29]

`FrameworkBundle` registers an *`ExceptionListener`*[30] that forwards the `Request` to a given Controller (the value of the `exception_listener.controller` parameter -- must be in the `class::method` notation).

A listener on this event can create and set a `Response` object, create and set a new `Exception` object, or do nothing:

Listing 18-8

```
use Symfony\Component\HttpKernel\Event\GetResponseForExceptionEvent;
use Symfony\Component\HttpFoundation\Response;

public function onKernelException(GetResponseForExceptionEvent $event)
{
    $exception = $event->getException();
    $response = new Response();
    // setup the Response object based on the caught exception
    $event->setResponse($response);

    // you can alternatively set a new Exception
    // $exception = new \Exception('Some special exception');
    // $event->setException($exception);
}
```

The Event Dispatcher

The event dispatcher is a standalone component that is responsible for much of the underlying logic and flow behind a Symfony request. For more information, see the *Event Dispatcher Component Documentation*.

Profiler

When enabled, the Symfony2 profiler collects useful information about each request made to your application and store them for later analysis. Use the profiler in the development environment to help you to debug your code and enhance performance; use it in the production environment to explore problems after the fact.

You rarely have to deal with the profiler directly as Symfony2 provides visualizer tools like the Web Debug Toolbar and the Web Profiler. If you use the Symfony2 Standard Edition, the profiler, the web debug toolbar, and the web profiler are all already configured with sensible settings.

The profiler collects information for all requests (simple requests, redirects, exceptions, Ajax requests, ESI requests; and for all HTTP methods and all formats). It means that for a single URL, you can have several associated profiling data (one per external request/response pair).

29. http://api.symfony.com/2.0/Symfony/Component/HttpKernel/Event/GetResponseForExceptionEvent.html
30. http://api.symfony.com/2.0/Symfony/Component/HttpKernel/EventListener/ExceptionListener.html

Visualizing Profiling Data

Using the Web Debug Toolbar

In the development environment, the web debug toolbar is available at the bottom of all pages. It displays a good summary of the profiling data that gives you instant access to a lot of useful information when something does not work as expected.

If the summary provided by the Web Debug Toolbar is not enough, click on the token link (a string made of 13 random characters) to access the Web Profiler.

If the token is not clickable, it means that the profiler routes are not registered (see below for configuration information).

Analyzing Profiling data with the Web Profiler

The Web Profiler is a visualization tool for profiling data that you can use in development to debug your code and enhance performance; but it can also be used to explore problems that occur in production. It exposes all information collected by the profiler in a web interface.

Accessing the Profiling information

You don't need to use the default visualizer to access the profiling information. But how can you retrieve profiling information for a specific request after the fact? When the profiler stores data about a Request, it also associates a token with it; this token is available in the `X-Debug-Token` HTTP header of the Response:

Listing 18-9
```php
$profile = $container->get('profiler')->loadProfileFromResponse($response);

$profile = $container->get('profiler')->loadProfile($token);
```

When the profiler is enabled but not the web debug toolbar, or when you want to get the token for an Ajax request, use a tool like Firebug to get the value of the `X-Debug-Token` HTTP header.

Use the `find()` method to access tokens based on some criteria:

Listing 18-10
```php
// get the latest 10 tokens
$tokens = $container->get('profiler')->find('', '', 10);

// get the latest 10 tokens for all URL containing /admin/
$tokens = $container->get('profiler')->find('', '/admin/', 10);

// get the latest 10 tokens for local requests
$tokens = $container->get('profiler')->find('127.0.0.1', '', 10);
```

If you want to manipulate profiling data on a different machine than the one where the information were generated, use the `export()` and `import()` methods:

Listing 18-11
```php
// on the production machine
$profile = $container->get('profiler')->loadProfile($token);
$data = $profiler->export($profile);

// on the development machine
$profiler->import($data);
```

Configuration

The default Symfony2 configuration comes with sensible settings for the profiler, the web debug toolbar, and the web profiler. Here is for instance the configuration for the development environment:

Listing 18-12

```
# load the profiler
framework:
    profiler: { only_exceptions: false }

# enable the web profiler
web_profiler:
    toolbar: true
    intercept_redirects: true
    verbose: true
```

When `only-exceptions` is set to `true`, the profiler only collects data when an exception is thrown by the application.

When `intercept-redirects` is set to `true`, the web profiler intercepts the redirects and gives you the opportunity to look at the collected data before following the redirect.

When `verbose` is set to `true`, the Web Debug Toolbar displays a lot of information. Setting `verbose` to `false` hides some secondary information to make the toolbar shorter.

If you enable the web profiler, you also need to mount the profiler routes:

Listing 18-13

```
_profiler:
    resource: @WebProfilerBundle/Resources/config/routing/profiler.xml
    prefix:   /_profiler
```

As the profiler adds some overhead, you might want to enable it only under certain circumstances in the production environment. The `only-exceptions` settings limits profiling to 500 pages, but what if you want to get information when the client IP comes from a specific address, or for a limited portion of the website? You can use a request matcher:

Listing 18-14

```
# enables the profiler only for request coming for the 192.168.0.0 network
framework:
    profiler:
        matcher: { ip: 192.168.0.0/24 }

# enables the profiler only for the /admin URLs
framework:
    profiler:
        matcher: { path: "^/admin/" }

# combine rules
framework:
    profiler:
        matcher: { ip: 192.168.0.0/24, path: "^/admin/" }

# use a custom matcher instance defined in the "custom_matcher" service
framework:
    profiler:
        matcher: { service: custom_matcher }
```

Learn more from the Cookbook

- *How to use the Profiler in a Functional Test*
- *How to create a custom Data Collector*
- *How to extend a Class without using Inheritance*

- *How to customize a Method Behavior without using Inheritance*

Chapter 19
The Symfony2 Stable API

The Symfony2 stable API is a subset of all Symfony2 published public methods (components and core bundles) that share the following properties:

- The namespace and class name won't change;
- The method name won't change;
- The method signature (arguments and return value type) won't change;
- The semantic of what the method does won't change.

The implementation itself can change though. The only valid case for a change in the stable API is in order to fix a security issue.

The stable API is based on a whitelist, tagged with *@api*. Therefore, everything not tagged explicitly is not part of the stable API.

Any third party bundle should also publish its own stable API.

As of Symfony 2.0, the following components have a public tagged API:

- BrowserKit
- ClassLoader
- Console
- CssSelector
- DependencyInjection
- DomCrawler
- EventDispatcher
- Finder
- HttpFoundation
- HttpKernel
- Locale
- Process
- Routing
- Templating
- Translation

- Validator
- Yaml