Symfony

The Book

Version: 3.0

generated on July 28, 2016

Contents at a Glance

Chapter 1
Symfony and HTTP Fundamentals

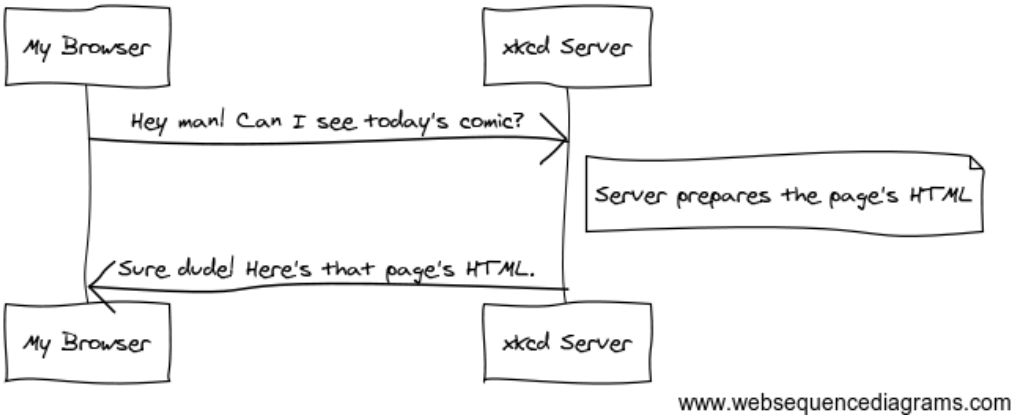
Congratulations! By learning about Symfony, you're well on your way towards being a more *productive*, *well-rounded* and *popular* web developer (actually, you're on your own for the last part). Symfony is built to get back to basics: to develop tools that let you develop faster and build more robust 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 independently of Symfony. So, get ready.

True to the Symfony 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/`



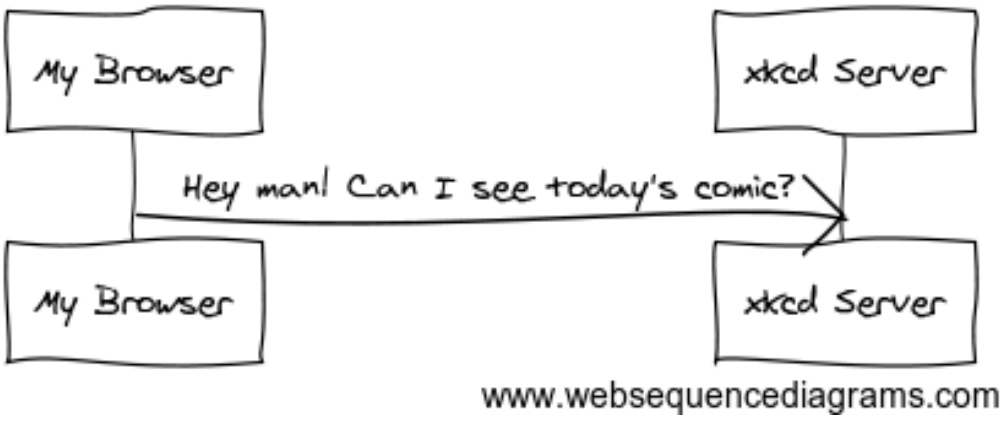
And while the actual language used is a bit more formal, it's still dead-simple. HTTP is the term used to describe this simple text-based language. No matter how you develop on the web, the goal of your server is *always* to understand simple text requests, and return simple text responses.

Symfony is built from the ground up around that reality. Whether you realize it or not, HTTP is something you use every day. With Symfony, 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, a smartphone 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, because it contains two important things: the HTTP method (GET) and the URL (`/`).

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 the client wants to *do* with the resource. The HTTP

methods (also known as verbs) define the few common ways that the client can act upon the resource - the most common HTTP methods are:

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:

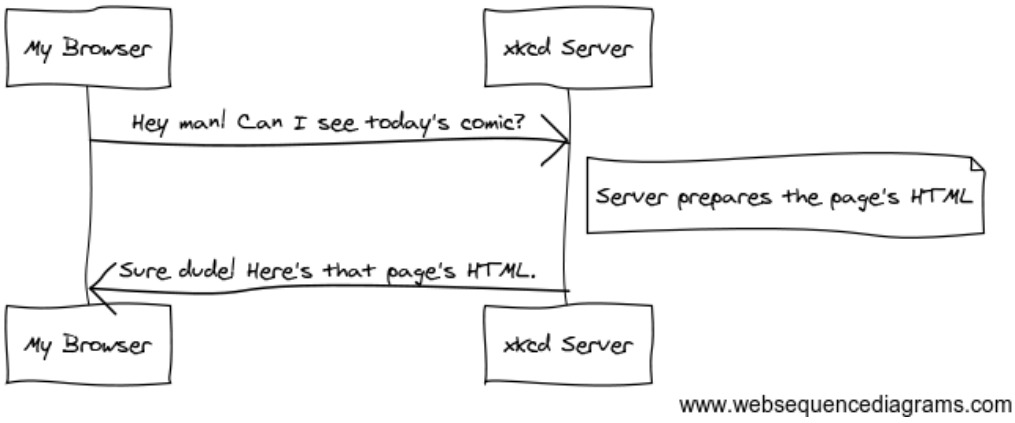
Listing 1-2
```
1 DELETE /blog/15 HTTP/1.1
```

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 only support `POST` and `GET` in HTML forms. Various others are however supported in *XMLHttpRequest*[2], as well as by Symfony's *Routing component*.

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 host of the resource being 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*[3] 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

2. https://en.wikipedia.org/wiki/XMLHttpRequest
3. https://en.wikipedia.org/wiki/List_of_HTTP_header_fields

```
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>
```

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*[4] article.

Like the request, an HTTP response contains additional pieces of information known as HTTP headers. 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. You can see a *List of common media types*[5] from IANA.

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*[6] or the *HTTP Bis*[7], 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*[8] 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
$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

4. https://en.wikipedia.org/wiki/List_of_HTTP_status_codes
5. https://www.iana.org/assignments/media-types/media-types.xhtml
6. http://www.w3.org/Protocols/rfc2616/rfc2616.html
7. http://datatracker.ietf.org/wg/httpbis/
8. https://addons.mozilla.org/en-US/firefox/addon/live-http-headers/

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 PHP `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:

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.

Symfony Request Object

The *Request*[9] 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

```
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).

9. http://api.symfony.com/3.0/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`*[10] object, which has methods like *`get()`*[11], *`has()`*[12], *`all()`*[13] 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 Symfony 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 Response Object

Symfony also provides a *`Response`*[14] 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

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

$response = new Response();

$response->setContent('<html><body><h1>Hello world!</h1></body></html>');
$response->setStatusCode(Response::HTTP_OK);

// set a HTTP response header
$response->headers->set('Content-Type', 'text/html');

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

There are also special classes to make certain types of responses easier to create:

- JsonResponse;
- BinaryFileResponse (for streaming files and sending file downloads);
- StreamedResponse (for streaming any other large responses);

The `Request` and `Response` classes are part of a standalone component called *symfony/http-foundation* that yo can use in *any* PHP project. This also contains classes for handling sessions, file uploads and more.

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*.

10. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/ParameterBag.html
11. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/ParameterBag.html#method_get
12. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/ParameterBag.html#method_has
13. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/ParameterBag.html#method_all
14. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Response.html

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.

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`

By using rewrite rules in your *web server configuration*, the `index.php` won't be needed and you will have beautiful, clean URLs (e.g. `/show`).

Now, every request is handled exactly the same way. 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

Inside your front controller, you have to figure out which code should be executed and what the content to return should be. To figure this out, you'll need to check the incoming URI and execute different parts of your code depending on that value. This can get ugly quickly:

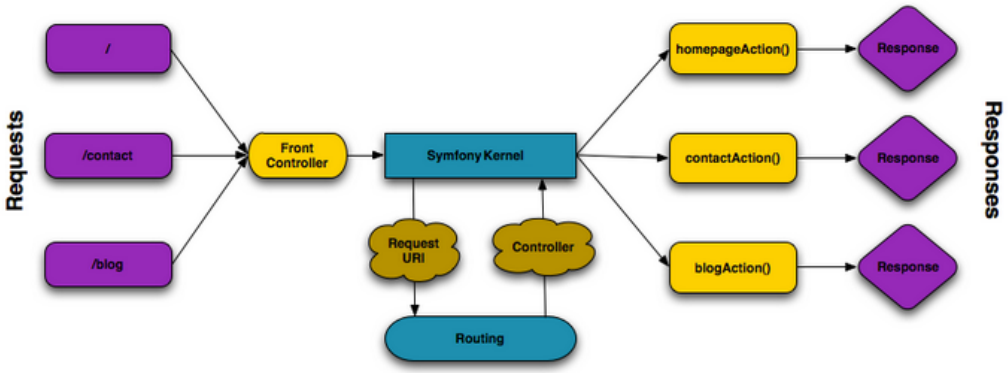
Listing 1-9
```
// index.php
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;

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

if (in_array($path, array('', '/'))) {
    $response = new Response('Welcome to the homepage.');
} elseif ('/contact' === $path) {
```

```
    $response = new Response('Contact us');
} else {
    $response = new Response('Page not found.', Response::HTTP_NOT_FOUND);
}
$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 component* and passed to PHP 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! To review:

- Each request executes the same, single file (called a "front controller");
- The front controller boots Symfony, and passes it request information;
- The router matches the incoming URL to a specific route and returns information about the route, including the controller (i.e. function) that should be executed;
- The controller (function) is executed: this is where *your* code creates and returns the appropriate `Response` object;
- The HTTP headers and content of the `Response` object are sent back to the client.

A Symfony Request in Action

Without diving into too much detail, here is 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
```
# app/config/routing.yml
contact:
    path:     /contact
    defaults: { _controller: AppBundle:Main:contact }
```

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 `AppBundle:Main:contact` string is a short syntax that points to a specific controller - `contactAction()` - inside a controller class called - `MainController`:

Listing 1-11
```php
// src/AppBundle/Controller/MainController.php
namespace AppBundle\Controller;

use Symfony\Component\HttpFoundation\Response;

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

In this example, the controller creates a *Response*[15] 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.

Symfony: 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 Symfony, 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 Symfony *Components*

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

HttpFoundation
: Contains the `Request` and `Response` classes, as well as other classes for handling sessions and file uploads.

Routing
: Powerful and fast routing system that allows you to map a specific URI (e.g. `/contact`) to information about how that request should be handled (e.g. that the `contactAction()` controller method should be executed).

Form
: A full-featured and flexible framework for creating forms and handling form submissions.

15. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Response.html

Validator[16]
: A system for creating rules about data and then validating whether or not user-submitted data follows those rules.

Templating
: A toolkit for rendering templates, handling template inheritance (i.e. a template is decorated with a layout) and performing other common template tasks.

Security
: A powerful library for handling all types of security inside an application.

Translation
: A framework for translating strings in your application.

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

The Full Solution: The Symfony *Framework*

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

1. Provides a selection of components (i.e. the Symfony Components) and third-party libraries (e.g. *Swift Mailer*[17] 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 Symfony bundle (i.e. a plugin) that can be configured or replaced entirely.

Symfony 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 Symfony distribution, which provides a project skeleton with sensible defaults. For more advanced users, the sky is the limit.

16. `https://github.com/symfony/validator`
17. `http://swiftmailer.org/`

Chapter 2
Symfony versus Flat PHP

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

If you've never used a PHP framework, aren't familiar with the *Model-View-Controller*[1] (MVC) philosophy, or just wonder what all the *hype* is around Symfony, this chapter is for you. Instead of *telling* you that Symfony 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 Symfony 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 = new PDO("mysql:host=localhost;dbname=blog_db", 'myuser', 'mypassword');

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

<!DOCTYPE html>
<html>
    <head>
        <title>List of Posts</title>
    </head>
    <body>
        <h1>List of Posts</h1>
        <ul>
            <?php while ($row = $result->fetch(PDO::FETCH_ASSOC)): ?>
            <li>
```

1. https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller

```
                <a href="/show.php?id=<?= $row['id'] ?>">
                    <?= $row['title'] ?>
                </a>
            </li>
            <?php endwhile ?>
        </ul>
    </body>
</html>

<?php
$link = null;
?>
```

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, Symfony fully integrates *Doctrine*[2], a library dedicated to database abstraction and mapping.

Isolating the Presentation

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

Listing 2-2

```
// index.php
$link = new PDO("mysql:host=localhost;dbname=blog_db", 'myuser', 'mypassword');

$result = $link->query('SELECT id, title FROM post');

$posts = array();
while ($row = $result->fetch(PDO::FETCH_ASSOC)) {
    $posts[] = $row;
}

$link = null;

// 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

```
<!-- templates/list.php -->
<!DOCTYPE html>
<html>
    <head>
        <title>List of Posts</title>
    </head>
    <body>
        <h1>List of Posts</h1>
        <ul>
            <?php foreach ($posts as $post): ?>
```

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

```
            <li>
                <a href="/show.php?id=<?= $post['id'] ?>">
                    <?= $post['title'] ?>
                </a>
            </li>
            <?php endforeach ?>
        </ul>
    </body>
</html>
```

By convention, the file that contains all 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, the 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 = new PDO("mysql:host=localhost;dbname=blog_db", 'myuser', 'mypassword');

    return $link;
}

function close_database_connection(&$link)
{
    $link = null;
}

function get_all_posts()
{
    $link = open_database_connection();

    $result = $link->query('SELECT id, title FROM post');

    $posts = array();
    while ($row = $result->fetch(PDO::FETCH_ASSOC)) {
        $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
// index.php
require_once 'model.php';

```

```
4  $posts = get_all_posts();
5
6  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 `templates/layout.php` file:

Listing 2-6

```
1  <!-- templates/layout.php -->
2  <!DOCTYPE html>
3  <html>
4      <head>
5          <title><?= $title ?></title>
6      </head>
7      <body>
8          <?= $content ?>
9      </body>
10 </html>
```

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

Listing 2-7

```
1  <!-- templates/list.php -->
2  <?php $title = 'List of Posts' ?>
3
4  <?php ob_start() ?>
5      <h1>List of Posts</h1>
6      <ul>
7          <?php foreach ($posts as $post): ?>
8          <li>
9              <a href="/show.php?id=<?= $post['id'] ?>">
10                 <?= $post['title'] ?>
11             </a>
12         </li>
13         <?php endforeach ?>
14     </ul>
15 <?php $content = ob_get_clean() ?>
16
17 <?php include 'layout.php' ?>
```

You now have a setup that will allow you to reuse the layout. Unfortunately, to accomplish this, you're forced to use a few ugly PHP functions (`ob_start()`, `ob_get_clean()`) in the template. Symfony 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();

    $query = 'SELECT created_at, title, body FROM post WHERE  id=:id';
    $statement = $link->prepare($query);
    $statement->bindValue(':id', $id, PDO::PARAM_INT);
    $statement->execute();

    $row = $statement->fetch(PDO::FETCH_ASSOC);

    close_database_connection($link);

    return $row;
}
```

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

Listing 2-9
```php
// show.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
<!-- templates/show.php -->
<?php $title = $post['title'] ?>

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

    <div class="date"><?= $post['created_at'] ?></div>
    <div class="body">
        <?= $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.

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)
```

By using rewrite rules in your *web server configuration*, the `index.php` won't be needed and you will have beautiful, clean URLs (e.g. `/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
$uri = parse_url($_SERVER['REQUEST_URI'], PHP_URL_PATH);
if ('/index.php' === $uri) {
    list_action();
} elseif ('/index.php/show' === $uri && isset($_GET['id'])) {
    show_action($_GET['id']);
} else {
    header('HTTP/1.1 404 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 named `controllers.php`:

Listing 2-13

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

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

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 how Symfony handles and routes requests.

But but careful not to confuse the terms *front controller* and *controller*. Your app will usually have just *one* front controller, which boots your code. You will have *many* controller functions: one for each page.

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 Symfony, 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 Symfony

Symfony to the rescue. Before actually using Symfony, you need to download it. This can be done by using *Composer*[3], which takes care of downloading the correct version and all its dependencies and provides an autoloader. An autoloader is a tool that makes it possible to start using PHP classes without explicitly including the file containing the class.

In your root directory, create a `composer.json` file with the following content:

Listing 2-14
```
{
    "require": {
        "symfony/symfony": "3.0.*"
    },
    "autoload": {
        "files": ["model.php","controllers.php"]
    }
}
```

Next, *download Composer*[4] and then run the following command, which will download Symfony into a `vendor/` directory:

Listing 2-15
```
$ composer install
```

Beside downloading your dependencies, Composer generates a `vendor/autoload.php` file, which takes care of autoloading for all the files in the Symfony Framework as well as the files mentioned in the autoload section of your `composer.json`.

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, Symfony provides both a *`Request`*[5] and a *`Response`*[6] 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:

Listing 2-16
```
// index.php
require_once 'vendor/autoload.php';

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

$request = Request::createFromGlobals();

$uri = $request->getPathInfo();
if ('/' === $uri) {
```

3. https://getcomposer.org
4. https://getcomposer.org/download/
5. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Request.html
6. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Response.html

```
    $response = list_action();
} elseif ('/show' === $uri && $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, Response::HTTP_NOT_FOUND);
}

// 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 Symfony templating engine:

Listing 2-17

```
// 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 Symfony, the application is more flexible and reliable. The `Request` provides a dependable way to access information about the HTTP request. Specifically, the *`getPathInfo()`*[7] 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 Symfony

The blog has come a *long* way, but it still contains a lot of code for such a simple application. Along the way, you've made 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,

7. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Request.html#method_getPathInfo

you could at least use Symfony's standalone *Routing* and *Templating* components, which already solve these problems.

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

Listing 2-18

```
// src/AppBundle/Controller/BlogController.php
namespace AppBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

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

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

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

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

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

Notice, both controller functions now live inside a "controller class". This is a nice way to group related pages. The controller functions are also sometimes called *actions*.

The two controllers (or actions) 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 `list.php` template is now quite a bit simpler:

Listing 2-19

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

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

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

The `layout.php` file is nearly identical:

Listing 2-20

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

The show `show.php` template is left as an exercise: updating it should be really similar to updating the `list.php` template.

When Symfony'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 - `app/config/routing.yml` - provides this information in a readable format:

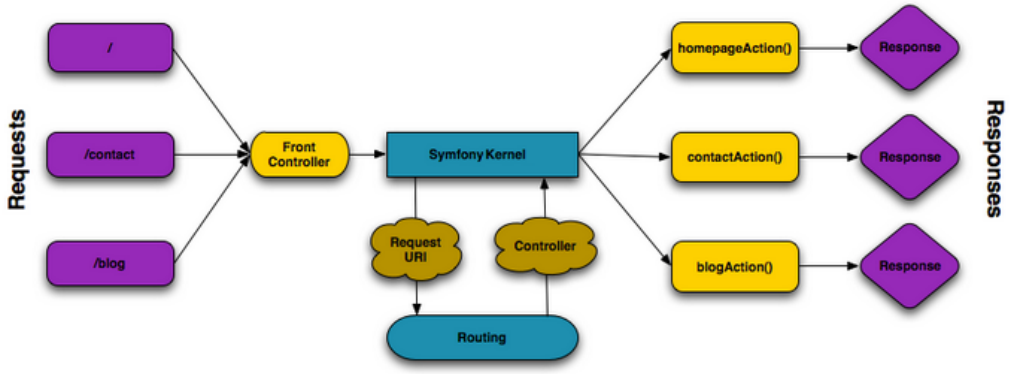
Listing 2-21
```yaml
# app/config/routing.yml
blog_list:
    path:     /blog
    defaults: { _controller: AppBundle:Blog:list }

blog_show:
    path:     /blog/show/{id}
    defaults: { _controller: AppBundle:Blog:show }
```

Now that Symfony is handling all the mundane tasks, the front controller `web/app.php` is dead simple. And since it does so little, you'll never have to touch it:

Listing 2-22
```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 Symfony's engine (called the Kernel) and pass it a `Request` object to handle. The Symfony core asks the router to inspect the request. The router matches the incoming URL to a specific route and returns information about the route, including the controller that should be executed. The correct controller from the matched route is executed and your code inside the controller creates and returns the appropriate `Response` object. The HTTP headers and content of the `Response` object are sent back to the client.

It's a beautiful thing.



Better Templates

If you choose to use it, Symfony comes standard with a templating engine called *Twig*[8] that makes templates faster to write and easier to read. It means that the sample application could contain even less code! Take, for example, rewriting `list.html.php` template in Twig would look like this:

Listing 2-23
```twig
{# app/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 %}
```

And rewriting `layout.html.php` template in Twig would look like this:

Listing 2-24
```twig
{# app/Resources/views/layout.html.twig #}
<!DOCTYPE html>
<html>
    <head>
        <title>{% block title %}Default title{% endblock %}</title>
    </head>
    <body>
        {% block body %}{% endblock %}
    </body>
</html>
```

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

Where Symfony Delivers

In the upcoming chapters, you'll learn more about how each piece of Symfony works and how you can organize your project. For now, celebrate at how migrating the blog from flat PHP to Symfony has improved life:

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

- 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*;
- Symfony gives you **access to open source tools** such as *Doctrine*[9] and the *Templating*, *Security*, *Form*, *Validator*[10] and *Translation* components (to name a few);
- The application now enjoys **fully-flexible URLs** thanks to the Routing component;
- Symfony's HTTP-centric architecture gives you access to powerful tools such as **HTTP caching** powered by **Symfony's internal HTTP cache** or more powerful tools such as *Varnish*[11]. This is covered in a later chapter all about *caching*.

And perhaps best of all, by using Symfony, you now have access to a whole set of **high-quality open source tools developed by the Symfony community**! A good selection of Symfony community tools can be found on *KnpBundles.com*[12].

Learn more from the Cookbook

- *How to Use PHP instead of Twig for Templates*
- *How to Define Controllers as Services*

9. `http://www.doctrine-project.org`
10. `https://github.com/symfony/validator`
11. `https://www.varnish-cache.org/`
12. `http://knpbundles.com/`

Chapter 3
Installing and Configuring Symfony

Welcome to Symfony! Starting a new Symfony project is easy. In fact, you'll have your first working Symfony application up and running in just a few short minutes.

> *Do you prefer video tutorials? Check out the Joyful Development with Symfony*[1] *screencast series from KnpUniversity.*

To make creating new applications even simpler, Symfony provides an installer. Downloading it is your first step.

Installing the Symfony Installer

Using the **Symfony Installer** is the only recommended way to create new Symfony applications. This installer is a PHP application that has to be installed in your system only once and then it can create any number of Symfony applications.

The installer requires PHP 5.4 or higher. If you still use the legacy PHP 5.3 version, you cannot use the Symfony Installer. Read the Creating Symfony Applications without the Installer section to learn how to proceed.

Depending on your operating system, the installer must be installed in different ways.

Linux and Mac OS X Systems

Open your command console and execute the following commands:

Listing 3-1
```
$ sudo curl -LsS https://symfony.com/installer -o /usr/local/bin/symfony
$ sudo chmod a+x /usr/local/bin/symfony
```

This will create a global `symfony` command in your system.

1. *http://knpuniversity.com/screencast/symfony*

Windows Systems

Open your command console and execute the following command:

Listing 3-2
```
c:\> php -r "readfile('https://symfony.com/installer');" > symfony
```

Then, move the downloaded `symfony` file to your project's directory and execute it as follows:

Listing 3-3
```
c:\> move symfony c:\projects
c:\projects\> php symfony
```

Creating the Symfony Application

Once the Symfony Installer is available, create your first Symfony application with the `new` command:

Listing 3-4
```
# Linux, Mac OS X
$ symfony new my_project_name

# Windows
c:\> cd projects/
c:\projects\> php symfony new my_project_name
```

This command creates a new directory called `my_project_name/` that contains a fresh new project based on the most recent stable Symfony version available. In addition, the installer checks if your system meets the technical requirements to execute Symfony applications. If not, you'll see the list of changes needed to meet those requirements.

For security reasons, all Symfony versions are digitally signed before distributing them. If you want to verify the integrity of any Symfony version, follow the steps *explained in this post*[2].

If the installer doesn't work for you or doesn't output anything, make sure that the PHP *Phar extension*[3] is installed and enabled on your computer.

Basing your Project on a Specific Symfony Version

In case your project needs to be based on a specific Symfony version, use the optional second argument of the `new` command:

Listing 3-5
```
# use the most recent version in any Symfony branch
$ symfony new my_project_name 2.8
$ symfony new my_project_name 3.0

# use a specific Symfony version
$ symfony new my_project_name 2.7.3
$ symfony new my_project_name 2.8.1

# use a beta or RC version (useful for testing new Symfony versions)
$ symfony new my_project 3.0.0-BETA1
$ symfony new my_project 2.7.0-RC1
```

The installer also supports a special version called `lts` which installs the most recent Symfony LTS version available:

2. http://fabien.potencier.org/signing-project-releases.html
3. http://php.net/manual/en/intro.phar.php

Listing 3-6
```
$ symfony new my_project_name lts
```

Read the *Symfony Release process* to better understand why there are several Symfony versions and which one to use for your projects.

Creating Symfony Applications without the Installer

If you still use PHP 5.3, or if you can't execute the installer for any reason, you can create Symfony applications using the alternative installation method based on *Composer*[4].

Composer is the dependency manager used by modern PHP applications and it can also be used to create new applications based on the Symfony Framework. If you don't have it installed globally, start by reading the next section.

Installing Composer Globally

Start with *installing Composer globally*.

Creating a Symfony Application with Composer

Once Composer is installed on your computer, execute the `create-project` Composer command to create a new Symfony application based on its latest stable version:

Listing 3-7
```
$ composer create-project symfony/framework-standard-edition my_project_name
```

If you need to base your application on a specific Symfony version, provide that version as the second argument of the `create-project` Composer command:

Listing 3-8
```
$ composer create-project symfony/framework-standard-edition my_project_name "3.0.*"
```

If your Internet connection is slow, you may think that Composer is not doing anything. If that's your case, add the `-vvv` flag to the previous command to display a detailed output of everything that Composer is doing.

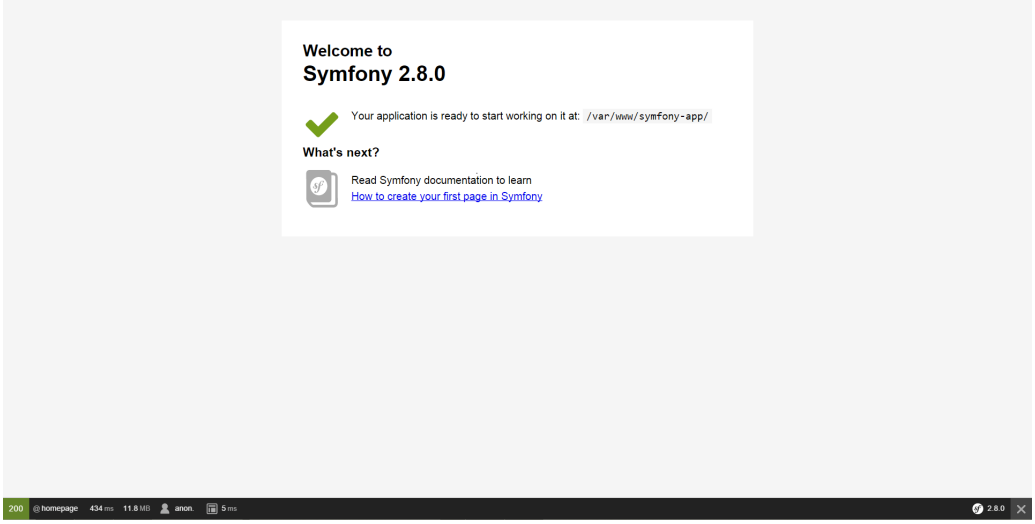
Running the Symfony Application

Symfony leverages the internal web server provided by PHP to run applications while developing them. Therefore, running a Symfony application is a matter of browsing the project directory and executing this command:

Listing 3-9
```
$ cd my_project_name/
$ php bin/console server:run
```

Then, open your browser and access the `http://localhost:8000/` URL to see the Welcome Page of Symfony:

4. https://getcomposer.org/



Instead of the Welcome Page, you may see a blank page or an error page. This is caused by a directory permission misconfiguration. There are several possible solutions depending on your operating system. All of them are explained in the Setting up Permissions section of this chapter.

PHP's internal web server is great for developing, but should **not** be used on production. Instead, use Apache or Nginx. See *Configuring a Web Server*.

PHP's internal web server is available in PHP 5.4 or higher versions.

When you are finished working on your Symfony application, you can stop the server by pressing *Ctrl+C* from terminal.

Checking Symfony Application Configuration and Setup

Symfony applications come with a visual server configuration tester to show if your environment is ready to use Symfony. Access the following URL to check your configuration:

Listing 3-10

```
1 http://localhost:8000/config.php
```

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

Setting up Permissions

One common issue when installing Symfony is that the `var` directory 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 who owns the files, you can try one of the following solutions.

1. Use the same user for the CLI and the web server

In development environments, it is a common practice to use the same UNIX user for the CLI and the web server because it avoids any of these permissions issues when setting up new projects. This can be done by editing your web server configuration (e.g. commonly httpd.conf or apache2.conf for Apache) and setting its user to be the same as your CLI user (e.g. for Apache, update the `User` and `Group` values).

If used in a production environment, be sure this user only has limited privileges (no access to private data or servers, launch of unsafe binaries, etc.) as a compromised server would give to the hacker those privileges.

2. Using ACL on a system that supports chmod +a (MacOS X)

MacOS X allows you to use the `chmod +a` command. This uses a command to try to determine your web server user and set it as `HTTPDUSER`:

Listing 3-11
```
$ rm -rf var/cache/* var/logs/* var/sessions/*

$ HTTPDUSER=`ps axo user,comm | grep -E '[a]pache|[h]ttpd|[_]www|[w]ww-data|[n]ginx' | grep -v root |
head -1 | cut -d\  -f1`
$ sudo chmod -R +a "$HTTPDUSER allow delete,write,append,file_inherit,directory_inherit" var
$ sudo chmod -R +a "`whoami` allow delete,write,append,file_inherit,directory_inherit" var
```

3. Using ACL on a system that supports setfacl (most Linux/BSD)

Most Linux and BSD distributions 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. This uses a command to try to determine your web server user and set it as `HTTPDUSER`:

Listing 3-12
```
$ HTTPDUSER=`ps axo user,comm | grep -E '[a]pache|[h]ttpd|[_]www|[w]ww-data|[n]ginx' | grep -v root |
head -1 | cut -d\  -f1`
$ sudo setfacl -R -m u:"$HTTPDUSER":rwX -m u:`whoami`:rwX var
$ sudo setfacl -dR -m u:"$HTTPDUSER":rwX -m u:`whoami`:rwX var
```

If this doesn't work, try adding `-n` option.

setfacl isn't available on NFS mount points. However, setting cache and logs over NFS is strongly not recommended for performance.

4. Without using ACL

If none of the previous methods work for you, 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 `bin/console`, `web/app.php` and `web/app_dev.php` files:

Listing 3-13
```
umask(0002); // This will let the permissions be 0775

// or

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

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

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

Updating Symfony Applications

At this point, you've created a fully-functional Symfony application in which you'll start to develop your own project. A Symfony application depends on a number of external libraries. These are downloaded into the `vendor/` directory and they are managed exclusively by Composer.

Updating those third-party libraries frequently is a good practice to prevent bugs and security vulnerabilities. Execute the `update` Composer command to update them all at once:

Listing 3-14
```
$ cd my_project_name/
$ composer update
```

Depending on the complexity of your project, this update process can take up to several minutes to complete.

Symfony provides a command to check whether your project's dependencies contain any known security vulnerability:

Listing 3-15
```
$ php bin/console security:check
```

A good security practice is to execute this command regularly to be able to update or replace compromised dependencies as soon as possible.

Installing the Symfony Demo Application

The Symfony Demo application is a fully-functional application that shows the recommended way to develop Symfony applications. The application has been conceived as a learning tool for Symfony newcomers and its source code contains tons of comments and helpful notes.

In order to download the Symfony Demo application, execute the `demo` command of the Symfony Installer anywhere in your system:

Listing 3-16
```
# Linux, Mac OS X
$ symfony demo

# Windows
c:\projects\> php symfony demo
```

Once downloaded, enter into the `symfony_demo/` directory and run the PHP's built-in web server executing the `php bin/console server:run` command. Access to the `http://localhost:8000` URL in your browser to start using the Symfony Demo application.

Installing a Symfony Distribution

Symfony project packages "distributions", which are fully-functional applications that include the Symfony core libraries, a selection of useful bundles, a sensible directory structure and some default configuration. In fact, when you created a Symfony application in the previous sections, you actually downloaded the default distribution provided by Symfony, which is called *Symfony Standard Edition*[6].

The Symfony Standard Edition is by far the most popular distribution and it's also the best choice for developers starting with Symfony. However, the Symfony Community has published other popular distributions that you may use in your applications:

- The *Symfony CMF Standard Edition*[7] is the best distribution to get started with the *Symfony CMF*[8] project, which is a project that makes it easier for developers to add CMS functionality to applications built with the Symfony Framework.
- The *Symfony REST Edition*[9] shows how to build an application that provides a RESTful API using the *FOSRestBundle*[10] and several other related bundles.

Using Source Control

If you're using a version control system like *Git*[11], you can safely commit all your project's code. The reason is that Symfony applications already contain a `.gitignore` file specially prepared for Symfony.

For specific instructions on how best to set up your project to be stored in Git, see *How to Create and Store a Symfony Project in Git*.

Checking out a versioned Symfony Application

When using Composer to manage application's dependencies, it's recommended to ignore the entire `vendor/` directory before committing its code to the repository. This means that when checking out a Symfony application from a Git repository, there will be no `vendor/` directory and the application won't work out-of-the-box.

In order to make it work, check out the Symfony application and then execute the `install` Composer command to download and install all the dependencies required by the application:

Listing 3-17
```
$ cd my_project_name/
$ composer install
```

How does Composer know which specific dependencies to install? Because when a Symfony application is committed to a repository, the `composer.json` and `composer.lock` files are also committed. These files tell Composer which dependencies (and which specific versions) to install for the application.

Beginning Development

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

If you're new to Symfony, check out "*Create your First Page in Symfony*", where you'll learn how to create pages, change configuration, and do everything else you'll need in your new application.

Be sure to also check out the *Cookbook*, which contains a wide variety of articles about solving specific problems with Symfony.

---

6. https://github.com/symfony/symfony-standard
7. https://github.com/symfony-cmf/symfony-cmf-standard
8. http://cmf.symfony.com/
9. https://github.com/gimler/symfony-rest-edition
10. https://github.com/FriendsOfSymfony/FOSRestBundle
11. http://git-scm.com/

Chapter 4
Create your First Page in Symfony

Creating a new page - whether it's an HTML page or a JSON endpoint - is a simple two-step process:

1. *Create a route*: A route is the URL (e.g. `/about`) to your page and points to a controller;
2. *Create a controller*: A controller is the PHP function you write that builds the page. You take the incoming request information and use it to create a Symfony `Response` object, which can hold HTML content, a JSON string or even a binary file like an image or PDF. The only rule is that a controller *must* return a Symfony Response object (and you'll even learn to bend this rule eventually).

Just like on the web, every interaction is initiated by an HTTP request. Your job is pure and simple: understand that request and return a response.

> *Do you prefer video tutorials? Check out the Joyful Development with Symfony*[1] *screencast series from KnpUniversity.*

Creating a Page: Route and Controller

Before continuing, make sure you've read the *Installation* chapter and can access your new Symfony app in the browser.

Suppose you want to create a page - `/lucky/number` - that generates a lucky (well, random) number and prints it. To do that, create a "Controller class" and a "controller" method inside of it that will be executed when someone goes to `/lucky/number`:

Listing 4-1
```php
// src/AppBundle/Controller/LuckyController.php
namespace AppBundle\Controller;

use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Component\HttpFoundation\Response;

class LuckyController
{
```

1. *http://knpuniversity.com/screencast/symfony/first-page*

```
    /**
     * @Route("/lucky/number")
     */
    public function numberAction()
    {
        $number = rand(0, 100);

        return new Response(
            '<html><body>Lucky number: '.$number.'</body></html>'
        );
    }
}
```

Before diving into this, test it out! If you are using PHP's internal web server go to:

> *http://localhost:8000/lucky/number*

If you set up a virtual host in Apache or Nginx replace `http://localhost:8000` with your host name and add `app_dev.php` to make sure Symfony loads in the "dev" environment:

> *http://symfony.dev/app_dev.php/lucky/number*

If you see a lucky number being printed back to you, congratulations! But before you run off to play the lottery, check out how this works.

The `@Route` above `numberAction()` is called an *annotation* and it defines the URL pattern. You can also write routes in YAML (or other formats): read about this in the *routing* chapter. Actually, most routing examples in the docs have tabs that show you how each format looks.

The method below the annotation - `numberAction` - is called the *controller* and is where you build the page. The only rule is that a controller *must* return a Symfony Response object (and you'll even learn to bend this rule eventually).

Creating a JSON Response

The `Response` object you return in your controller can contain HTML, JSON or even a binary file like an image or PDF. You can easily set HTTP headers or the status code.

Suppose you want to create a JSON endpoint that returns the lucky number. Just add a second method to `LuckyController`:

Listing 4-2
```
// src/AppBundle/Controller/LuckyController.php

// ...
class LuckyController
{
    // ...

    /**
     * @Route("/api/lucky/number")
     */
    public function apiNumberAction()
    {
        $data = array(
            'lucky_number' => rand(0, 100),
        );

        return new Response(
            json_encode($data),
```

```
            200,
            array('Content-Type' => 'application/json')
        );
    }
}
```

Try this out in your browser:

> *http://localhost:8000/api/lucky/number*

You can even shorten this with the handy *JsonResponse*[2]:

Listing 4-3

```
// src/AppBundle/Controller/LuckyController.php

// ...
// --> don't forget this new use statement
use Symfony\Component\HttpFoundation\JsonResponse;

class LuckyController
{
    // ...

    /**
     * @Route("/api/lucky/number")
     */
    public function apiNumberAction()
    {
        $data = array(
            'lucky_number' => rand(0, 100),
        );

        // calls json_encode() and sets the Content-Type header
        return new JsonResponse($data);
    }
}
```

Dynamic URL Patterns: /lucky/number/{count}

Woh, you're doing great! But Symfony's routing can do a lot more. Suppose now that you want a user to be able to go to `/lucky/number/5` to generate 5 lucky numbers at once. Update the route to have a `{wildcard}` part at the end:

Listing 4-4

```
// src/AppBundle/Controller/LuckyController.php

// ...
class LuckyController
{
    /**
     * @Route("/lucky/number/{count}")
     */
    public function numberAction()
    {
        // ...
    }

    // ...
}
```

2. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/JsonResponse.html

Because of the `{count}` "wildcard" placeholder, the URL to the page is *different*: it now works for URLs matching `/lucky/number/*` - for example `/lucky/number/5`. The best part is that you can access this value and use it in your controller:

Listing 4-5

```
// src/AppBundle/Controller/LuckyController.php
// ...

class LuckyController
{

    /**
     * @Route("/lucky/number/{count}")
     */
    public function numberAction($count)
    {
        $numbers = array();
        for ($i = 0; $i < $count; $i++) {
            $numbers[] = rand(0, 100);
        }
        $numbersList = implode(', ', $numbers);

        return new Response(
            '<html><body>Lucky numbers: '.$numbersList.'</body></html>'
        );
    }

    // ...
}
```

Try it by printing 7 lucky numbers:

> *http://localhost:8000/lucky/number/7*

You can get the value of any ``{placeholder}`` in your route by adding a ``$placeholder`` argument to your controller. Just make sure that the placeholder (e.g. ``{id}``) matches the argument name (e.g. ``$id``).

The routing system can do a *lot* more, like supporting multiple placeholders (e.g. `/blog/{category}/{page}`), making placeholders optional and forcing placeholder to match a regular expression (e.g. so that `{count}` *must* be a number). Find out about all of this and become a routing expert in the *Routing* chapter.

Rendering a Template (with the Service Container)

If you're returning HTML from your controller, you'll probably want to render a template. Fortunately, Symfony comes with *Twig*[3]: a templating language that's easy, powerful and actually quite fun.

So far, `LuckyController` doesn't extend any base class. The easiest way to use Twig - or many other tools in Symfony - is to extend Symfony's base *`Controller`*[4] class:

Listing 4-6

```
// src/AppBundle/Controller/LuckyController.php

// ...
// --> add this new use statement
use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class LuckyController extends Controller
```

3. http://twig.sensiolabs.org
4. http://api.symfony.com/3.0/Symfony/Bundle/FrameworkBundle/Controller/Controller.html

```
{
    // ...
}
```

Using the `templating` Service

This doesn't change anything, but it *does* give you access to Symfony's *service container*: an array-like object that gives you access to *every* useful object in the system. These useful objects are called *services*, and Symfony ships with a service object that can render Twig templates, another that can log messages and many more.

To render a Twig template, use a service called `templating`:

Listing 4-7

```
// src/AppBundle/Controller/LuckyController.php

// ...
class LuckyController extends Controller
{
    /**
     * @Route("/lucky/number/{count}")
     */
    public function numberAction($count)
    {
        // ...
        $numbersList = implode(', ', $numbers);

        $html = $this->container->get('templating')->render(
            'lucky/number.html.twig',
            array('luckyNumberList' => $numbersList)
        );

        return new Response($html);
    }

    // ...
}
```

You'll learn a lot more about the important "service container" as you keep reading. For now, you just need to know that it holds a lot of objects, and you can *`get()`*[5] any object by using its nickname, like `templating` or `logger`. The `templating` service is an instance of *`TwigEngine`*[6] and this has a *`render()`*[7] method.

But this can get even easier! By extending the `Controller` class, you also get a lot of shortcut methods, like *`render()`*[8]:

Listing 4-8

```
// src/AppBundle/Controller/LuckyController.php

// ...
/**
 * @Route("/lucky/number/{count}")
 */
public function numberAction($count)
{
    // ...

    /*
    $html = $this->container->get('templating')->render(
        'lucky/number.html.twig',
```

5. http://api.symfony.com/3.0/Symfony/Bundle/FrameworkBundle/Controller/Controller.html#method_get
6. http://api.symfony.com/3.0/Symfony/Bundle/TwigBundle/TwigEngine.html
7. http://api.symfony.com/3.0/Symfony/Bundle/TwigBundle/TwigEngine.html#method_render
8. http://api.symfony.com/3.0/Symfony/Bundle/FrameworkBundle/Controller/Controller.html#method_render

```
14          array('luckyNumberList' => $numbersList)
15      );
16
17      return new Response($html);
18      */
19
20      // render(): a shortcut that does the same as above
21      return $this->render(
22          'lucky/number.html.twig',
23          array('luckyNumberList' => $numbersList)
24      );
25  }
```

You will learn more about these shortcut methods and how they work in the *Controller* chapter.

Create the Template

If you refresh your browser now, you'll get an error:

> Unable to find template "lucky/number.html.twig"

Fix that by creating a new `app/Resources/views/lucky` directory and putting a `number.html.twig` file inside of it:

Listing 4-9

```
1  {# app/Resources/views/lucky/number.html.twig #}
2  {% extends 'base.html.twig' %}
3
4  {% block body %}
5      <h1>Lucky Numbers: {{ luckyNumberList }}</h1>
6  {% endblock %}
```

Welcome to Twig! This simple file already shows off the basics:

- The `{{ variableName }}` syntax is used to print something. In this template, `luckyNumberList` is a variable that you're passing into the template from the `render` call in the controller.
- The `{% extends 'base.html.twig' %}` points to a layout file that lives at *app/Resources/views/base.html.twig*[9] and came with your new project. It's *really* basic (an unstyled HTML structure) and it's yours to customize.
- The `{% block body %}` part uses Twig's inheritance system to put the content into the middle of the `base.html.twig` layout.

Refresh to see your template in action!

> *http://localhost:8000/lucky/number/7*

If you view the source code of the displayed page, you now have a basic HTML structure thanks to `base.html.twig`.

This is just the surface of Twig's power. When you're ready to master its syntax, loop over arrays, render other templates and other cool things, read the *Templating* chapter.

9. `https://github.com/symfony/symfony-standard/blob/2.7/app/Resources/views/base.html.twig`

Exploring the Project

You've already created a flexible URL, rendered a template that uses inheritance and created a JSON endpoint. Nice!

It's time to explore and demystify the files in your project. You've already worked inside the two most important directories:

`app/`
: Contains things like configuration and templates. Basically, anything that is *not* PHP code goes here.

`src/`
: Your PHP code lives here.

99% of the time, you'll be working in `src/` (PHP files) or `app/` (everything else). As you get more advanced, you'll learn what can be done inside each of these.

The `app/` directory also holds some other things, like `app/AppKernel.php`, which you'll use to enable new bundles (this is one of a *very* short list of PHP files in `app/`).

The `src/` directory has just one directory - `src/AppBundle` - and everything lives inside of it. A bundle is like a "plugin" and you can *find open source bundles*[10] and install them into your project. But even *your* code lives in a bundle - typically *AppBundle* (though there's nothing special about AppBundle). To find out more about bundles and why you might create multiple bundles (hint: sharing code between projects), see the *Bundles* chapter.

So what about the other directories in the project?

`web/`
: This is the document root for the project and contains any publicly accessible files, like CSS, images and the Symfony development and production front controllers that execute the app (`app_dev.php` and `app.php`).

`tests/`
: The automatic tests (e.g. Unit tests) of your application live here.

`bin/`
: The "binary" files live here. The most important one is the `console` file which is used to execute Symfony commands via the console.

`var/`
: This is where automatically created files are stored, like cache files (`var/cache/`) and logs (`var/logs/`).

`vendor/`
: Third-party (i.e. "vendor") libraries live here! These are typically downloaded via the *Composer*[11] package manager.

> *Symfony is flexible. If you need to, you can easily override the default directory structure. See How to Override Symfony's default Directory Structure.*

Application Configuration

Symfony comes with several built-in bundles (open your `app/AppKernel.php` file) and you'll probably install more. The main configuration file for bundles is `app/config/config.yml`:

10. `http://knpbundles.com`
11. `https://getcomposer.org`

Listing 4-10
```yaml
# app/config/config.yml

# ...
framework:
    secret: '%secret%'
    router:
        resource: '%kernel.root_dir%/config/routing.yml'
    # ...

twig:
    debug:            '%kernel.debug%'
    strict_variables: '%kernel.debug%'

# ...
```

The `framework` key configures FrameworkBundle, the `twig` key configures TwigBundle and so on. A *lot* of behavior in Symfony can be controlled just by changing one option in this configuration file. To find out how, see the *Configuration Reference* section.

Or, to get a big example dump of all of the valid configuration under a key, use the handy `bin/console` command:

Listing 4-11
```
$ php bin/console config:dump-reference framework
```

There's a lot more power behind Symfony's configuration system, including environments, imports and parameters. To learn all of it, see the *Configuring Symfony (and Environments)* chapter.

What's Next?

Congrats! You're already starting to master Symfony and learn a whole new way of building beautiful, functional, fast and maintainable apps.

Ok, time to finish mastering the fundamentals by reading these chapters:

- *Controller*
- *Routing*
- *Creating and Using Templates*

Then, in the *Symfony Book*, learn about the *service container*, the *form system*, using *Doctrine* (if you need to query a database) and more!

There's also a *Cookbook packed* with more advanced "how to" articles to solve *a lot* of problems.

Have fun!

Chapter 5
Controller

A controller is a PHP callable you create that takes information from the HTTP request and creates and returns an HTTP response (as a Symfony `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.

See how simple this is by looking at a Symfony controller in action. This renders a page that prints the famous `Hello world!`:

Listing 5-1
```php
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:

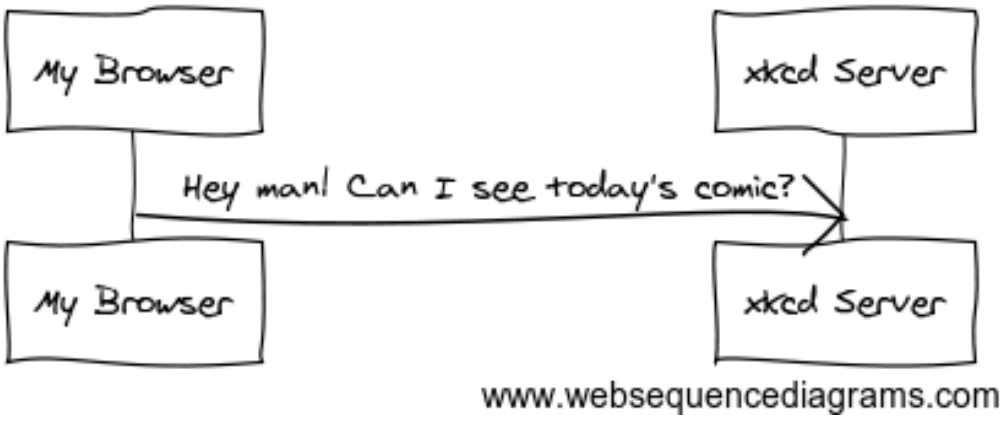
- *Controller A* prepares a `Response` object representing the content for the homepage of the site.
- *Controller B* reads the `{slug}` placeholder from the request to load a blog entry from the database and creates a `Response` object displaying that blog. If the `{slug}` can't be found in the database, 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 you. 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 Symfony project goes through the same simple lifecycle. The framework takes care of all the repetitive stuff: you just need to write your custom code in the controller function:

1. Each request executes a single front controller file (e.g. `app.php` on production or `app_dev.php` on development) that bootstraps the application;
2. The front controller's only job is to initialize Symfony's engine (called the `Kernel`) and pass it a `Request` object to handle;
3. The Symfony core asks the router to inspect the request;
4. The router matches the incoming URL to a specific route and returns information about the route, including the controller that should be executed;
5. The correct controller from the matched route is executed and the code inside the controller creates and returns the appropriate `Response` object;
6. 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 (#5) and making a route that maps a URL to that controller (#4).



Though similarly named, a "front controller" is different from the PHP functions called "controllers" talked about in this chapter. A front controller is a short PHP file that lives in your `web/` directory 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. The "controller class" is a convenient way to group several "controllers", also called actions, together in one class (e.g. `updateAction()`, `deleteAction()`, etc). So, a controller is a method inside a controller class. They hold your code which creates and returns the appropriate `Response` object.

A Simple Controller

While a controller can be any PHP callable (a function, method on an object, or a `Closure`), a controller is usually a method inside a controller class:

Listing 5-2

```
1  // src/AppBundle/Controller/HelloController.php
2  namespace AppBundle\Controller;
3
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 }
```

The controller is the `indexAction()` method, which lives inside a controller class `HelloController`.

This controller is pretty straightforward:

- *line 2*: Symfony takes advantage of PHP's namespace functionality to namespace the entire controller class.
- *line 4*: Symfony again takes advantage of PHP's namespace functionality: the `use` keyword imports the `Response` class, which the 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 (e.g. `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 controller 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 path to the controller:

Listing 5-3

```
1  // src/AppBundle/Controller/HelloController.php
2  namespace AppBundle\Controller;
3
4  use Symfony\Component\HttpFoundation\Response;
5  use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
6
7  class HelloController
8  {
9      /**
10      * @Route("/hello/{name}", name="hello")
11      */
12     public function indexAction($name)
13     {
14         return new Response('<html><body>Hello '.$name.'!</body></html>');
15     }
16 }
```

Now, you can go to `/hello/ryan` (e.g. `http://localhost:8000/hello/ryan` if you're using the *built-in web server*) and Symfony will execute the `HelloController::indexAction()` controller and pass in `ryan` for the `$name` variable. Creating a "page" means simply creating a controller method and an associated route.

Simple, right?

The AppBundle:Hello:index controller syntax

If you use the YAML or XML formats, you'll refer to the controller using a special shortcut syntax called the *logical controller name* which, for example, looks like `AppBundle:Hello:index`. For more details on the controller format, read Controller Naming Pattern subtitle of the Routing chapter.

Route Parameters as Controller Arguments

You already know that the route points to the `HelloController::indexAction()` controller method that lives inside AppBundle. What's more interesting is the argument that is passed to that controller method:

Listing 5-4
```php
// src/AppBundle/Controller/HelloController.php
// ...
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;

/**
 * @Route("/hello/{name}", name="hello")
 */
public function indexAction($name)
{
    // ...
}
```

The controller has a single argument, `$name`, which corresponds to the `{name}` placeholder from the matched route (e.g. `ryan` if you go to `/hello/ryan`). When executing the controller, Symfony matches each argument with a placeholder from the route. So the value for `{name}` is passed to `$name`. Just make sure that the name of the placeholder is the same as the name of the argument variable.

Take the following more-interesting example, where the controller has two arguments:

Listing 5-5
```php
// src/AppBundle/Controller/HelloController.php
// ...

use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;

class HelloController
{
    /**
     * @Route("/hello/{firstName}/{lastName}", name="hello")
     */
    public function indexAction($firstName, $lastName)
    {
        // ...
    }
}
```

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

1. **The order of the controller arguments does not matter**

   Symfony matches the parameter **names** from the route to the variable **names** of the controller. The arguments of the controller could be totally reordered and still work perfectly:

   Listing 5-6
   ```php
   public function indexAction($lastName, $firstName)
   {
       // ...
   }
   ```

2. **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 5-7
```
public function indexAction($firstName, $lastName, $foo)
{
    // ...
}
```

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

Listing 5-8
```
public function indexAction($firstName, $lastName, $foo = 'bar')
{
    // ...
}
```

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

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

Listing 5-9
```
public function indexAction($firstName)
{
    // ...
}
```

You can also pass other variables from your route to your controller arguments. See *How to Pass Extra Information from a Route to a Controller*.

The Base Controller Class

For convenience, Symfony comes with an optional base *`Controller`*[1] class. If you extend it, this won't change anything about how your controller works, but you'll get access to a number of **helper methods** and the **service container** (see Accessing other Services): an array-like object that gives you access to every useful object in the system. These useful objects are called **services**, and Symfony ships with a service object that can render Twig templates, another that can log messages and many more.

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

Listing 5-10
```
1 // src/AppBundle/Controller/HelloController.php
2 namespace AppBundle\Controller;
3
4 use Symfony\Bundle\FrameworkBundle\Controller\Controller;
5
6 class HelloController extends Controller
7 {
8     // ...
9 }
```

Helper methods are just shortcuts to using core Symfony 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`*[2] class.

Generating URLs

The *`generateUrl()`*[3] method is just a helper method that generates the URL for a given route.

1. http://api.symfony.com/3.0/Symfony/Bundle/FrameworkBundle/Controller/Controller.html
2. http://api.symfony.com/3.0/Symfony/Bundle/FrameworkBundle/Controller/Controller.html
3. http://api.symfony.com/3.0/Symfony/Bundle/FrameworkBundle/Controller/Controller.html#method_generateUrl

Redirecting

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

Listing 5-11
```
public function indexAction()
{
    return $this->redirectToRoute('homepage');

    // redirectToRoute is equivalent to using redirect() and generateUrl() together:
    // return $this->redirect($this->generateUrl('homepage'));
}
```

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

Listing 5-12
```
public function indexAction()
{
    return $this->redirectToRoute('homepage', array(), 301);
}
```

To redirect to an *external* site, use `redirect()` and pass it the external URL:

Listing 5-13
```
public function indexAction()
{
    return $this->redirect('http://symfony.com/doc');
}
```

For more information, see the *Routing chapter*.

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

Listing 5-14
```
use Symfony\Component\HttpFoundation\RedirectResponse;

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

Rendering Templates

If you're serving HTML, you'll want to render a template. The `render()` method renders a template **and** puts that content into a `Response` object for you:

Listing 5-15
```
// renders app/Resources/views/hello/index.html.twig
return $this->render('hello/index.html.twig', array('name' => $name));
```

Templates can also live in deeper sub-directories. Just try to avoid creating unnecessarily deep structures:

Listing 5-16
```
// renders app/Resources/views/hello/greetings/index.html.twig
return $this->render('hello/greetings/index.html.twig', array(
    'name' => $name
));
```

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. To learn how to render different templating formats read the Template Formats section of the Creating and Using Templates chapter.

The Symfony templating engine is explained in great detail in the *Creating and Using Templates chapter*.

Templating Naming Pattern

You can also put templates in the `Resources/views` directory of a bundle and reference them with a special shortcut syntax like `@App/Hello/index.html.twig` or `@App/layout.html.twig`. These would live in at `Resources/views/Hello/index.html.twig` and `Resources/views/layout.html.twig` inside the bundle respectively.

Accessing other Services

Symfony comes packed with a lot of useful objects, called services. These are used for rendering templates, sending emails, querying the database and any other "work" you can think of. When you install a new bundle, it probably brings in even *more* services.

When extending the base controller class, you can access any Symfony service via the *`get()`*[4] method of the `Controller` class. Here are several common services you might need:

Listing 5-17
```
$templating = $this->get('templating');

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

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

What other services exist? To list all services, use the `debug:container` console command:

Listing 5-18
```
$ php bin/console debug:container
```

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

To get a container configuration parameter in controller you can use the *`getParameter()`*[5] method:

Listing 5-19
```
$from = $this->getParameter('app.mailer.from');
```

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 5-20
```
public function indexAction()
{
    // retrieve the object from database
    $product = ...;
    if (!$product) {
        throw $this->createNotFoundException('The product does not exist');
    }

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

4. http://api.symfony.com/3.0/Symfony/Bundle/FrameworkBundle/Controller/Controller.html#method_get
5. http://api.symfony.com/3.0/Symfony/Bundle/FrameworkBundle/Controller/Controller.html#method_getParameter

The *createNotFoundException()*[6] method is just a shortcut to create a special *NotFoundHttpException*[7] object, which ultimately triggers a 404 HTTP response inside Symfony.

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

Listing 5-21
```php
throw new \Exception('Something went wrong!');
```

In every case, an error page is shown to the end user and a full debug error page is shown to the developer (i.e. when you're using the `app_dev.php` front controller - see Environments).

You'll want to customize the error page your user sees. To do that, see the "*How to Customize Error Pages*" cookbook recipe.

The Request object as a Controller Argument

What if you need to read query parameters, grab a request header or get access to an uploaded file? All of that information is stored in Symfony's `Request` object. To get it in your controller, just add it as an argument and **type-hint it with the ``Request`` class**:

Listing 5-22
```php
use Symfony\Component\HttpFoundation\Request;

public function indexAction($firstName, $lastName, Request $request)
{
    $page = $request->query->get('page', 1);

    // ...
}
```

Managing the Session

Symfony 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, Symfony stores the attributes in a cookie by using the native PHP sessions.

To retrieve the session, call *getSession()*[8] method on the `Request` object. This method returns a *SessionInterface*[9] with easy methods for storing and fetching things from the session:

Listing 5-23
```php
use Symfony\Component\HttpFoundation\Request;

public function indexAction(Request $request)
{
    $session = $request->getSession();

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

    // get the attribute set by another controller in another request
    $foobar = $session->get('foobar');

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

6. http://api.symfony.com/3.0/Symfony/Bundle/FrameworkBundle/Controller/Controller.html#method_createNotFoundException
7. http://api.symfony.com/3.0/Symfony/Component/HttpKernel/Exception/NotFoundHttpException.html
8. http://api.symfony.com/3.0/Symfony/Bundle/FrameworkBundle/Controller/Controller.html#method_getSession
9. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/SessionInterface.html

Stored attributes remain in the session for the remainder of that user's session.

Flash Messages

You can also store special messages, called "flash" messages, on the user's session. By design, flash messages are meant to be used exactly once: they vanish from the session automatically as soon as you retrieve them. This feature makes "flash" messages particularly great for storing user notifications.

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

Listing 5-24
```php
use Symfony\Component\HttpFoundation\Request;

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

    $form->handleRequest($request);

    if ($form->isValid()) {
        // do some sort of processing

        $this->addFlash(
            'notice',
            'Your changes were saved!'
        );

        // $this->addFlash is equivalent to $this->get('session')->getFlashBag()->add

        return $this->redirectToRoute(...);
    }

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

After processing the request, the controller sets a flash message in the session and then redirects. The message key (`notice` in this example) can be anything: you'll use this key to retrieve the message.

In the template of the next page (or even better, in your base layout template), read any flash messages from the session:

Listing 5-25
```twig
{% for flash_message in app.session.flashBag.get('notice') %}
    <div class="flash-notice">
        {{ flash_message }}
    </div>
{% endfor %}
```

It's common to use `notice`, `warning` and `error` as the keys of the different types of flash messages, but you can use any key that fits your needs.

You can use the *peek()*[10] method instead to retrieve the message while keeping it in the bag.

The Request and Response Object

As mentioned earlier, the framework will pass the `Request` object to any controller argument that is type-hinted with the `Request` class:

10. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Flash/FlashBagInterface.html#method_peek

Listing 5-26
```php
use Symfony\Component\HttpFoundation\Request;

public function indexAction(Request $request)
{
    $request->isXmlHttpRequest(); // is it an Ajax request?

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

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

    // 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');
}
```

The `Request` class has several public properties and methods that return any information you need about the request.

Like the `Request`, the `Response` object has also a public `headers` property. This is a *`ResponseHeaderBag`*[11] that has some nice methods for getting and setting response headers. The header names are normalized so that using `Content-Type` is equivalent to `content-type` or even `content_type`.

The only requirement for a controller is to return a `Response` object. The *`Response`*[12] class is an abstraction around the HTTP response - the text-based message filled with headers and content that's sent back to the client:

Listing 5-27
```php
use Symfony\Component\HttpFoundation\Response;

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

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

There are also special classes to make certain kinds of responses easier:

- For JSON, there is *`JsonResponse`*[13]. See Creating a JSON Response.
- For files, there is *`BinaryFileResponse`*[14]. See Serving Files.
- For streamed responses, there is *`StreamedResponse`*[15]. See Streaming a Response.

Now that you know the basics you can continue your research on Symfony `Request` and `Response` object in the HttpFoundation component documentation.

11. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/ResponseHeaderBag.html
12. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Response.html
13. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/JsonResponse.html
14. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/BinaryFileResponse.html
15. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/StreamedResponse.html

Creating Static Pages

You can create a static page without even creating a controller (only a route and template are needed). See cookbook article *How to Render a Template without a custom Controller*.

Forwarding to Another Controller

Though not very common, you can also forward to another controller internally with the *`forward()`*[16] method. Instead of redirecting the user's browser, this makes an "internal" sub-request and calls the defined controller. The `forward()` method returns the `Response` object that's returned from *that* controller:

Listing 5-28
```php
public function indexAction($name)
{
    $response = $this->forward('AppBundle:Something:fancy', array(
        'name'  => $name,
        'color' => 'green',
    ));

    // ... further modify the response or return it directly

    return $response;
}
```

The array passed to the method becomes the arguments for the resulting controller. The target controller method might look something like this:

Listing 5-29
```php
public function fancyAction($name, $color)
{
    // ... create and return a Response object
}
```

Just like when creating a controller for a route, the order of the arguments of `fancyAction()` doesn't matter: the matching is done by name.

Validating a CSRF Token

Sometimes, you want to use CSRF protection in an action where you don't want to use the Symfony Form component. If, for example, you're doing a DELETE action, you can use the *`isCsrfTokenValid()`*[17] method to check the CSRF token:

Listing 5-30
```php
if ($this->isCsrfTokenValid('token_id', $submittedToken)) {
    // ... do something, like deleting an object
}

// isCsrfTokenValid() is equivalent to:
// $this->get('security.csrf.token_manager')->isTokenValid(
//     new \Symfony\Component\Security\Csrf\CsrfToken\CsrfToken('token_id', $token)
// );
```

16. http://api.symfony.com/3.0/Symfony/Bundle/FrameworkBundle/Controller/Controller.html#method_forward
17. http://api.symfony.com/3.0/Symfony/Bundle/FrameworkBundle/Controller/Controller.html#method_isCsrfTokenValid

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 where you can do anything 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.

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 Symfony 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 path 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

```
// src/AppBundle/Controller/BlogController.php
namespace AppBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;

class BlogController extends Controller
{
    /**
     * @Route("/blog/{slug}", name="blog_show")
     */
    public function showAction($slug)
    {
        // ...
    }
}
```

The path 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 `blog_show` is the internal name of the route, which doesn't have any meaning yet and just needs to be unique. Later, you'll use it to generate URLs.

If you don't want to use annotations, because you don't like them or because you don't want to depend on the SensioFrameworkExtraBundle, you can also use Yaml, XML or PHP. In these formats, 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, in this case the `AppBundle\Controller\BlogController::showAction` method.

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 Symfony 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

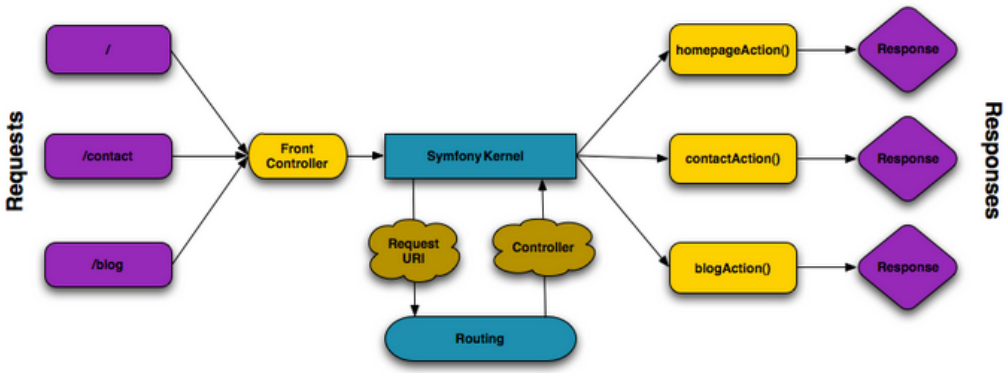
When a request is made to your application, it contains an address to the exact "resource" that the client is requesting. This address is called the URL, (or URI), and could be `/contact`, `/blog/read-me`, or anything else. Take the following HTTP request for example:

Listing 6-2

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

The goal of the Symfony 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 Symfony front controller (e.g. `app.php`);
2. The Symfony 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 Symfony 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 6-3
```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. To do so, just point out in the main routing configuration file which external files should be included. 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 `path` to match and a `defaults` array:

Listing 6-4
```php
// src/AppBundle/Controller/MainController.php

// ...
class MainController extends Controller
{
    /**
     * @Route("/")
     */
    public function homepageAction()
    {
        // ...
    }
}
```

This route matches the homepage (`/`) and maps it to the `AppBundle:Main:homepage` controller. The `_controller` string is translated by Symfony 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 6-5
```php
// src/AppBundle/Controller/BlogController.php

// ...
class BlogController extends Controller
{
    /**
     * @Route("/blog/{slug}")
     */
    public function showAction($slug)
    {
        // ...
    }
}
```

The path 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 path 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 6-6
```php
// src/AppBundle/Controller/BlogController.php

// ...
class BlogController extends Controller
{
    // ...

    /**
     * @Route("/blog")
     */
    public function indexAction()
    {
        // ...
    }
}
```

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 6-7
```php
// src/AppBundle/Controller/BlogController.php

// ...

/**
 * @Route("/blog/{page}")
 */
public function indexAction($page)
{
    // ...
}
```

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 6-8
```php
// src/AppBundle/Controller/BlogController.php

// ...

/**
 * @Route("/blog/{page}", defaults={"page" = 1})
 */
public function indexAction($page)
{
    // ...
}
```

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.

URL	Route	Parameters
/blog	blog	{page} = 1
/blog/1	blog	{page} = 1
/blog/2	blog	{page} = 2

Of course, you can have more than one optional placeholder (e.g. `/blog/{slug}/{page}`), but everything after an optional placeholder must be optional. For example, `/{page}/blog` is a valid path, but `page` will always be required (i.e. simply `/blog` will not match this route).

Routes with optional parameters at the end will not match on requests with a trailing slash (i.e. `/blog/` will not match, `/blog` will match).

Adding Requirements

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

Listing 6-9

```
// src/AppBundle/Controller/BlogController.php

// ...
class BlogController extends Controller
{
    /**
     * @Route("/blog/{page}", defaults={"page" = 1})
     */
    public function indexAction($page)
    {
        // ...
    }

    /**
     * @Route("/blog/{slug}")
     */
    public function showAction($slug)
    {
        // ...
    }
}
```

Can you spot the problem? Notice that both routes have patterns that match URLs 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* or route *conditions* (see Completely Customized Route Matching with Conditions). The routes in this example would work perfectly if the `/blog/`

`{page}` path *only* matched URLs where the `{page}` portion is an integer. Fortunately, regular expression requirements can easily be added for each parameter. For example:

Listing 6-10

```
// src/AppBundle/Controller/BlogController.php

// ...

/**
 * @Route("/blog/{page}", defaults={"page": 1}, requirements={
 *     "page": "\d+"
 * })
 */
public function indexAction($page)
{
    // ...
}
```

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
/blog/2-my-blog-post	blog_show	{slug} = 2-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 6-11

```
// src/AppBundle/Controller/MainController.php

// ...
class MainController extends Controller
{
    /**
     * @Route("/{_locale}", defaults={"_locale": "en"}, requirements={
     *     "_locale": "en|fr"
     * })
     */
    public function homepageAction($_locale)
    {
    }
}
```

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

Path	Parameters
`/`	`{_locale} = "en"`
`/en`	`{_locale} = "en"`
`/fr`	`{_locale} = "fr"`
`/es`	*won't match this route*

The route requirements can also include container parameters, as explained in *this article*. This comes in handy when the regular expression is very complex and used repeatedly in your application.

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 create an API for your blog and you have 2 routes: One for displaying a post (on a GET or HEAD request) and one for updating a post (on a PUT request). This can be accomplished with the following route configuration:

Listing 6-12

```
// src/AppBundle/Controller/MainController.php
namespace AppBundle\Controller;

use Sensio\Bundle\FrameworkExtraBundle\Configuration\Method;
// ...

class BlogApiController extends Controller
{
    /**
     * @Route("/api/posts/{id}")
     * @Method({"GET","HEAD"})
     */
    public function showAction($id)
    {
        // ... return a JSON response with the post
    }

    /**
     * @Route("/api/posts/{id}")
     * @Method("PUT")
     */
    public function editAction($id)
    {
        // ... edit a post
    }
}
```

Despite the fact that these two routes have identical paths (`/api/posts/{id}`), the first route will match only GET or HEAD requests and the second route will match only PUT requests. This means that you can display and edit the post with the same URL, while using distinct controllers for the two actions.

If no `methods` are specified, the route will match on *all* methods.

Adding a Host Requirement

You can also match on the HTTP *host* of the incoming request. For more information, see *How to Match a Route Based on the Host* in the Routing component documentation.

Completely Customized Route Matching with Conditions

As you've seen, a route can be made to match only certain routing wildcards (via regular expressions), HTTP methods, or host names. But the routing system can be extended to have an almost infinite flexibility using `conditions`:

Listing 6-13
```
contact:
    path:     /contact
    defaults: { _controller: AcmeDemoBundle:Main:contact }
    condition: "context.getMethod() in ['GET', 'HEAD'] and request.headers.get('User-Agent') matches '/firefox/i'"
```

The `condition` is an expression, and you can learn more about its syntax here: *The Expression Syntax*. With this, the route won't match unless the HTTP method is either GET or HEAD *and* if the `User-Agent` header matches `firefox`.

You can do any complex logic you need in the expression by leveraging two variables that are passed into the expression:

`context`
: An instance of *`RequestContext`*[1], which holds the most fundamental information about the route being matched.

`request`
: The Symfony *`Request`*[2] object (see Request).

Conditions are *not* taken into account when generating a URL.

Expressions are Compiled to PHP

Behind the scenes, expressions are compiled down to raw PHP. Our example would generate the following PHP in the cache directory:

Listing 6-14
```
if (rtrim($pathinfo, '/contact') === '' && (
    in_array($context->getMethod(), array(0 => "GET", 1 => "HEAD"))
    && preg_match("/firefox/i", $request->headers->get("User-Agent"))
)) {
    // ...
}
```

Because of this, using the `condition` key causes no extra overhead beyond the time it takes for the underlying PHP to execute.

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 6-15
```
// src/AppBundle/Controller/ArticleController.php

// ...
class ArticleController extends Controller
{
    /**
     * @Route(
```

1. http://api.symfony.com/3.0/Symfony/Component/Routing/RequestContext.html
2. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Request.html

```
8         *     "/articles/{_locale}/{year}/{title}.{_format}",
9         *     defaults={"_format": "html"},
10        *     requirements={
11        *         "_locale": "en|fr",
12        *         "_format": "html|rss",
13        *         "year": "\d+"
14        *     }
15        * )
16        */
17       public function showAction($_locale, $year, $title)
18       {
19       }
20   }
```

As you've seen, this route will only match if the `{_locale}` 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 dot between placeholders instead of a slash. URLs matching this route might look like:

- `/articles/en/2010/my-post`
- `/articles/fr/2010/my-post.rss`
- `/articles/en/2013/my-latest-post.html`

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 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.

In Symfony versions previous to 3.0, it is possible to override the request format by adding a query parameter named `_format` (for example: `/foo/bar?_format=json`). Relying on this behavior not only is considered a bad practice but it will complicate the upgrade of your applications to Symfony 3.

Sometimes you want to make certain parts of your routes globally configurable. Symfony provides you with a way to do this by leveraging service container parameters. Read more about this in "*How to Use Service Container Parameters in your Routes*".

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 request (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 `AppBundle:Blog:show` means:

Bundle	Controller Class	Method Name
AppBundle	BlogController	showAction

The controller might look like this:

Listing 6-16

```
// src/AppBundle/Controller/BlogController.php
namespace AppBundle\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: `AppBundle\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 6-17

```
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:

- `$_locale`
- `$year`
- `$title`
- `$_format`
- `$_controller`
- `$_route`

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.

The special `$_route` variable is set to the name of the route that was matched.

You can even add extra information to your route definition and access it within your controller. For more information on this topic, see *How to Pass Extra Information from a Route to a Controller*.

Including External Routing Resources

All routes are loaded via a single configuration file - usually `app/config/routing.yml` (see Creating Routes above). However, if you use routing annotations, you'll need to point the router to the controllers with the annotations. This can be done by "importing" directories into the routing configuration:

Listing 6-18
```yaml
# app/config/routing.yml
app:
    resource: '@AppBundle/Controller/'
    type:     annotation # required to enable the Annotation reader for this resource
```

When importing resources from YAML, the key (e.g. `app`) 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 a directory, where the `@AppBundle` shortcut syntax resolves to the full path of the AppBundle. When pointing to a directory, all files in that directory are parsed and put into the routing.

You can also include other routing configuration files, this is often used to import the routing of third party bundles:

Listing 6-19
```yaml
# app/config/routing.yml
app:
    resource: '@AcmeOtherBundle/Resources/config/routing.yml'
```

Prefixing Imported Routes

You can also choose to provide a "prefix" for the imported routes. For example, suppose you want to prefix all routes in the AppBundle with `/site` (e.g. `/site/blog/{slug}` instead of `/blog/{slug}`):

Listing 6-20

```yaml
# app/config/routing.yml
app:
    resource: '@AppBundle/Controller/'
    type:     annotation
    prefix:   /site
```

The path of each route being loaded from the new routing resource will now be prefixed with the string `/site`.

Adding a Host Requirement to Imported Routes

You can set the host regex on imported routes. For more information, see Using Host Matching of Imported Routes.

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 `debug:router` console command. Execute the command by running the following from the root of your project.

Listing 6-21
```
$ php bin/console debug:router
```

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

Listing 6-22
```
homepage             ANY       /
contact              GET       /contact
contact_process      POST      /contact
article_show         ANY       /articles/{_locale}/{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 6-23
```
$ php bin/console debug:router article_show
```

Likewise, if you want to test whether a URL matches a given route, you can use the `router:match` console command:

Listing 6-24
```
$ php bin/console router:match /blog/my-latest-post
```

This command will print which route the URL matches.

Listing 6-25
```
Route "blog_show" matches
```

Generating URLs

The routing system should also be used to generate URLs. In reality, routing is a bidirectional system: mapping the URL to a controller+parameters and a route+parameters back to a URL. The *`match()`*[3]

3. http://api.symfony.com/3.0/Symfony/Component/Routing/Router.html#method_match

and *generate()*[4] methods form this bidirectional system. Take the `blog_show` example route from earlier:

Listing 6-26
```
$params = $this->get('router')->match('/blog/my-blog-post');
// array(
//     'slug'        => 'my-blog-post',
//     '_controller' => 'AppBundle:Blog:show',
// )

$uri = $this->get('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 path for that route. With this information, any URL can easily be generated:

Listing 6-27
```
class MainController extends Controller
{
    public function showAction($slug)
    {
        // ...

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

The `generateUrl()` method defined in the base *Controller*[5] class is just a shortcut for this code:

Listing 6-28
```
$url = $this->container->get('router')->generate(
    'blog_show',
    array('slug' => 'my-blog-post')
);
```

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

If the front-end 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*[6], you can do exactly that:

Listing 6-29
```
var url = Routing.generate(
    'blog_show',
    {'slug': 'my-blog-post'}
);
```

For more information, see the documentation for that bundle.

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:

4. http://api.symfony.com/3.0/Symfony/Component/Routing/Router.html#method_generate
5. http://api.symfony.com/3.0/Symfony/Bundle/FrameworkBundle/Controller/Controller.html
6. https://github.com/FriendsOfSymfony/FOSJsRoutingBundle

Listing 6-30
```php
$this->get('router')->generate('blog', array(
    'page' => 2,
    'category' => 'Symfony'
));
// /blog/2?category=Symfony
```

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 the `path()` function to generate a relative URL:

Listing 6-31
```twig
<a href="{{ path('blog_show', {'slug': 'my-blog-post'}) }}">
  Read this blog post.
</a>
```

If you are generating the route inside a `<script>` element, it's a good practice to escape it for JavaScript:

Listing 6-32
```twig
<script>
var route = "{{ path('blog_show', {'slug': 'my-blog-post'})|escape('js') }}";
</script>
```

Generating Absolute URLs

By default, the router will generate relative URLs (e.g. `/blog`). From a controller, simply pass `UrlGeneratorInterface::ABSOLUTE_URL` to the third argument of the `generateUrl()` method:

Listing 6-33
```php
use Symfony\Component\Routing\Generator\UrlGeneratorInterface;

$this->generateUrl('blog_show', array('slug' => 'my-blog-post'), UrlGeneratorInterface::ABSOLUTE_URL);
// http://www.example.com/blog/my-blog-post
```

From a template, simply use the `url()` function (which generates an absolute URL) rather than the `path()` function (which generates a relative URL):

Listing 6-34
```twig
<a href="{{ url('blog_show', {'slug': 'my-blog-post'}) }}">
  Read this blog post.
</a>
```

The host that's used when generating an absolute URL is automatically detected using the current `Request` object. When generating absolute URLs from outside the web context (for instance in a console command) this doesn't work. See *How to Generate URLs from the Console* to learn how to solve this problem.

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 your application decoupled from those URLs. Routing is a bidirectional 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*
- *How to Allow a "/" Character in a Route Parameter*
- *How to Configure a Redirect without a custom Controller*
- *How to Use HTTP Methods beyond GET and POST in Routes*
- *How to Use Service Container Parameters in your Routes*
- *How to Create a custom Route Loader*
- *Redirect URLs with a Trailing Slash*
- *How to Pass Extra Information from a Route to a Controller*

Chapter 7
Creating and Using Templates

As you know, the *controller* is responsible for handling each request that comes into a Symfony application. In reality, the controller delegates 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.

How to render templates is covered in the controller page of the book.

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 Symfony 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 three 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.

`{# ... #}`
: "Comment something": it's the equivalent of the PHP `/* comment */` syntax. It's used to add single or multi-line comments. The content of the comments isn't included in the rendered pages.

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 %}
```

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

---

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

If you *do* choose to not use Twig and you disable it, you'll need to implement your own exception handler via the `kernel.exception` event.

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 whitespace control, sandboxing, automatic HTML escaping, manual contextual output escaping, 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 if user.active %}
        <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 `var/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 Symfony, this problem is thought about 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 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
```
{# app/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`). This path is relative to the `app/Resources/views` directory of the project. You could also use the logical name equivalent: `::base.html.twig`. 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 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 Symfony 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 application's `views` directory can contain application-wide base templates (i.e. your application's layouts and templates of the application bundle) as well as templates that override third party bundle templates (see Overriding Bundle Templates).

path/to/bundle/Resources/views/

Each third party bundle houses its templates in its `Resources/views/` directory (and subdirectories). When you plan to share your bundle, you should put the templates in the bundle instead of the `app/` directory.

Most of the templates you'll use live in the `app/Resources/views/` directory. The path you'll use will be relative to this directory. For example, to render/extend `app/Resources/views/base.html.twig`, you'll use the `base.html.twig` path and to render/extend `app/Resources/views/blog/index.html.twig`, you'll use the `blog/index.html.twig` path.

Referencing Templates in a Bundle

Symfony uses a **bundle**:**directory**:**filename** string syntax for templates that live inside a bundle. This allows for several 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`: (*directory*) indicates that the template lives inside the `Blog` subdirectory of `Resources/views`;
  - `index.html.twig`: (*filename*) 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, "directory", portion is missing (e.g. `Blog`), the template lives at `Resources/views/layout.html.twig` inside AcmeBlogBundle. Yes, there are 2 colons in the middle of the string when the "controller" subdirectory part is missing.

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 similar to the naming convention used to refer to Controller Naming Pattern.

Template Suffix

Every template name also has two extensions that specify the *format* and *engine* for that template.

Filename	Format	Engine
`blog/index.html.twig`	HTML	Twig
`blog/index.html.php`	HTML	PHP
`blog/index.css.twig`	CSS	Twig

By default, any Symfony 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 Symfony 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 Template Formats 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.

Symfony 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.

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

Including other Templates

You'll often want to include the same template or code fragment on several 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

```twig
{# app/Resources/views/article/article_details.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

```twig
{# app/Resources/views/article/list.html.twig #}
{% extends 'layout.html.twig' %}

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

    {% for article in articles %}
        {{ include('article/article_details.html.twig', { 'article': article }) }}
    {% endfor %}
{% endblock %}
```

The template is included using the `{{ include() }}` function. Notice that the template name follows the same typical convention. The `article_details.html.twig` template uses an `article` variable, which we pass to it. In this case, you could avoid doing this entirely, as all of the variables available in `list.html.twig` are also available in `article_details.html.twig` (unless you set *with_context*[5] to false).

5. http://twig.sensiolabs.org/doc/functions/include.html

The `{'article': article}` syntax is the standard Twig syntax for hash maps (i.e. an array with named keys). If you 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/AppBundle/Controller/ArticleController.php
namespace AppBundle\Controller;

// ...

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(
            'article/recent_list.html.twig',
            array('articles' => $articles)
        );
    }
}
```

The `recent_list` template is perfectly straightforward:

Listing 7-13

```
{# app/Resources/views/article/recent_list.html.twig #}
{% for article in articles %}
    <a href="/article/{{ article.slug }}">
        {{ article.title }}
    </a>
{% endfor %}
```

Notice that the article URL is hardcoded 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(controller(
        'AppBundle:Article:recentArticles',
        { '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. Of course, like all controllers, they should ideally be "skinny", meaning that as much code as possible lives in reusable *services*.

Asynchronous Content with hinclude.js

Controllers can be embedded asynchronously using the *hinclude.js*[6] JavaScript library. As the embedded content comes from another page (or controller for that matter), Symfony uses a version of the standard `render` function to configure `hinclude` tags:

Listing 7-15
```
{{ render_hinclude(controller('...')) }}
{{ render_hinclude(url('...')) }}
```

hinclude.js[7] needs to be included in your page to work.

When using a controller instead of a URL, you must enable the Symfony `fragments` configuration:

Listing 7-16
```
# app/config/config.yml
framework:
    # ...
    fragments: { path: /_fragment }
```

Default content (while loading or if JavaScript is disabled) can be set globally in your application configuration:

Listing 7-17
```
# app/config/config.yml
framework:
    # ...
    templating:
        hinclude_default_template: hinclude.html.twig
```

You can define default templates per `render` function (which will override any global default template that is defined):

Listing 7-18
```
{{ render_hinclude(controller('...'),  {
    'default': 'default/content.html.twig'
}) }}
```

Or you can also specify a string to display as the default content:

Listing 7-19
```
{{ render_hinclude(controller('...'), {'default': 'Loading...'}) }}
```

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:

---

6. http://mnot.github.io/hinclude/
7. http://mnot.github.io/hinclude/

Listing 7-20
```php
// src/AppBundle/Controller/WelcomeController.php

// ...
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;

class WelcomeController extends Controller
{
    /**
     * @Route("/", name="_welcome")
     */
    public function indexAction()
    {
        // ...
    }
}
```

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

Listing 7-21
```twig
<a href="{{ path('_welcome') }}">Home</a>
```

As expected, this will generate the URL `/`. Now, for a more complicated route:

Listing 7-22
```php
// src/AppBundle/Controller/ArticleController.php

// ...
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;

class ArticleController extends Controller
{
    /**
     * @Route("/article/{slug}", name="article_show")
     */
    public function showAction($slug)
    {
        // ...
    }
}
```

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

Listing 7-23
```twig
{# app/Resources/views/article/recent_list.html.twig #}
{% for article in articles %}
    <a href="{{ path('article_show', {'slug': article.slug}) }}">
        {{ article.title }}
    </a>
{% endfor %}
```

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

Listing 7-24
```twig
<a href="{{ url('_welcome') }}">Home</a>
```

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 Symfony provides a more dynamic option via the `asset` Twig function:

Listing 7-25

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

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

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`), then the rendered paths should be `/images/logo.png`. But if your application lives in a subdirectory (e.g. `http://example.com/my_app`), 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 loaded from cache after being deployed. For example, `/images/logo.png` might look like `/images/logo.png?v2`. For more information, see the version configuration option.

If you need absolute URLs for assets, use the `absolute_url()` Twig function as follows:

Listing 7-26
```twig
<img src="{{ absolute_url(asset('images/logo.png')) }}" alt="Symfony!" />
```

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 is also compatible with 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 7-27
```twig
{# app/Resources/views/base.html.twig #}
<html>
    <head>
        {# ... #}

        {% block stylesheets %}
            <link href="{{ asset('css/main.css') }}" rel="stylesheet" />
        {% endblock %}
    </head>
    <body>
        {# ... #}

        {% block javascripts %}
            <script src="{{ asset('js/main.js') }}"></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 7-28

```twig
{# app/Resources/views/contact/contact.html.twig #}
{% extends 'base.html.twig' %}

{% block stylesheets %}
    {{ parent() }}

    <link href="{{ asset('css/contact.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 bin/console assets:install target [--symlink]` command, which moves (or symlinks) files into the correct location. (target is by default "web").

Listing 7-29
```twig
<link href="{{ asset('bundles/acmedemo/css/contact.css') }}" rel="stylesheet" />
```

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

Global Template Variables

During each request, Symfony will set a global template variable `app` in both Twig and PHP template engines by default. The `app` variable is a *`GlobalVariables`*[8] instance which will give you access to some application specific variables automatically:

`app.user`
: The representation of the current user or `null` if there is none. The value stored in this variable can be a *`UserInterface`*[9] object, any other object which implements a `__toString()` method or even a regular string.

`app.request`
: The *`Request`*[10] object that represents the current request (depending on your application, this can be a sub-request or a regular request, as explained later).

`app.session`
: The *`Session`*[11] object that represents the current user's session or `null` if there is none.

`app.environment`
: The name of the current environment (`dev`, `prod`, etc).

`app.debug`
: True if in debug mode. False otherwise.

Listing 7-30
```twig
<p>Username: {{ app.user.username }}</p>
{% if app.debug %}
    <p>Request method: {{ app.request.method }}</p>
    <p>Application Environment: {{ app.environment }}</p>
{% endif %}
```

8. http://api.symfony.com/3.0/Symfony/Bundle/FrameworkBundle/Templating/GlobalVariables.html
9. http://api.symfony.com/3.0/Symfony/Component/Security/Core/User/UserInterface.html
10. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Request.html
11. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Session/Session.html

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 Symfony 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:

Listing 7-31
```
return $this->render('article/index.html.twig');
```

is equivalent to:

Listing 7-32
```
use Symfony\Component\HttpFoundation\Response;

$engine = $this->container->get('templating');
$content = $engine->render('article/index.html.twig');

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

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

Listing 7-33
```
# app/config/config.yml
framework:
    # ...
    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 Symfony community prides itself on creating and maintaining high quality bundles (see *KnpBundles.com*[12]) 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 installed the imaginary open-source AcmeBlogBundle in your project. 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 7-34
```
public function indexAction()
{
    // some logic to retrieve the blogs
    $blogs = ...;

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

12. http://knpbundles.com

```
9       );
10  }
```

When the `AcmeBlogBundle:Blog:index.html.twig` is rendered, Symfony 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.

If you add a template in a new location, you *may* need to clear your cache (`php bin/console cache:clear`), even if you are in debug mode.

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, Symfony 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 Symfony 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*.

Overriding Core Templates

Since the Symfony 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 that were just covered:

- Create an `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, the blog functionality would have a template called `blog/layout.html.twig` that contains only blog section-specific elements;

Listing 7-35
```twig
{# app/Resources/views/blog/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 `blog/index.html.twig` and list the actual blog posts.

Listing 7-36
```twig
{# app/Resources/views/blog/index.html.twig #}
{% extends 'blog/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 (`blog/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*[13] (XSS) attack. Consider this classic example:

Listing 7-37
```twig
Hello {{ name }}
```

Imagine the user enters the following code for their name:

Listing 7-38
```twig
<script>alert('hello!')</script>
```

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

Listing 7-39
```twig
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 7-40

13. https://en.wikipedia.org/wiki/Cross-site_scripting

```
1 Hello &lt;script&gt;alert(&#39;hello!&#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:

Listing 7-41
```
1 {{ article.body|raw }}
```

You can also disable output escaping inside a `{% block %}` area or for an entire template. For more information, see *Output Escaping*[14] 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 7-42
```
1 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 7-43
```
1 var myMsg = 'Hello <?php echo $view->escape($name, 'js') ?>';
```

Debugging

When using PHP, you can use the dump() function from the VarDumper component if you need to quickly find the value of a variable passed. This is useful, for example, inside your controller:

Listing 7-44
```
1  // src/AppBundle/Controller/ArticleController.php
2  namespace AppBundle\Controller;
3
4  // ...
5
6  class ArticleController extends Controller
7  {
8      public function recentListAction()
9      {
10         $articles = ...;
11         dump($articles);
12
```

14. http://twig.sensiolabs.org/doc/api.html#escaper-extension

```
13          // ...
14      }
15  }
```

The output of the `dump()` function is then rendered in the web developer toolbar.

The same mechanism can be used in Twig templates thanks to `dump` function:

Listing 7-45
```
1  {# app/Resources/views/article/recent_list.html.twig #}
2  {{ dump(articles) }}
3
4  {% for article in articles %}
5      <a href="/article/{{ article.slug }}">
6          {{ article.title }}
7      </a>
8  {% 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.

Syntax Checking

You can check for syntax errors in Twig templates using the `lint:twig` console command:

Listing 7-46
```
1  # You can check by filename:
2  $ php bin/console lint:twig app/Resources/views/article/recent_list.html.twig
3
4  # or by directory:
5  $ php bin/console lint:twig app/Resources/views
```

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 formats. To render an article index page in XML, simply include the format in the template name:

- *XML template name*: `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 7-47
```
1  public function indexAction(Request $request)
2  {
3      $format = $request->getRequestFormat();
4
5      return $this->render('article/index.'.$format.'.twig');
6  }
```

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 7-48
```
<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 7-49
```
// creates a Response object whose content is the rendered template
$response = $this->render('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 Symfony, 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
Configuring Symfony (and Environments)

An application consists of a collection of bundles representing all 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 8-1
```yaml
# app/config/config.yml
imports:
    - { resource: parameters.yml }
    - { resource: security.yml }

framework:
    secret:          '%secret%'
    router:          { resource: '%kernel.root_dir%/config/routing.yml' }
    # ...

# 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 Symfony, 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 (learn more about YAML in "*The YAML Format*");
- *XML*: More powerful than YAML at times and supports IDE autocompletion;
- *PHP*: Very powerful but less readable than standard configuration formats.

Default Configuration Dump

You can dump the default configuration for a bundle in YAML to the console using the `config:dump-reference` command. Here is an example of dumping the default FrameworkBundle configuration:

Listing 8-2
```
$ php bin/console config:dump-reference FrameworkBundle
```

The extension alias (configuration key) can also be used:

Listing 8-3
```
$ php bin/console config:dump-reference framework
```

See the cookbook article: *How to Load Service Configuration inside a Bundle* for information on adding configuration for your own bundle.

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 Symfony 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 8-4
```
http://localhost/app_dev.php/random/10
```

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

Listing 8-5
```
http://localhost/app.php/random/10
```

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 8-6
```
$ php bin/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 8-7
```
$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.

When using the `server:run` command to start a server, `http://localhost:8000/` will use the dev front controller of your application.

Environment Configuration

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

Listing 8-8
```
// 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 8-9
```
# 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.

Chapter 9
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 Symfony, including both the core framework functionality and the code written for your application. Bundles are first-class citizens in Symfony. This gives you the flexibility to use pre-built features packaged in third-party bundles 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, JavaScript files, tests and anything else. Every aspect of a feature exists in a bundle and every feature lives in a bundle.

Bundles used in your applications must be enabled by registering them in the `registerBundles()` method of the `AppKernel` class:

Listing 9-1
```php
// app/AppKernel.php
public function registerBundles()
{
    $bundles = array(
        new Symfony\Bundle\FrameworkBundle\FrameworkBundle(),
        new Symfony\Bundle\SecurityBundle\SecurityBundle(),
        new Symfony\Bundle\TwigBundle\TwigBundle(),
        new Symfony\Bundle\MonologBundle\MonologBundle(),
        new Symfony\Bundle\SwiftmailerBundle\SwiftmailerBundle(),
        new Symfony\Bundle\DoctrineBundle\DoctrineBundle(),
        new Sensio\Bundle\FrameworkExtraBundle\SensioFrameworkExtraBundle(),
        new AppBundle\AppBundle(),
    );

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

```
20
21      return $bundles;
22  }
```

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.

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 9-2
```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 9-3
```php
// app/AppKernel.php
public function registerBundles()
{
    $bundles = array(
        // ...

        // register your bundle
        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 9-4
```
$ php bin/console generate:bundle --namespace=Acme/TestBundle
```

The bundle skeleton generates a basic controller, template and routing resource that can be customized. You'll learn more about Symfony'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 Symfony bundles. Take a look at AcmeDemoBundle, as it contains some of the most common elements of a bundle:

`Controller/`
: Contains the controllers of the bundle (e.g. `RandomController.php`).

`DependencyInjection/`
: Holds certain Dependency Injection Extension classes, which may import service configuration, register compiler passes or more (this directory is not necessary).

`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.

third-party bundles: *http://knpbundles.com*

Chapter 10
Databases and Doctrine

One of the most common and challenging tasks for any application involves persisting and reading information to and from a database. Although the Symfony full-stack Framework doesn't integrate any ORM by default, the Symfony Standard Edition, which is the most widely used distribution, 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 DBAL*" cookbook entry.

You can also persist data to *MongoDB*[2] using Doctrine ODM library. For more information, read the "*DoctrineMongoDBBundle*[3]" 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.

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.yml` file:

Listing 10-1
```
1  # app/config/parameters.yml
2  parameters:
3      database_host:      localhost
4      database_name:      test_project
```

1. http://www.doctrine-project.org/
2. https://www.mongodb.org/
3. https://symfony.com/doc/current/bundles/DoctrineMongoDBBundle/index.html

```
    database_user:      root
    database_password:  password

# ...
```

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

Listing 10-2
```
# app/config/config.yml
doctrine:
    dbal:
        driver:   pdo_mysql
        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 can connect to your database, the following command can automatically generate an empty `test_project` database for you:

Listing 10-3
```
$ php bin/console doctrine:database:create
```

Setting up the Database to be UTF8

One mistake even seasoned developers make when starting a Symfony project is forgetting to set up default charset and collation on their database, ending up with latin type collations, which are default for most databases. They might even remember to do it the very first time, but forget that it's all gone after running a relatively common command during development:

Listing 10-4
```
$ php bin/console doctrine:database:drop --force
$ php bin/console doctrine:database:create
```

Setting UTF8 defaults for MySQL is as simple as adding a few lines to your configuration file (typically `my.cnf`):

Listing 10-5
```
[mysqld]
# Version 5.5.3 introduced "utf8mb4", which is recommended
collation-server     = utf8mb4_general_ci # Replaces utf8_general_ci
character-set-server = utf8mb4            # Replaces utf8
```

You can also change the defaults for Doctrine so that the generated SQL uses the correct character set.

Listing 10-6
```
# app/config/config.yml
doctrine:
    dbal:
        charset: utf8mb4
        default_table_options:
            charset: utf8mb4
            collate: utf8mb4_unicode_ci
```

We recommend against MySQL's `utf8` character set, since it does not support 4-byte unicode characters, and strings containing them will be truncated. This is fixed by the *newer utf8mb4 character set*[4].

If you want to use SQLite as your database, you need to set the path where your database file should be stored:

Listing 10-7
```yaml
# app/config/config.yml
doctrine:
    dbal:
        driver: pdo_sqlite
        path: '%kernel.root_dir%/sqlite.db'
        charset: UTF8
```

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 AppBundle:

Listing 10-8
```php
// src/AppBundle/Entity/Product.php
namespace AppBundle\Entity;

class Product
{
    private $name;
    private $price;
    private $description;
}
```

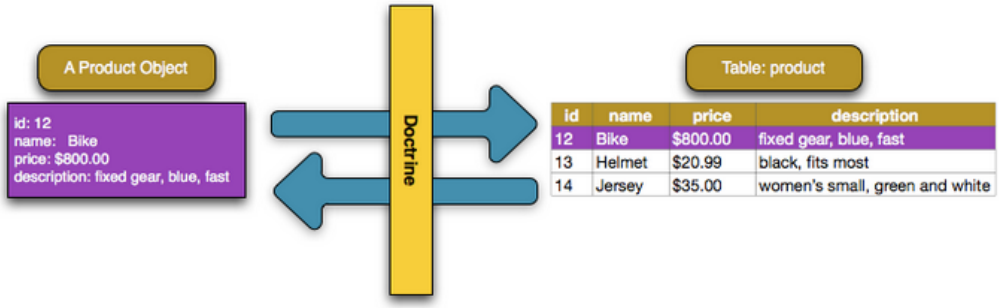
The class - often called an "entity", meaning *a basic class that holds data* - is simple and helps fulfill the business requirement of needing products in your application. This class can't be persisted to a database yet - it's just a simple PHP class.

Once you learn the concepts behind Doctrine, you can have Doctrine create simple entity classes for you. This will ask you interactive questions to help you build any entity:

Listing 10-9
```
$ php bin/console doctrine:generate:entity
```

Add Mapping Information

Doctrine allows you to work with databases in a much more interesting way than just fetching rows of scalar data into an array. Instead, Doctrine allows you to fetch entire *objects* out of the database, and to persist entire objects to the database. For Doctrine to be able to do this, you must *map* your database tables to specific PHP classes, and the columns on those tables must be mapped to specific properties on their corresponding PHP classes.



4. https://dev.mysql.com/doc/refman/5.5/en/charset-unicode-utf8mb4.html

You'll provide this mapping information in the form of "metadata", a collection of rules that tells Doctrine exactly how the `Product` class and its properties should be *mapped* to a specific database table. This metadata can be specified in a number of different formats, including YAML, XML or directly inside the `Product` class via DocBlock annotations:

Listing 10-10

```php
// src/AppBundle/Entity/Product.php
namespace AppBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity
 * @ORM\Table(name="product")
 */
class Product
{
    /**
     * @ORM\Column(type="integer")
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="AUTO")
     */
    private $id;

    /**
     * @ORM\Column(type="string", length=100)
     */
    private $name;

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

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

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.

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*[5] *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.*

5. *http://docs.doctrine-project.org/projects/doctrine-orm/en/latest/reference/basic-mapping.html*

Be careful if the names of your entity classes (or their properties) are also reserved SQL keywords like `GROUP` or `USER`. For example, if your entity's class name is `Group`, then, by default, the corresponding table name would be `group`. This will cause an SQL error in some database engines. See Doctrine's *Reserved SQL keywords documentation*[6] for details on how to properly escape these names. Alternatively, if you're free to choose your database schema, simply map to a different table name or column name. See Doctrine's *Creating Classes for the Database*[7] and *Property Mapping*[8] documentation.

When using another library or program (e.g. 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 10-11
```
/**
 * @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 with `private` properties, you need to create `public` getter and setter methods (e.g. `getName()`, `setName($name)`) in order to access its properties in the rest of your application's code. Fortunately, the following command can generate these boilerplate methods automatically:

Listing 10-12
```
$ php bin/console doctrine:generate:entities AppBundle/Entity/Product
```

This command makes sure that all 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).

Keep in mind that Doctrine's entity generator produces simple getters/setters. You should review the generated methods and add any logic, if necessary, to suit the needs of your application.

6. http://docs.doctrine-project.org/projects/doctrine-orm/en/latest/reference/basic-mapping.html#quoting-reserved-words
7. http://docs.doctrine-project.org/projects/doctrine-orm/en/latest/reference/basic-mapping.html#creating-classes-for-the-database
8. http://docs.doctrine-project.org/projects/doctrine-orm/en/latest/reference/basic-mapping.html#property-mapping

More about `doctrine:generate:entities`

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

- generate getter and setter methods in entity classes;
- generate repository classes on behalf of entities 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. You can also use the `--no-backup` option to prevent generating these backup files.

Note that you don't *need* to use this command. You could also write the necessary getters and setters by hand. This option simply exists to save you time, since creating these methods is often a common task during development.

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

Listing 10-13
```
# generates all entities in the AppBundle
$ php bin/console doctrine:generate:entities AppBundle

# generates all entities of bundles in the Acme namespace
$ php bin/console doctrine:generate:entities Acme
```

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:

Listing 10-14
```
$ php bin/console doctrine:schema:update --force
```

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 executes the SQL statements needed to *update* the database schema to where it should be. In other words, if you add a new property with mapping metadata to `Product` and run this task, it will execute 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*[9], 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 update and track changes to your database schema safely and reliably.

Whether or not you take advantage of migrations, the `doctrine:schema:update` command should only be used during development. It should not be used in a production environment.

Your database now has a fully-functional `product` table with columns that match the metadata you've specified.

9. `https://symfony.com/doc/current/bundles/DoctrineMigrationsBundle/index.html`

Persisting Objects to the Database

Now that you have mapped the `Product` entity to its corresponding `product` table, you're ready to persist `Product` objects to the database. From inside a controller, this is pretty easy. Add the following method to the `DefaultController` of the bundle:

Listing 10-15

```
1  // src/AppBundle/Controller/DefaultController.php
2
3  // ...
4  use AppBundle\Entity\Product;
5  use Symfony\Component\HttpFoundation\Response;
6
7  // ...
8  public function createAction()
9  {
10     $product = new Product();
11     $product->setName('Keyboard');
12     $product->setPrice(19.99);
13     $product->setDescription('Ergonomic and stylish!');
14
15     $em = $this->getDoctrine()->getManager();
16
17     // tells Doctrine you want to (eventually) save the Product (no queries yet)
18     $em->persist($product);
19
20     // actually executes the queries (i.e. the INSERT query)
21     $em->flush();
22
23     return new Response('Saved new product with id '.$product->getId());
24 }
```

If you're following along with this example, you'll need to create a route that points to this action to see it work.

This article shows working with Doctrine from within a controller by using the *`getDoctrine()`*[10] method of the controller. This method is a shortcut to get the `doctrine` service. You can work with Doctrine anywhere else by injecting that service in the service. See *Service Container* for more on creating your own services.

Take a look at the previous example in more detail:

- **lines 10-13** In this section, you instantiate and work with the `$product` object like any other normal PHP object.
- **line 15** This line fetches Doctrine's *entity manager* object, which is responsible for the process of persisting objects to, and fetching objects from, the database.
- **line 17** The `persist($product)` call tells Doctrine to "manage" the `$product` object. This does **not** cause a query to be made to the database.
- **line 18** When the `flush()` method is called, Doctrine looks through all of the objects that it's managing to see if they need to be persisted to the database. In this example, the `$product` object's data doesn't exist in the database, so the entity manager executes an `INSERT` query, creating a new row in the `product` table.

10. http://api.symfony.com/3.0/Symfony/Bundle/FrameworkBundle/Controller/Controller.html#method_getDoctrine

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 queries in the correct order. It utilizes cached prepared statement to slightly improve the performance. For example, if you persist a total of 100 `Product` objects and then subsequently call `flush()`, Doctrine will execute 100 `INSERT` queries using a single prepared statement object.

Whether 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 entity 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 the "*DoctrineFixturesBundle*[11]" documentation.

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 10-16
```php
public function showAction($productId)
{
    $product = $this->getDoctrine()
        ->getRepository('AppBundle:Product')
        ->find($productId);

    if (!$product) {
        throw $this->createNotFoundException(
            'No product found for id '.$productId
        );
    }

    // ... do something, like pass the $product object into a template
}
```

You can achieve the equivalent of this without writing any code by using the `@ParamConverter` shortcut. See the *FrameworkExtraBundle documentation*[12] for more details.

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 10-17
```php
$repository = $this->getDoctrine()
    ->getRepository('AppBundle:Product');
```

The `AppBundle:Product` string is a shortcut you can use anywhere in Doctrine instead of the full class name of the entity (i.e. `AppBundle\Entity\Product`). As long as your entity lives under the `Entity` namespace of your bundle, this will work.

Once you have a repository object, you can access all sorts of helpful methods:

Listing 10-18
```php
// query for a single product by its primary key (usually "id")
$product = $repository->find($productId);
```

11. https://symfony.com/doc/current/bundles/DoctrineFixturesBundle/index.html
12. https://symfony.com/doc/current/bundles/SensioFrameworkExtraBundle/annotations/converters.html

```
3
4  // dynamic method names to find a single product based on a column value
5  $product = $repository->findOneById($productId);
6  $product = $repository->findOneByName('Keyboard');
7
8  // dynamic method names to find a group of products based on a column value
9  $products = $repository->findByPrice(19.99);
10
11 // find *all* products
12 $products = $repository->findAll();
```

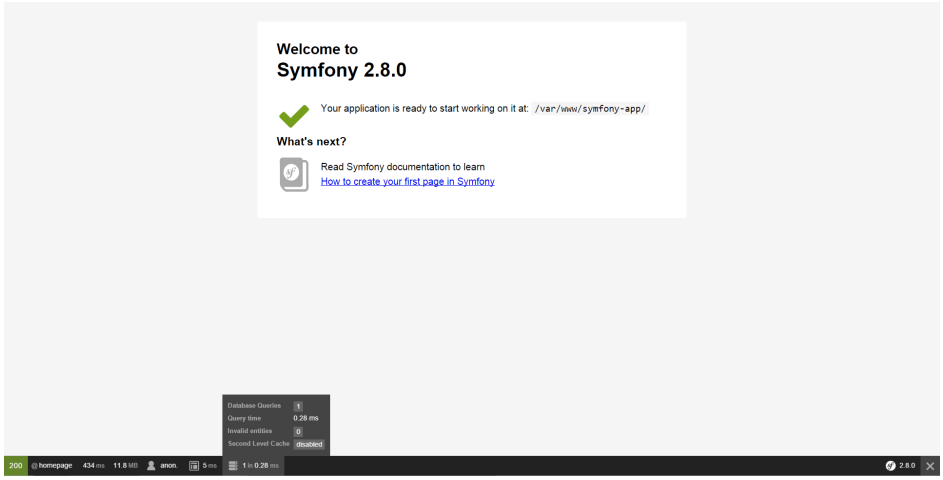
Of course, you can also issue complex queries, which you'll learn more about in the Querying for Objects section.

You can also take advantage of the useful `findBy` and `findOneBy` methods to easily fetch objects based on multiple conditions:

Listing 10-19

```
1  // query for a single product matching the given name and price
2  $product = $repository->findOneBy(
3      array('name' => 'Keyboard', 'price' => 19.99)
4  );
5
6  // query for multiple products matching the given name, ordered by price
7  $products = $repository->findBy(
8      array('name' => 'Keyboard'),
9      array('price' => 'ASC')
10 );
```

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.

The icon will turn yellow if there were more than 50 queries on the page. This could indicate that something is not correct.

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 10-20
```php
public function updateAction($productId)
{
    $em = $this->getDoctrine()->getManager();
    $product = $em->getRepository('AppBundle:Product')->find($productId);

    if (!$product) {
        throw $this->createNotFoundException(
            'No product found for id '.$productId
        );
    }

    $product->setName('New product name!');
    $em->flush();

    return $this->redirectToRoute('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:

Listing 10-21
```php
$em->remove($product);
$em->flush();
```

As you might expect, the `remove()` method notifies Doctrine that you'd like to remove the given object 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:

Listing 10-22
```php
$product = $repository->find($productId);
$product = $repository->findOneByName('Keyboard');
```

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 main options: writing pure DQL queries or using Doctrine's Query Builder.

Querying for Objects with DQL

Imagine that you want to query for products that cost more than `19.99`, ordered from least to most expensive. You can use DQL, Doctrine's native SQL-like language, to construct a query for this scenario:

Listing 10-23
```php
$em = $this->getDoctrine()->getManager();
$query = $em->createQuery(
    'SELECT p
```

```
    FROM AppBundle:Product p
    WHERE p.price > :price
    ORDER BY p.price ASC'
)->setParameter('price', 19.99);

$products = $query->getResult();
```

If you're comfortable with SQL, then DQL should feel very natural. The biggest difference is that you need to think in terms of selecting PHP objects, instead of rows in a database. For this reason, you select *from* the `AppBundle:Product` *entity* (an optional shortcut for the `AppBundle\Entity\Product` class) and then alias it as `p`.

Take note of the `setParameter()` method. When working with Doctrine, it's always a good idea to set any external values as "placeholders" (`:price` in the example above) as it prevents SQL injection attacks.

The `getResult()` method returns an array of results. To get only one result, you can use `getOneOrNullResult()`:

Listing 10-24
```
$product = $query->setMaxResults(1)->getOneOrNullResult();
```

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 Query Language*[13] documentation.

Querying for Objects Using Doctrine's Query Builder

Instead of writing a DQL string, you can use a helpful object called the `QueryBuilder` to build that string for you. This is useful when the actual query depends on dynamic conditions, as your code soon becomes hard to read with DQL as you start to concatenate strings:

Listing 10-25
```
$repository = $this->getDoctrine()
    ->getRepository('AppBundle:Product');

// createQueryBuilder automatically selects FROM AppBundle:Product
// and aliases it to "p"
$query = $repository->createQueryBuilder('p')
    ->where('p.price > :price')
    ->setParameter('price', '19.99')
    ->orderBy('p.price', 'ASC')
    ->getQuery();

$products = $query->getResult();
// to get just one result:
// $product = $query->setMaxResults(1)->getOneOrNullResult();
```

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 can be used to get the result of the query.

For more information on Doctrine's Query Builder, consult Doctrine's *Query Builder*[14] documentation.

13. http://docs.doctrine-project.org/projects/doctrine-orm/en/latest/reference/dql-doctrine-query-language.html
14. http://docs.doctrine-project.org/projects/doctrine-orm/en/latest/reference/query-builder.html

Custom Repository Classes

In the previous sections, you began constructing and using more complex queries from inside a controller. In order to isolate, reuse and test these queries, it's a good practice to create a custom repository class for your entity. Methods containing your query logic can then be stored in this class.

To do this, add the repository class name to your entity's mapping definition:

Listing 10-26
```php
// src/AppBundle/Entity/Product.php
namespace AppBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity(repositoryClass="AppBundle\Entity\ProductRepository")
 */
class Product
{
    //...
}
```

Doctrine can generate empty repository classes for all the entities in your application via the same command used earlier to generate the missing getter and setter methods:

Listing 10-27
```
$ php bin/console doctrine:generate:entities AppBundle
```

If you opt to create the repository classes yourself, they must extend `Doctrine\ORM\EntityRepository`.

Next, add a new method - `findAllOrderedByName()` - to the newly-generated `ProductRepository` class. This method will query for all the `Product` entities, ordered alphabetically by name.

Listing 10-28
```php
// src/AppBundle/Entity/ProductRepository.php
namespace AppBundle\Entity;

use Doctrine\ORM\EntityRepository;

class ProductRepository extends EntityRepository
{
    public function findAllOrderedByName()
    {
        return $this->getEntityManager()
            ->createQuery(
                'SELECT p FROM AppBundle: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:

Listing 10-29
```php
$em = $this->getDoctrine()->getManager();
$products = $em->getRepository('AppBundle: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 each product in your application belongs to exactly one category. In this case, you'll need a `Category` class, 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 category objects through Doctrine, you can let Doctrine create the class for you.

Listing 10-30
```
$ php bin/console doctrine:generate:entity --no-interaction \
    --entity="AppBundle: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

In this example, each category can be associated with *many* products, while each product can be associated with only *one* category. This relationship can be summarized as: *many* products to *one* category (or equivalently, *one* category to *many* products).

From the perspective of the `Product` entity, this is a many-to-one relationship. From the perspective of the `Category` entity, this is a one-to-many relationship. This is important, because the relative nature of the relationship determines which mapping metadata to use. It also determines which class *must* hold a reference to the other class.

To relate the `Product` and `Category` entities, simply create a `category` property on the `Product` class, annotated as follows:

Listing 10-31
```
// src/AppBundle/Entity/Product.php

// ...
class Product
{
    // ...

    /**
     * @ORM\ManyToOne(targetEntity="Category", inversedBy="products")
     * @ORM\JoinColumn(name="category_id", referencedColumnName="id")
     */
    private $category;
}
```

This many-to-one mapping is critical. It tells Doctrine to use the `category_id` column on the `product` table to relate each record in that table with a record in the `category` table.

Next, since a single `Category` object will relate to many `Product` objects, a `products` property can be added to the `Category` class to hold those associated objects.

Listing 10-32
```
// src/AppBundle/Entity/Category.php

// ...
use Doctrine\Common\Collections\ArrayCollection;

```

```php
class Category
{
    // ...

    /**
     * @ORM\OneToMany(targetEntity="Product", mappedBy="category")
     */
    private $products;

    public function __construct()
    {
        $this->products = new ArrayCollection();
    }
}
```

While the many-to-one mapping shown earlier was mandatory, this one-to-many mapping is optional. It is included here to help demonstrate Doctrine's range of relationship management capabailties. Plus, in the context of this application, it will likely be convenient for each `Category` object to automatically own a collection of its related `Product` objects.

The code in the constructor is important. Rather than being instantiated as a traditional `array`, the `$products` property must be of a type that implements Doctrine's `Collection` interface. In this case, an `ArrayCollection` object is used. 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 metadata used above can reference any entity with a valid namespace, not just entities defined in the same namespace. To relate to an entity defined in a different class or bundle, enter a full namespace as the targetEntity.

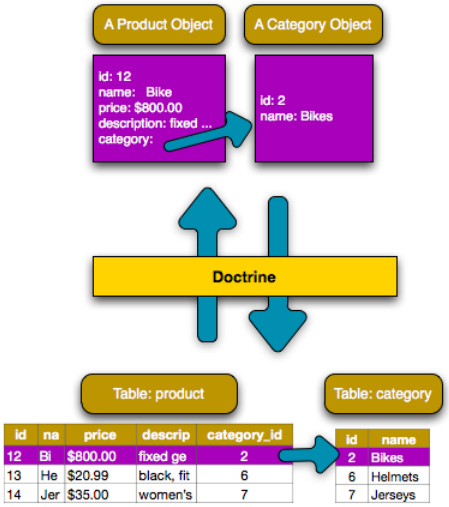
Now that you've added new properties to both the `Product` and `Category` classes, tell Doctrine to generate the missing getter and setter methods for you:

Listing 10-33
```
$ php bin/console doctrine:generate:entities AppBundle
```

Ignore the Doctrine metadata for a moment. You now have two classes - `Product` and `Category`, with a natural many-to-one relationship. The `Product` class holds a *single* `Category` object, and the `Category` class holds a *collection* of `Product` objects. In other words, you've built your classes in a way that makes sense for your application. The fact that the data needs to be persisted to a database is always secondary.

Now, review the metadata above the `Product` entity's `$category` property. It tells Doctrine that the related class is `Category`, and that the `id` of the related category record should be stored in a `category_id` field on the `product` table.

In other words, the related `Category` object will be stored in the `$category` property, but behind the scenes, Doctrine will persist this relationship by storing the category's id in the `category_id` column of the `product` table.



The metadata above the `Category` entity's `$products` property is less complicated. It 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, the new `product.category_id` column, and the new foreign key:

Listing 10-34
```
$ php bin/console doctrine:schema:update --force
```

Saving Related Entities

Now you can see this new code in action! Imagine you're inside a controller:

Listing 10-35
```
// ...

use AppBundle\Entity\Category;
use AppBundle\Entity\Product;
use Symfony\Component\HttpFoundation\Response;

class DefaultController extends Controller
{
    public function createProductAction()
    {
        $category = new Category();
        $category->setName('Computer Peripherals');

        $product = new Product();
        $product->setName('Keyboard');
        $product->setPrice(19.99);
        $product->setDescription('Ergonomic and stylish!');

        // relate this product to the category
        $product->setCategory($category);

        $em = $this->getDoctrine()->getManager();
        $em->persist($category);
        $em->persist($product);
        $em->flush();

        return new Response(
            'Saved new product with id: '.$product->getId()
            .' and new category with id: '.$category->getId()
        );
    }
}
```

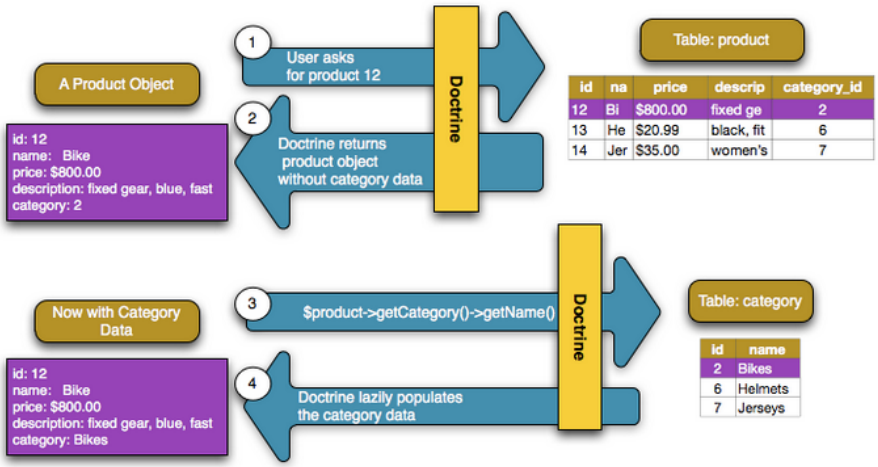
Now, a single row is added to both the `category` and `product` tables. The `product.category_id` column for the new product is set to whatever the `id` is of the new category. Doctrine manages the persistence of this relationship for you.

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` object:

Listing 10-36
```php
public function showAction($productId)
{
    $product = $this->getDoctrine()
        ->getRepository('AppBundle:Product')
        ->find($productId);

    $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 10-37
```php
public function showProductsAction($categoryId)
{
    $category = $this->getDoctrine()
        ->getRepository('AppBundle:Category')
        ->find($categoryId);

    $products = $category->getProducts();

    // ...
}
```

In this case, the same things occur: 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 10-38
```php
$product = $this->getDoctrine()
    ->getRepository('AppBundle:Product')
    ->find($productId);

$category = $product->getCategory();

// prints "Proxies\AppBundleEntityCategoryProxy"
dump(get_class($category));
die();
```

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 it 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 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 10-39
```php
// src/AppBundle/Entity/ProductRepository.php
public function findOneByIdJoinedToCategory($productId)
{
    $query = $this->getEntityManager()
        ->createQuery(
            'SELECT p, c FROM AppBundle:Product p
            JOIN p.category c
            WHERE p.id = :id'
        )->setParameter('id', $productId);

    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 10-40
```php
public function showAction($productId)
{
    $product = $this->getDoctrine()
        ->getRepository('AppBundle:Product')
        ->findOneByIdJoinedToCategory($productId);

    $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*[15].

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 *config reference*.

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 10-41
```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 `createdAt` date column to the current date, only when the entity is first persisted (i.e. inserted):

Listing 10-42

15. http://docs.doctrine-project.org/projects/doctrine-orm/en/latest/reference/association-mapping.html

```php
// src/AppBundle/Entity/Product.php

/**
 * @ORM\PrePersist
 */
public function setCreatedAtValue()
{
    $this->createdAt = new \DateTime();
}
```

The above example assumes that you've created and mapped a `createdAt` property (not shown here).

Now, right before the entity is first persisted, Doctrine will automatically call this method and the `createdAt` field will be set to the current date.

There are several other lifecycle events that you can hook into. For more information on other lifecycle events and lifecycle callbacks in general, see Doctrine's *Lifecycle Events documentation*[16].

Lifecycle Callbacks and Event Listeners

Notice that the `setCreatedAtValue()` 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 performing logging or sending 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 *How to Register Event Listeners and Subscribers*.

Doctrine Field Types Reference

Doctrine comes with numerous field types available. Each of these maps a PHP data type to a specific column type in whatever database you're using. For each field type, the `Column` can be configured further, setting the `length`, `nullable` behavior, `name` and other options. To see a list of all available types and more information, see Doctrine's *Mapping Types documentation*[17].

Summary

With Doctrine, you can focus on your objects and how they're used 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.

16. http://docs.doctrine-project.org/projects/doctrine-orm/en/latest/reference/events.html#lifecycle-events
17. http://docs.doctrine-project.org/projects/doctrine-orm/en/latest/reference/basic-mapping.html#property-mapping

Learn more

For more information about Doctrine, see the *Doctrine* section of the *cookbook*. Some useful articles might be:

- *How to use Doctrine Extensions: Timestampable, Sluggable, Translatable, etc.*
- *Console Commands*
- *DoctrineFixturesBundle*[18]
- *DoctrineMongoDBBundle*[19]

18. https://symfony.com/doc/current/bundles/DoctrineFixturesBundle/index.html
19. https://symfony.com/doc/current/bundles/DoctrineMongoDBBundle/index.html

Chapter 11
Databases and Propel

Propel is an open-source Object-Relational Mapping (ORM) for PHP which implements the *ActiveRecord pattern*[1]. It allows you to access your database using a set of objects, providing a simple API for storing and retrieving data. Propel uses PDO as an abstraction layer and code generation to remove the burden of runtime introspection.

A few years ago, Propel was a very popular alternative to Doctrine. However, its popularity has rapidly declined and that's why the Symfony book no longer includes the Propel documentation. Read the *official PropelBundle documentation*[2] to learn how to integrate Propel into your Symfony projects.

1. https://en.wikipedia.org/wiki/Active_record_pattern
2. https://github.com/propelorm/PropelBundle/blob/1.4/Resources/doc/index.markdown

Chapter 12
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

Symfony 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].

It's recommended to use the latest stable PHPUnit version, installed as PHAR.

Each test - whether it's a unit test or a functional test - is a PHP class that should live in the `tests/` directory of your application. If you follow this rule, then you can run all of your application's tests with the following command:

Listing 12-1
```
$ phpunit
```

PHPunit is configured by the `phpunit.xml.dist` file in the root of your Symfony application.

Code coverage can be generated with the `--coverage-*` options, see the help information that is shown when using `--help` for more information.

Unit Tests

A unit test is a test against a single PHP class, also called a *unit*. If you want to test the overall behavior of your application, see the section about Functional Tests.

1. https://phpunit.de/manual/current/en/

Writing Symfony unit tests is no different from writing standard PHPUnit unit tests. Suppose, for example, that you have an *incredibly* simple class called `Calculator` in the `Util/` directory of the app bundle:

Listing 12-2
```php
// src/AppBundle/Util/Calculator.php
namespace AppBundle\Util;

class Calculator
{
    public function add($a, $b)
    {
        return $a + $b;
    }
}
```

To test this, create a `CalculatorTest` file in the `tests/AppBundle/Util` directory of your application:

Listing 12-3
```php
// tests/AppBundle/Util/CalculatorTest.php
namespace Tests\AppBundle\Util;

use AppBundle\Util\Calculator;

class CalculatorTest extends \PHPUnit_Framework_TestCase
{
    public function testAdd()
    {
        $calc = new Calculator();
        $result = $calc->add(30, 12);

        // assert that your calculator added the numbers correctly!
        $this->assertEquals(42, $result);
    }
}
```

By convention, the `tests/AppBundle` directory should replicate the directory of your bundle for unit tests. So, if you're testing a class in the `src/AppBundle/Util/` directory, put the test in the `tests/AppBundle/Util/` directory.

Just like in your real application - autoloading is automatically enabled via the `app/autoload.php` file (as configured by default in the `phpunit.xml.dist` file).

Running tests for a given file or directory is also very easy:

Listing 12-4
```
# run all tests of the application
$ phpunit

# run all tests in the Util directory
$ phpunit tests/AppBundle/Util

# run tests for the Calculator class
$ phpunit tests/AppBundle/Util/CalculatorTest.php

# run all tests for the entire Bundle
$ phpunit tests/AppBundle/
```

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/AppBundle/Controller` directory for your bundle. If you want to test the pages handled by your `PostController` class, start by creating a new `PostControllerTest.php` file that extends a special `WebTestCase` class.

As an example, a test could look like this:

Listing 12-5
```php
// tests/AppBundle/Controller/PostControllerTest.php
namespace Tests\AppBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Test\WebTestCase;

class PostControllerTest extends WebTestCase
{
    public function testShowPost()
    {
        $client = static::createClient();

        $crawler = $client->request('GET', '/post/hello-world');

        $this->assertGreaterThan(
            0,
            $crawler->filter('html:contains("Hello World")')->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 12-6
```xml
<?xml version="1.0" charset="utf-8" ?>
<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:

Listing 12-7
```php
$crawler = $client->request('GET', '/post/hello-world');
```

The `request()` method (read more about the request method) returns a *`Crawler`*[2] 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()`.

2. http://api.symfony.com/3.0/Symfony/Component/DomCrawler/Crawler.html

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:

Listing 12-8
```
$link = $crawler
    ->filter('a:contains("Greet")') // find all links with the text "Greet"
    ->eq(1) // select the second link in the list
    ->link()
;

// and click it
$crawler = $client->click($link);
```

Submitting a form is very similar: select a form button, optionally override some form values and submit the corresponding form:

Listing 12-9
```
$form = $crawler->selectButton('submit')->form();

// set some values
$form['name'] = 'Lucas';
$form['form_name[subject]'] = 'Hey there!';

// submit the form
$crawler = $client->submit($form);
```

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:

Listing 12-10
```
// Assert that the response matches a given CSS selector.
$this->assertGreaterThan(0, $crawler->filter('h1')->count());
```

Or test against the response content directly if you just want to assert that the content contains some text or in case that the response is not an XML/HTML document:

Listing 12-11
```
$this->assertContains(
    'Hello World',
    $client->getResponse()->getContent()
);
```

Useful Assertions

To get you started faster, here is a list of the most common and useful test assertions:

Listing 12-12

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

// ...

// Assert that there is at least 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'));

// Assert that the "Content-Type" header is "application/json"
$this->assertTrue(
    $client->getResponse()->headers->contains(
        'Content-Type',
        'application/json'
    ),
    'the "Content-Type" header is "application/json"' // optional message shown on failure
);

// Assert that the response content contains a string
$this->assertContains('foo', $client->getResponse()->getContent());
// ...or matches a regex
$this->assertRegExp('/foo(bar)?/', $client->getResponse()->getContent());

// Assert that the response status code is 2xx
$this->assertTrue($client->getResponse()->isSuccessful(), 'response status is 2xx');
// Assert that the response status code is 404
$this->assertTrue($client->getResponse()->isNotFound());
// Assert a specific 200 status code
$this->assertEquals(
    200, // or Symfony\Component\HttpFoundation\Response::HTTP_OK
    $client->getResponse()->getStatusCode()
);

// Assert that the response is a redirect to /demo/contact
$this->assertTrue(
    $client->getResponse()->isRedirect('/demo/contact'),
    'response is a redirect to /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 Symfony application:

Listing 12-13

```
$crawler = $client->request('GET', '/post/hello-world');
```

The `request()` method takes the HTTP method and a URL as arguments and returns a `Crawler` instance.

Hardcoding the request URLs is a best practice for functional tests. If the test generates URLs using the Symfony router, it won't detect any change made to the application URLs which may impact the end users.

More about the `request()` Method:

The full signature of the `request()` method is:

Listing 12-14
```
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*[3] superglobal. For example, to set the `Content-Type`, `Referer` and `X-Requested-With` HTTP headers, you'd pass the following (mind the `HTTP_` prefix for non standard headers):

Listing 12-15
```
$client->request(
    'GET',
    '/post/hello-world',
    array(),
    array(),
    array(
        'CONTENT_TYPE'          => 'application/json',
        'HTTP_REFERER'          => '/foo/bar',
        'HTTP_X-Requested-With' => 'XMLHttpRequest',
    )
);
```

Use the crawler to find DOM elements in the response. These elements can then be used to click on links and submit forms:

Listing 12-16
```
$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. Some useful examples:

Listing 12-17
```
// Directly submit a form (but using the Crawler is easier!)
$client->request('POST', '/submit', array('name' => 'Fabien'));

// Submit a raw JSON string in the request body
$client->request(
    'POST',
    '/submit',
    array(),
    array(),
    array('CONTENT_TYPE' => 'application/json'),
```

3. http://php.net/manual/en/reserved.variables.server.php

```
    '{"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
);
$client->request(
    'POST',
    '/submit',
    array('name' => 'Fabien'),
    array('photo' => $photo)
);

// Perform a DELETE request 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 12-18
```
$client->insulate();
```

Browsing

The Client supports many operations that can be done in a real browser:

Listing 12-19
```
$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:

Listing 12-20
```
$history = $client->getHistory();
$cookieJar = $client->getCookieJar();
```

You can also get the objects related to the latest request:

Listing 12-21
```
// the HttpKernel request instance
$request = $client->getRequest();

// the BrowserKit request instance
$request = $client->getInternalRequest();

// the HttpKernel response instance
$response = $client->getResponse();

// the BrowserKit response instance
$response = $client->getInternalResponse();
```

```
12
13  $crawler = $client->getCrawler();
```

If your requests are not insulated, you can also access the `Container` and the `Kernel`:

Listing 12-22
```
$container = $client->getContainer();
$kernel = $client->getKernel();
```

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:

Listing 12-23
```
$container = $client->getContainer();
```

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 `debug:container` console task.

If the information you need to check is available from the profiler, use it instead.

Accessing the Profiler Data

On each request, you can enable the Symfony profiler to collect 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:

Listing 12-24
```
1  // enable the profiler for the very next request
2  $client->enableProfiler();
3
4  $crawler = $client->request('GET', '/profiler');
5
6  // get the profile
7  $profile = $client->getProfile();
```

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:

Listing 12-25
```
$crawler = $client->followRedirect();
```

If you want the client to automatically follow all redirects, you can force him with the `followRedirects()` method:

Listing 12-26
```
$client->followRedirects();
```

If you pass `false` to the `followRedirects()` method, the redirects will no longer be followed:

Listing 12-27
```
$client->followRedirects(false);
```

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 12-28
```
$newCrawler = $crawler->filter('input[type=submit]')
    ->last()
    ->parents()
    ->first()
;
```

Many other methods are also available:

`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 12-29
```
$crawler
    ->filter('h1')
    ->reduce(function ($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 12-30

```
// 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 12-31

```
$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 12-32

```
$link = $crawler->selectLink('Click here')->link();

$client->click($link);
```

Forms

Forms can be selected using their buttons, which can be selected with the `selectButton()` method, just like links:

Listing 12-33

```
$buttonCrawlerNode = $crawler->selectButton('submit');
```

Notice that you 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 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 12-34
```
$form = $buttonCrawlerNode->form();
```

When calling the `form()` method, you can also pass an array of field values that overrides the default ones:

Listing 12-35
```
$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 12-36
```
$form = $buttonCrawlerNode->form(array(), 'DELETE');
```

The Client can submit `Form` instances:

Listing 12-37
```
$client->submit($form);
```

The field values can also be passed as a second argument of the `submit()` method:

Listing 12-38
```
$client->submit($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 12-39
```
// 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 12-40
```
1  // Select an option or a radio
2  $form['country']->select('France');
3
4  // Tick a checkbox
5  $form['like_symfony']->tick();
6
7  // Upload a file
8  $form['photo']->upload('/path/to/lucas.jpg');
```

If you purposefully want to select "invalid" select/radio values, see Selecting Invalid Choice Values.

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).

Adding and Removing Forms to a Collection

If you use a *Collection of Forms*, you can't add fields to an existing form with `$form['task[tags][0][name]'] = 'foo';`. This results in an error `Unreachable field "…"` because `$form` can only be used in order to set values of existing fields. In order to add new fields, you have to add the values to the raw data array:

Listing 12-41
```php
// Get the form.
$form = $crawler->filter('button')->form();

// Get the raw values.
$values = $form->getPhpValues();

// Add fields to the raw values.
$values['task']['tag'][0]['name'] = 'foo';
$values['task']['tag'][1]['name'] = 'bar';

// Submit the form with the existing and new values.
$crawler = $this->client->request($form->getMethod(), $form->getUri(), $values,
    $form->getPhpFiles());

// The 2 tags have been added to the collection.
$this->assertEquals(2, $crawler->filter('ul.tags > li')->count());
```

Where `task[tags][0][name]` is the name of a field created with JavaScript.

You can remove an existing field, e.g. a tag:

Listing 12-42
```php
// Get the values of the form.
$values = $form->getPhpValues();

// Remove the first tag.
unset($values['task']['tags'][0]);

// Submit the data.
$crawler = $client->request($form->getMethod(), $form->getUri(),
    $values, $form->getPhpFiles());

// The tag has been removed.
$this->assertEquals(0, $crawler->filter('ul.tags > li')->count());
```

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 Swift Mailer is configured to *not* actually deliver emails in the `test` environment. You can see this under the `swiftmailer` configuration option:

Listing 12-43
```yaml
# app/config/config_test.yml

# ...
```

```
4  swiftmailer:
5      disable_delivery: true
```

You can also use a different environment entirely, or override the default debug mode (`true`) by passing each as options to the `createClient()` method:

Listing 12-44
```
$client = static::createClient(array(
    'environment' => 'my_test_env',
    'debug'       => false,
));
```

If your application behaves according to some HTTP headers, pass them as the second argument of `createClient()`:

Listing 12-45
```
$client = static::createClient(array(), array(
    'HTTP_HOST'       => 'en.example.com',
    'HTTP_USER_AGENT' => 'MySuperBrowser/1.0',
));
```

You can also override HTTP headers on a per request basis:

Listing 12-46
```
$client->request('GET', '/', array(), array(), array(
    'HTTP_HOST'       => 'en.example.com',
    'HTTP_USER_AGENT' => 'MySuperBrowser/1.0',
));
```

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 set up a configuration for your local machine only.

Store the `phpunit.xml.dist` file in your code repository and ignore the `phpunit.xml` file.

By default, only the tests stored in `/tests` are run via the `phpunit` command, as configured in the `phpunit.xml.dist` file:

Listing 12-47
```
1  <!-- phpunit.xml.dist -->
2  <phpunit>
3      <!-- ... -->
4      <testsuites>
5          <testsuite name="Project Test Suite">
6              <directory>tests</directory>
7          </testsuite>
8      </testsuites>
9      <!-- ... -->
10 </phpunit>
```

But you can easily add more directories. For instance, the following configuration adds tests from a custom `lib/tests` directory:

Listing 12-48
```
1  <!-- phpunit.xml.dist -->
2  <phpunit>
```

```
    <!-- ... -->
    <testsuites>
        <testsuite name="Project Test Suite">
            <!-- ... --->
            <directory>lib/tests</directory>
        </testsuite>
    </testsuites>
    <!-- ... --->
</phpunit>
```

To include other directories in the code coverage, also edit the `<filter>` section:

Listing 12-49
```
<!-- phpunit.xml.dist -->
<phpunit>
    <!-- ... -->
    <filter>
        <whitelist>
            <!-- ... -->
            <directory>lib</directory>
            <exclude>
                <!-- ... -->
                <directory>lib/tests</directory>
            </exclude>
        </whitelist>
    </filter>
    <!-- ... --->
</phpunit>
```

Learn more

- *The chapter about tests in the Symfony Framework Best Practices*
- *The DomCrawler Component*
- *The CssSelector Component*
- *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*
- *How to Customize the Bootstrap Process before Running Tests*

Chapter 13
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.

Symfony ships with a *Validator*[1] component that makes this task easy and transparent. This component is based on the *JSR303 Bean Validation specification*[2].

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 13-1
```php
// src/AppBundle/Entity/Author.php
namespace AppBundle\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 if 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 13-2
```php
// src/AppBundle/Entity/Author.php

// ...
use Symfony\Component\Validator\Constraints as Assert;

class Author
{
    /**
```

1. https://github.com/symfony/validator
2. http://jcp.org/en/jsr/detail?id=303

```
     * @Assert\NotBlank()
     */
    public $name;
}
```

Protected and private properties can also be validated, as well as "getter" methods (see 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 if the data on the object satisfies those constraints. If validation fails, a non-empty list of errors (class *`ConstraintViolationList`*[4]) is returned. Take this simple example from inside a controller:

Listing 13-3

```
// ...
use Symfony\Component\HttpFoundation\Response;
use AppBundle\Entity\Author;

// ...
public function authorAction()
{
    $author = new Author();

    // ... do something to the $author object

    $validator = $this->get('validator');
    $errors = $validator->validate($author);

    if (count($errors) > 0) {
        /*
         * Uses a __toString method on the $errors variable which is a
         * ConstraintViolationList object. This gives us a nice string
         * for debugging.
         */
        $errorsString = (string) $errors;

        return new Response($errorsString);
    }

    return new Response('The author is valid! Yes!');
}
```

If the `$name` property is empty, you will see the following error message:

Listing 13-4

```
AppBundle\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:

3. http://api.symfony.com/3.0/Symfony/Component/Validator/Validator.html
4. http://api.symfony.com/3.0/Symfony/Component/Validator/ConstraintViolationList.html

Listing 13-5
```php
if (count($errors) > 0) {
    return $this->render('author/validation.html.twig', array(
        'errors' => $errors,
    ));
}
```

Inside the template, you can output the list of errors exactly as needed:

Listing 13-6
```twig
{# app/Resources/views/author/validation.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*[5] 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. The constraint violations on the object are converted into `FormError` objects that can easily be displayed with your form. The typical form submission workflow looks like the following from inside a controller:

Listing 13-7
```php
// ...
use AppBundle\Entity\Author;
use AppBundle\Form\AuthorType;
use Symfony\Component\HttpFoundation\Request;

// ...
public function updateAction(Request $request)
{
    $author = new Author();
    $form = $this->createForm(AuthorType::class, $author);

    $form->handleRequest($request);

    if ($form->isValid()) {
        // the validation passed, do something with the $author object

        return $this->redirectToRoute(...);
    }

    return $this->render('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.

5. http://api.symfony.com/3.0/Symfony/Component/Validator/ConstraintViolation.html

Configuration

The Symfony validator is enabled by default, but you must explicitly enable annotations if you're using the annotation method to specify your constraints:

Listing 13-8
```yaml
# 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 Symfony, constraints are similar: they are assertions that a condition is true. Given a value, a constraint will tell you if that value adheres to the rules of the constraint.

Supported Constraints

Symfony packages many 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*
- *IsNull*
- *IsTrue*
- *IsFalse*
- *Type*

String Constraints

- *Email*
- *Length*
- *Url*
- *Regex*
- *Ip*
- *Uuid*

Number Constraints

- *Range*

Comparison Constraints

- *EqualTo*
- *NotEqualTo*

- *IdenticalTo*
- *NotIdenticalTo*
- *LessThan*
- *LessThanOrEqual*
- *GreaterThan*
- *GreaterThanOrEqual*

Date Constraints

- *Date*
- *DateTime*
- *Time*

Collection Constraints

- *Choice*
- *Collection*
- *Count*
- *UniqueEntity*
- *Language*
- *Locale*
- *Country*

File Constraints

- *File*
- *Image*

Financial and other Number Constraints

- *Bic*
- *CardScheme*
- *Currency*
- *Luhn*
- *Iban*
- *Isbn*
- *Issn*

Other Constraints

- *Callback*
- *Expression*
- *All*
- *UserPassword*
- *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 called `gender` that can be set to either "male", "female" or "other":

Listing 13-9

```php
// src/AppBundle/Entity/Author.php

// ...
use Symfony\Component\Validator\Constraints as Assert;

class Author
{
    /**
     * @Assert\Choice(
     *     choices = { "male", "female", "other" },
     *     message = "Choose a valid gender."
     * )
     */
    public $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 13-10

```php
// src/AppBundle/Entity/Author.php

// ...
use Symfony\Component\Validator\Constraints as Assert;

class Author
{
    /**
     * @Assert\Choice({"male", "female", "other"})
     */
    protected $gender;

    // ...
}
```

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`), a public getter method (e.g. `getFullName`) or an entire class. Property constraints are the most common and easy to use. Getter constraints allow you to specify more complex validation rules. Finally, class constraints are intended for scenarios where you want to validate a class as a whole.

Properties

Validating class properties is the most basic validation technique. Symfony 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 13-11
```php
// src/AppBundle/Entity/Author.php

// ...
use Symfony\Component\Validator\Constraints as Assert;

class Author
{
    /**
     * @Assert\NotBlank()
     * @Assert\Length(min=3)
     */
    private $firstName;
}
```

Getters

Constraints can also be applied to the return value of a method. Symfony allows you to add a constraint to any public method whose name starts with "get", "is" or "has". In this guide, 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 13-12
```php
// src/AppBundle/Entity/Author.php

// ...
use Symfony\Component\Validator\Constraints as Assert;

class Author
{
    /**
     * @Assert\IsTrue(message = "The password cannot match your first name")
     */
    public function isPasswordLegal()
    {
        // ... return true or false
    }
}
```

Now, create the `isPasswordLegal()` method and include the logic you need:

Listing 13-13
```php
public function isPasswordLegal()
{
    return $this->firstName !== $this->password;
}
```

The keen-eyed among you will have noticed that the prefix of the getter ("get", "is" or "has") 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 the defined constraints. In some cases, however, you'll need to validate an object against only *some* 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 their contact information later:

Listing 13-14

```php
// src/AppBundle/Entity/User.php
namespace AppBundle\Entity;

use Symfony\Component\Security\Core\User\UserInterface;
use Symfony\Component\Validator\Constraints as Assert;

class User implements UserInterface
{
    /**
     * @Assert\Email(groups={"registration"})
     */
    private $email;

    /**
     * @Assert\NotBlank(groups={"registration"})
     * @Assert\Length(min=7, groups={"registration"})
     */
    private $password;

    /**
     * @Assert\Length(min=2)
     */
    private $city;
}
```

With this configuration, there are three validation groups:

`Default`
: Contains the constraints in the current class and all referenced classes that belong to no other group.

`User`
: Equivalent to all constraints of the `User` object in the `Default` group. This is always the name of the class. The difference between this and `Default` is explained below.

`registration`
: Contains the constraints on the `email` and `password` fields only.

Constraints in the `Default` group of a class are the constraints that have either no explicit group configured or that are configured to a group equal to the class name or the string `Default`.

When validating *just* the User object, there is no difference between the `Default` group and the `User` group. But, there is a difference if `User` has embedded objects. For example, imagine `User` has an `address` property that contains some `Address` object and that you've added the *Valid* constraint to this property so that it's validated when you validate the `User` object.

If you validate `User` using the `Default` group, then any constraints on the `Address` class that are in the `Default` group *will* be used. But, if you validate `User` using the `User` validation group, then only constraints on the `Address` class with the `User` group will be validated.

In other words, the `Default` group and the class name group (e.g. `User`) are identical, except when the class is embedded in another object that's actually the one being validated.

If you have inheritance (e.g. `User extends BaseUser`) and you validate with the class name of the subclass (i.e. `User`), then all constraints in the `User` and `BaseUser` will be validated. However, if you validate using the base class (i.e. `BaseUser`), then only the default constraints in the `BaseUser` class will be validated.

To tell the validator to use a specific group, pass one or more group names as the third argument to the `validate()` method:

Listing 13-15
```
$errors = $validator->validate($author, null, array('registration'));
```

If no groups are specified, all constraints that belong to the group `Default` will be applied.

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.

Group Sequence

In some cases, you want to validate your groups by steps. To do this, you can use the `GroupSequence` feature. In this case, an object defines a group sequence, which determines the order groups should be validated.

For example, suppose you have a `User` class and want to validate that the username and the password are different only if all other validation passes (in order to avoid multiple error messages).

Listing 13-16
```
// src/AppBundle/Entity/User.php
namespace AppBundle\Entity;

use Symfony\Component\Security\Core\User\UserInterface;
use Symfony\Component\Validator\Constraints as Assert;

/**
 * @Assert\GroupSequence({"User", "Strict"})
 */
class User implements UserInterface
{
    /**
     * @Assert\NotBlank
     */
    private $username;

    /**
     * @Assert\NotBlank
     */
    private $password;

    /**
     * @Assert\IsTrue(message="The password cannot match your username", groups={"Strict"})
     */
    public function isPasswordLegal()
```

```
26      {
27          return ($this->username !== $this->password);
28      }
29  }
```

In this example, it will first validate all constraints in the group `User` (which is the same as the `Default` group). Only if all constraints in that group are valid, the second group, `Strict`, will be validated.

As you have already seen in the previous section, the `Default` group and the group containing the class name (e.g. `User`) were identical. However, when using Group Sequences, they are no longer identical. The `Default` group will now reference the group sequence, instead of all constraints that do not belong to any group.

This means that you have to use the `{ClassName}` (e.g. `User`) group when specifying a group sequence. When using `Default`, you get an infinite recursion (as the `Default` group references the group sequence, which will contain the `Default` group which references the same group sequence, ...).

Group Sequence Providers

Imagine a `User` entity which can be a normal user or a premium user. When it's a premium user, some extra constraints should be added to the user entity (e.g. the credit card details). To dynamically determine which groups should be activated, you can create a Group Sequence Provider. First, create the entity and a new constraint group called `Premium`:

Listing 13-17
```
1  // src/AppBundle/Entity/User.php
2  namespace AppBundle\Entity;
3
4  use Symfony\Component\Validator\Constraints as Assert;
5
6  class User
7  {
8      /**
9       * @Assert\NotBlank()
10      */
11     private $name;
12
13     /**
14      * @Assert\CardScheme(
15      *     schemes={"VISA"},
16      *     groups={"Premium"},
17      * )
18      */
19     private $creditCard;
20
21     // ...
22 }
```

Now, change the `User` class to implement *`GroupSequenceProviderInterface`*[6] and add the *`getGroupSequence()`*[7], method, which should return an array of groups to use:

Listing 13-18
```
1  // src/AppBundle/Entity/User.php
2  namespace AppBundle\Entity;
3
4  // ...
5  use Symfony\Component\Validator\GroupSequenceProviderInterface;
6
```

6. http://api.symfony.com/3.0/Symfony/Component/Validator/GroupSequenceProviderInterface.html
7. http://api.symfony.com/3.0/Symfony/Component/Validator/GroupSequenceProviderInterface.html#method_getGroupSequence

```php
class User implements GroupSequenceProviderInterface
{
    // ...

    public function getGroupSequence()
    {
        $groups = array('User');

        if ($this->isPremium()) {
            $groups[] = 'Premium';
        }

        return $groups;
    }
}
```

At last, you have to notify the Validator component that your `User` class provides a sequence of groups to be validated:

Listing 13-19

```php
// src/AppBundle/Entity/User.php
namespace AppBundle\Entity;

// ...

/**
 * @Assert\GroupSequenceProvider
 */
class User implements GroupSequenceProviderInterface
{
    // ...
}
```

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 13-20

```php
// ...
use Symfony\Component\Validator\Constraints as Assert;

// ...
public function addEmailAction($email)
{
    $emailConstraint = new Assert\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')->validate(
        $email,
        $emailConstraint
    );

    if (0 === count($errorList)) {
        // ... 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 `validate` 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 `validate` method returns a *`ConstraintViolationList`*[8] object, which acts just like an array of errors. Each error in the collection is a *`ConstraintViolation`*[9] object, which holds the error message on its `getMessage` method.

Final Thoughts

The Symfony `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*

8. http://api.symfony.com/3.0/Symfony/Component/Validator/ConstraintViolationList.html
9. http://api.symfony.com/3.0/Symfony/Component/Validator/ConstraintViolation.html

Chapter 14
Forms

Dealing with HTML forms is one of the most common - and challenging - tasks for a web developer. Symfony 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 Symfony projects. For more information, see the *Form component documentation* 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 14-1

```php
// src/AppBundle/Entity/Task.php
namespace AppBundle\Entity;

class Task
{
    protected $task;
    protected $dueDate;

    public function getTask()
    {
        return $this->task;
    }

    public function setTask($task)
    {
        $this->task = $task;
    }

    public function getDueDate()
    {
        return $this->dueDate;
    }
```

```
23
24      public function setDueDate(\DateTime $dueDate = null)
25      {
26          $this->dueDate = $dueDate;
27      }
28  }
```

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 Symfony, 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 14-2

```
1  // src/AppBundle/Controller/DefaultController.php
2  namespace AppBundle\Controller;
3
4  use AppBundle\Entity\Task;
5  use Symfony\Bundle\FrameworkBundle\Controller\Controller;
6  use Symfony\Component\HttpFoundation\Request;
7  use Symfony\Component\Form\Extension\Core\Type\TextType;
8  use Symfony\Component\Form\Extension\Core\Type\DateType;
9  use Symfony\Component\Form\Extension\Core\Type\SubmitType;
10
11 class DefaultController extends Controller
12 {
13     public function newAction(Request $request)
14     {
15         // create a task and give it some dummy data for this example
16         $task = new Task();
17         $task->setTask('Write a blog post');
18         $task->setDueDate(new \DateTime('tomorrow'));
19
20         $form = $this->createFormBuilder($task)
21             ->add('task', TextType::class)
22             ->add('dueDate', DateType::class)
23             ->add('save', SubmitType::class, array('label' => 'Create Task'))
24             ->getForm();
25
26         return $this->render('default/new.html.twig', array(
27             'form' => $form->createView(),
28         ));
29     }
30 }
```

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 Symfony 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. `TextType` and `DateType`), represented by its fully qualified class name. Among other things, it determines which HTML form tag(s) is rendered for that field.

Finally, you added a submit button with a custom label for submitting the form to the server.

Symfony 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 14-3
```twig
{# app/Resources/views/default/new.html.twig #}
{{ form_start(form) }}
{{ form_widget(form) }}
{{ form_end(form) }}
```

This example assumes that you submit the form in a "POST" request and to the same URL that it was displayed in. You will learn later how to change the request method and the target URL of the form.

That's it! Just three lines are needed to render the complete form:

`form_start(form)`
: Renders the start tag of the form, including the correct enctype attribute when using file uploads.

`form_widget(form)`
: Renders all the fields, which includes the field element itself, a label and any validation error messages for the field.

`form_end(form)`
: Renders the end tag of the form and any fields that have not yet been rendered, in case you rendered each field yourself. This is useful for rendering hidden fields and taking advantage of the automatic CSRF Protection.

> *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" or "hasser" method (e.g. `isPublished()` or `hasReminder()`) instead of a getter (e.g. `getPublished()` or `getReminder()`).

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 written into the Form object. Add the following functionality to your controller:

Listing 14-4

```php
// ...
use Symfony\Component\HttpFoundation\Request;

public function newAction(Request $request)
{
    // just setup a fresh $task object (remove the dummy data)
    $task = new Task();

    $form = $this->createFormBuilder($task)
        ->add('task', TextType::class)
        ->add('dueDate', DateType::class)
        ->add('save', SubmitType::class, array('label' => 'Create Task'))
        ->getForm();

    $form->handleRequest($request);

    if ($form->isSubmitted() && $form->isValid()) {
        // ... perform some action, such as saving the task to the database

        return $this->redirectToRoute('task_success');
    }

    return $this->render('default/new.html.twig', array(
        'form' => $form->createView(),
    ));
}
```

Be aware that the `createView()` method should be called *after* `handleRequest` is called. Otherwise, changes done in the `*_SUBMIT` events aren't applied to the view (like validation errors).

This controller follows a common pattern for handling forms, and has three possible paths:

1. When initially loading the page in a browser, the form is simply created and rendered. *`handleRequest()`*[1] recognizes that the form was not submitted and does nothing. *`isSubmitted()`*[2] returns `false` if the form was not submitted.
2. When the user submits the form, *`handleRequest()`*[3] recognizes this and immediately writes the submitted data back into the `task` and `dueDate` properties of the `$task` object. Then this object is validated. If it is invalid (validation is covered in the next section), *`isValid()`*[4] returns `false`, so the form is rendered together with all validation errors;

1. http://api.symfony.com/3.0/Symfony/Component/Form/FormInterface.html#method_handleRequest
2. http://api.symfony.com/3.0/Symfony/Component/Form/FormInterface.html#method_isSubmitted
3. http://api.symfony.com/3.0/Symfony/Component/Form/FormInterface.html#method_handleRequest
4. http://api.symfony.com/3.0/Symfony/Component/Form/FormInterface.html#method_isValid

3. When the user submits the form with valid data, the submitted data is again written into the form, but this time *`isValid()`*[5] returns `true`. Now 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 the "Refresh" button of their browser and re-post the data.

If you need more control over exactly when your form is submitted or which data is passed to it, you can use the `submit()`[6] for this. Read more about it in the cookbook.

Submitting Forms with Multiple Buttons

When your form contains more than one submit button, you will want to check which of the buttons was clicked to adapt the program flow in your controller. To do this, add a second button with the caption "Save and add" to your form:

Listing 14-5
```php
$form = $this->createFormBuilder($task)
    ->add('task', TextType::class)
    ->add('dueDate', DateType::class)
    ->add('save', SubmitType::class, array('label' => 'Create Task'))
    ->add('saveAndAdd', SubmitType::class, array('label' => 'Save and Add'))
    ->getForm();
```

In your controller, use the button's *`isClicked()`*[7] method for querying if the "Save and add" button was clicked:

Listing 14-6
```php
if ($form->isValid()) {
    // ... perform some action, such as saving the task to the database

    $nextAction = $form->get('saveAndAdd')->isClicked()
        ? 'task_new'
        : 'task_success';

    return $this->redirectToRoute($nextAction);
}
```

Form Validation

In the previous section, you learned how a form can be submitted with valid or invalid data. In Symfony, 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 14-7

5. http://api.symfony.com/3.0/Symfony/Component/Form/FormInterface.html#method_isValid
6. *http://api.symfony.com/3.0/Symfony/Component/Form/FormInterface.html#method_submit*
7. http://api.symfony.com/3.0/Symfony/Component/Form/ClickableInterface.html#method_isClicked

```php
// src/AppBundle/Entity/Task.php
namespace AppBundle\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Task
{
    /**
     * @Assert\NotBlank()
     */
    public $task;

    /**
     * @Assert\NotBlank()
     * @Assert\Type("\DateTime")
     */
    protected $dueDate;
}
```

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.

Listing 14-8
```twig
{# app/Resources/views/default/new.html.twig #}
{{ form(form, {'attr': {'novalidate': 'novalidate'}}) }}
```

Validation is a very powerful feature of Symfony and has its own *dedicated chapter*.

Validation Groups

If your object takes advantage of validation groups, you'll need to specify which validation group(s) your form should use:

Listing 14-9
```php
$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 `configureOptions()` method:

Listing 14-10
```php
use Symfony\Component\OptionsResolver\OptionsResolver;

public function configureOptions(OptionsResolver $resolver)
{
    $resolver->setDefaults(array(
        'validation_groups' => array('registration'),
    ));
}
```

In both of these cases, *only* the `registration` validation group will be used to validate the underlying object.

Disabling Validation

Sometimes it is useful to suppress the validation of a form altogether. For these cases you can set the `validation_groups` option to `false`:

Listing 14-11
```php
use Symfony\Component\OptionsResolver\OptionsResolver;

public function configureOptions(OptionsResolver $resolver)
{
    $resolver->setDefaults(array(
        'validation_groups' => false,
    ));
}
```

Note that when you do that, the form will still run basic integrity checks, for example whether an uploaded file was too large or whether non-existing fields were submitted. If you want to suppress validation, you can use the POST_SUBMIT event.

Groups based on the Submitted Data

If you need some advanced logic to determine the validation groups (e.g. based on submitted data), you can set the `validation_groups` option to an array callback:

Listing 14-12
```php
use Symfony\Component\OptionsResolver\OptionsResolver;

// ...
public function configureOptions(OptionsResolver $resolver)
{
    $resolver->setDefaults(array(
        'validation_groups' => array(
            'AppBundle\Entity\Client',
            'determineValidationGroups',
        ),
    ));
}
```

This will call the static method `determineValidationGroups()` on the `Client` class after the form is submitted, but before validation is executed. The Form object is passed as an argument to that method (see next example). You can also define whole logic inline by using a `Closure`:

Listing 14-13
```php
use AppBundle\Entity\Client;
use Symfony\Component\Form\FormInterface;
use Symfony\Component\OptionsResolver\OptionsResolver;

// ...
public function configureOptions(OptionsResolver $resolver)
{
    $resolver->setDefaults(array(
        'validation_groups' => function (FormInterface $form) {
            $data = $form->getData();

            if (Client::TYPE_PERSON == $data->getType()) {
                return array('person');
            }

            return array('company');
        },
    ));
}
```

Using the `validation_groups` option overrides the default validation group which is being used. If you want to validate the default constraints of the entity as well you have to adjust the option as follows:

Listing 14-14
```php
use AppBundle\Entity\Client;
use Symfony\Component\Form\FormInterface;
use Symfony\Component\OptionsResolver\OptionsResolver;

// ...
public function configureOptions(OptionsResolver $resolver)
{
    $resolver->setDefaults(array(
        'validation_groups' => function (FormInterface $form) {
            $data = $form->getData();

            if (Client::TYPE_PERSON == $data->getType()) {
                return array('Default', 'person');
            }

            return array('Default', 'company');
        },
    ));
}
```

You can find more information about how the validation groups and the default constraints work in the book section about validation groups.

Groups based on the Clicked Button

When your form contains multiple submit buttons, you can change the validation group depending on which button is used to submit the form. For example, consider a form in a wizard that lets you advance to the next step or go back to the previous step. Also assume that when returning to the previous step, the data of the form should be saved, but not validated.

First, we need to add the two buttons to the form:

Listing 14-15
```php
$form = $this->createFormBuilder($task)
    // ...
    ->add('nextStep', SubmitType::class)
    ->add('previousStep', SubmitType::class)
    ->getForm();
```

Then, we configure the button for returning to the previous step to run specific validation groups. In this example, we want it to suppress validation, so we set its `validation_groups` option to false:

Listing 14-16
```php
$form = $this->createFormBuilder($task)
    // ...
    ->add('previousStep', SubmitType::class, array(
        'validation_groups' => false,
    ))
    ->getForm();
```

Now the form will skip your validation constraints. It will still validate basic integrity constraints, such as checking whether an uploaded file was too large or whether you tried to submit text in a number field.

> *To see how to use a service to resolve `validation_groups` dynamically read the How to Dynamically Configure Validation Groups chapter in the cookbook.*

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

- *TextType*
- *TextareaType*
- *EmailType*
- *IntegerType*
- *MoneyType*
- *NumberType*
- *PasswordType*
- *PercentType*
- *SearchType*
- *UrlType*
- *RangeType*

Choice Fields

- *ChoiceType*
- *EntityType*
- *CountryType*
- *LanguageType*
- *LocaleType*
- *TimezoneType*
- *CurrencyType*

Date and Time Fields

- *DateType*
- *DateTimeType*
- *TimeType*
- *BirthdayType*

Other Fields

- *CheckboxType*
- *FileType*
- *RadioType*

Field Groups

- *CollectionType*
- *RepeatedType*

Hidden Fields

- *HiddenType*

Buttons

- *ButtonType*
- *ResetType*
- *SubmitType*

Base Fields

- *FormType*

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 *DateType* can be configured to be rendered as a single text box (where the user would enter the date as a string in the box):

Listing 14-17
```
->add('dueDate', DateType::class, array('widget' => 'single_text'))
```

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 disable HTML5 validation or set the `required` option on your field to `false`:

Listing 14-18
```
->add('dueDate', 'date', array(
    'widget' => 'single_text',
    'required' => false
))
```

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 14-19
```
->add('dueDate', DateType::class, array(
    'widget' => 'single_text',
    'label'  => 'Due Date',
))
```

The label for a field can also be set in the template rendering the form, see below. If you don't need a label associated to your input, you can disable it by setting its value to `false`.

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 `TextType` field and the `dueDate` field is a `DateType` field:

Listing 14-20
```
public function newAction()
{
    $task = new Task();

    $form = $this->createFormBuilder($task)
        ->add('task')
        ->add('dueDate', null, array('widget' => 'single_text'))
        ->add('save', SubmitType::class)
        ->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\Length`). 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 on 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 `Length` or `Range` 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:

Listing 14-21
```
->add('task', null, array('attr' => array('maxlength' => 4)))
```

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:

Listing 14-22
```twig
{# app/Resources/views/default/new.html.twig #}
{{ form_start(form) }}
    {{ form_errors(form) }}

    {{ form_row(form.task) }}
    {{ form_row(form.dueDate) }}
{{ form_end(form) }}
```

You already know the `form_start()` and `form_end()` functions, but what do the other functions do?

`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.

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`:

Listing 14-23
```twig
{{ form.vars.value.task }}
```

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 14-24
```twig
{{ form_start(form) }}
    {{ 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>

    <div>
        {{ form_widget(form.save) }}
    </div>

{{ form_end(form) }}
```

If the auto-generated label for a field isn't quite right, you can explicitly specify it:

Listing 14-25
```
{{ 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 option 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 14-26
```
{{ 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 14-27
```
{{ 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 14-28
```
{{ 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.

Changing the Action and Method of a Form

So far, the `form_start()` helper has been used to render the form's start tag and we assumed that each form is submitted to the same URL in a POST request. Sometimes you want to change these parameters. You can do so in a few different ways. If you build your form in the controller, you can use `setAction()` and `setMethod()`:

Listing 14-29
```
$form = $this->createFormBuilder($task)
    ->setAction($this->generateUrl('target_route'))
    ->setMethod('GET')
    ->add('task', TextType::class)
    ->add('dueDate', DateType::class)
    ->add('save', SubmitType::class)
    ->getForm();
```

This example assumes that you've created a route called `target_route` that points to the controller that processes the form.

In Creating Form Classes you will learn how to move the form building code into separate classes. When using an external form class in the controller, you can pass the action and method as form options:

Listing 14-30
```
use AppBundle\Form\TaskType;
// ...

$form = $this->createForm(TaskType::class, $task, array(
    'action' => $this->generateUrl('target_route'),
    'method' => 'GET',
));
```

Finally, you can override the action and method in the template by passing them to the `form()` or the `form_start()` helper:

Listing 14-31
```twig
{# app/Resources/views/default/new.html.twig #}
{{ form_start(form, {'action': path('target_route'), 'method': 'GET'}) }}
```

If the form's method is not GET or POST, but PUT, PATCH or DELETE, Symfony will insert a hidden field with the name `_method` that stores this method. The form will be submitted in a normal POST request, but Symfony's router is capable of detecting the `_method` parameter and will interpret it as a PUT, PATCH or DELETE request. Read the cookbook chapter "*How to Use HTTP Methods beyond GET and POST in Routes*" for more information.

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 14-32
```php
// src/AppBundle/Form/TaskType.php
namespace AppBundle\Form;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\Form\Extension\Core\Type\SubmitType;

class TaskType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('task')
            ->add('dueDate', null, array('widget' => 'single_text'))
            ->add('save', SubmitType::class)
        ;
    }
}
```

This new class contains all the directions needed to create the task form. It can be used to quickly build a form object in the controller:

Listing 14-33
```php
// src/AppBundle/Controller/DefaultController.php
use AppBundle\Form\TaskType;

public function newAction()
{
    $task = ...;
    $form = $this->createForm(TaskType::class, $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. `AppBundle\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 14-34
```php
use Symfony\Component\OptionsResolver\OptionsResolver;

public function configureOptions(OptionsResolver $resolver)
{
    $resolver->setDefaults(array(
        'data_class' => 'AppBundle\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 `mapped` option to `false`:

Listing 14-35
```php
use Symfony\Component\Form\FormBuilderInterface;

public function buildForm(FormBuilderInterface $builder, array $options)
{
    $builder
        ->add('task')
        ->add('dueDate', null, array('mapped' => false))
        ->add('save', SubmitType::class)
    ;
}
```

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`.

The field data can be accessed in a controller with:

Listing 14-36
```php
$form->get('dueDate')->getData();
```

In addition, the data of an unmapped field can also be modified directly:

Listing 14-37
```php
$form->get('dueDate')->setData(new \DateTime());
```

Defining your Forms as Services

Your form type might have some external dependencies. You can define your form type as a service, and inject all dependencies you need.

Services and the service container will be handled *later on in this book*. Things will be more clear after reading that chapter.

You might want to use a service defined as `app.my_service` in your form type. Create a constructor to your form type to receive the service:

Listing 14-38

```php
// src/AppBundle/Form/Type/TaskType.php
namespace AppBundle\Form\Type;

use App\Utility\MyService;
use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\Form\Extension\Core\Type\SubmitType;

class TaskType extends AbstractType
{
    private $myService;

    public function __construct(MyService $myService)
    {
        $this->myService = $myService;
    }

    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        // You can now use myService.
        $builder
            ->add('task')
            ->add('dueDate', null, array('widget' => 'single_text'))
            ->add('save', SubmitType::class)
        ;
    }
}
```

Define your form type as a service.

Listing 14-39

```yaml
# src/AppBundle/Resources/config/services.yml
services:
    app.form.type.task:
        class: AppBundle\Form\TaskType
        arguments: ["@app.my_service"]
        tags:
            - { name: form.type }
```

Read Creating your Field Type as a Service for more information.

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 14-40

```php
if ($form->isValid()) {
    $em = $this->getDoctrine()->getManager();
    $em->persist($task);
    $em->flush();

    return $this->redirectToRoute('task_success');
}
```

If, for some reason, you don't have access to your original `$task` object, you can fetch it from the form:

Listing 14-41

```php
$task = $form->getData();
```

For more information, see the *Doctrine ORM chapter*.

The key thing to understand is that when the form is submitted, 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 14-42
```php
// src/AppBundle/Entity/Category.php
namespace AppBundle\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 14-43
```php
// ...

class Task
{
    // ...

    /**
     * @Assert\Type(type="AppBundle\Entity\Category")
     * @Assert\Valid()
     */
    protected $category;

    // ...

    public function getCategory()
    {
        return $this->category;
    }

    public function setCategory(Category $category = null)
    {
        $this->category = $category;
    }
}
```

The `Valid` Constraint has been added to the property `category`. This cascades the validation to the corresponding entity. If you omit this constraint the child entity would not be validated.

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 14-44
```php
// src/AppBundle/Form/CategoryType.php
namespace AppBundle\Form;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolver;

class CategoryType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder->add('name');
    }

    public function configureOptions(OptionsResolver $resolver)
    {
        $resolver->setDefaults(array(
            'data_class' => 'AppBundle\Entity\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 14-45
```php
use Symfony\Component\Form\FormBuilderInterface;
use AppBundle\Form\CategoryType;

public function buildForm(FormBuilderInterface $builder, array $options)
{
    // ...

    $builder->add('category', CategoryType::class);
}
```

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 14-46
```twig
{# ... #}

<h3>Category</h3>
<div class="category">
    {{ form_row(form.category.name) }}
</div>

{# ... #}
```

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 *CollectionType* 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, 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 14-47
```twig
{# app/Resources/views/form/fields.html.twig #}
{% block form_row %}
{% spaceless %}
    <div class="form_row">
        {{ form_label(form) }}
        {{ form_errors(form) }}
        {{ form_widget(form) }}
    </div>
{% endspaceless %}
{% endblock form_row %}
```

The `form_row` form fragment is used when rendering most fields via the `form_row` function. To tell the Form component to use your new `form_row` fragment defined above, add the following to the top of the template that renders the form:

Listing 14-48
```twig
{# app/Resources/views/default/new.html.twig #}
{% form_theme form 'form/fields.html.twig' %}

{# or if you want to use multiple themes #}
{% form_theme form 'form/fields.html.twig' 'form/fields2.html.twig' %}

{# ... render the 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 `form_row` block from your custom theme (instead of the default `form_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 (e.g. *form_div_layout.html.twig*[8]) that lives inside the *Twig Bridge*[9]. 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 FrameworkBundle (*view on GitHub*[10]).

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:

- `form_row` - used by `form_row` to render most fields;
- `textarea_widget` - used by `form_widget` to render a `textarea` field type;
- `form_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. `form_label`)	renders the field's label
`widget`	(e.g. `form_widget`)	renders the field's HTML representation
`errors`	(e.g. `form_errors`)	renders the field's errors
`row`	(e.g. `form_row`)	renders the field's entire row (label, widget & errors)

There are actually 2 other *parts* - `rows` and `rest` - 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 `form_errors` fragment. When Symfony renders the errors for a textarea type, it looks first for a `textarea_errors` fragment before falling back to the `form_errors` fragment. Each field type has a *parent* type (the parent type of `textarea` is `text`, its parent is `form`), 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 `form_errors` fragment, rename it to `textarea_errors` and customize it. To override the default error rendering for *all* fields, copy and customize the `form_errors` fragment directly.

The "parent" type of each field type is available in the *form type reference* for each field type.

8. https://github.com/symfony/symfony/blob/master/src/Symfony/Bridge/Twig/Resources/views/Form/form_div_layout.html.twig
9. https://github.com/symfony/symfony/tree/master/src/Symfony/Bridge/Twig
10. https://github.com/symfony/symfony/tree/master/src/Symfony/Bundle/FrameworkBundle/Resources/views/Form

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 14-49
```yaml
# app/config/config.yml
twig:
    form_themes:
        - '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 14-50
```twig
{% extends 'base.html.twig' %}

{# import "_self" as the form theme #}
{% form_theme form _self %}

{# make the form fragment customization #}
{% block form_row %}
    {# custom field row output #}
{% endblock form_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.

This `{% form_theme form _self %}` functionality will *only* work if your template extends another. If your template does not, you must point `form_theme` to a separate template.

PHP

To automatically include the customized templates from the `app/Resources/views/Form` directory created earlier in *all* templates, modify your application configuration file:

Listing 14-51
```yaml
# app/config/config.yml
framework:
    templating:
        form:
            resources:
                - 'Form'
# ...
```

Any fragments inside the `app/Resources/views/Form` directory are now used globally to define form output.

CSRF Protection

CSRF - or *Cross-site request forgery*[11] - 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.

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_end()` function in your template, which ensures that all un-rendered fields are output.

Since the token is stored in the session, a session is started automatically as soon as you render a form with CSRF protection.

The CSRF token can be customized on a form-by-form basis. For example:

Listing 14-52

```
use Symfony\Component\OptionsResolver\OptionsResolver;

class TaskType extends AbstractType
{
    // ...

    public function configureOptions(OptionsResolver $resolver)
    {
        $resolver->setDefaults(array(
            'data_class'      => 'AppBundle\Entity\Task',
            'csrf_protection' => true,
            'csrf_field_name' => '_token',
            // a unique key to help generate the secret token
            'csrf_token_id'   => '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 `csrf_token_id` option is optional but greatly enhances the security of the generated token by making it different for each form.

CSRF tokens are meant to be different for every user. This is why you need to be cautious if you try to cache pages with forms including this kind of protection. For more information, see *Caching Pages that Contain CSRF Protected Forms*.

11. http://en.wikipedia.org/wiki/Cross-site_request_forgery

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 14-53

```
// 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', TextType::class)
        ->add('email', EmailType::class)
        ->add('message', TextareaType::class)
        ->add('send', SubmitType::class)
        ->getForm();

    $form->handleRequest($request);

    if ($form->isValid()) {
        // 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:

Listing 14-54

```
$request->request->get('name');
```

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 component.

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. If your form is mapped to an object (i.e. you're using the `data_class` option or passing an object to your form), this is almost always the approach you want to use. See *Validation* for more details.

But if the form is not mapped to an object and you instead want to retrieve a simple array of your submitted data, how can you add constraints to the data of your form?

The answer is to setup the constraints yourself, and attach them to the individual fields. The overall approach is covered a bit more in the validation chapter, but here's a short example:

Listing 14-55

```php
use Symfony\Component\Validator\Constraints\Length;
use Symfony\Component\Validator\Constraints\NotBlank;
use Symfony\Component\Form\Extension\Core\Type\TextType;

$builder
    ->add('firstName', TextType::class, array(
        'constraints' => new Length(array('min' => 3)),
    ))
    ->add('lastName', TextType::class, array(
        'constraints' => array(
            new NotBlank(),
            new Length(array('min' => 3)),
        ),
    ))
;
```

If you are using validation groups, you need to either reference the `Default` group when creating the form, or set the correct group on the constraint you are adding.

Listing 14-56
```php
new NotBlank(array('groups' => array('create', 'update'))
```

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* 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 Upload Files*
- *File Field Reference*
- *Creating Custom Field Types*
- *How to Customize Form Rendering*
- *How to Dynamically Modify Forms Using Form Events*
- *How to Use Data Transformers*
- *Using CSRF Protection in the Login Form*
- *Caching Pages that Contain CSRF Protected Forms*

Chapter 15
Security

Symfony's security system is incredibly powerful, but it can also be confusing to set up. In this chapter, you'll learn how to set up your application's security step-by-step, from configuring your firewall and how you load users to denying access and fetching the User object. Depending on what you need, sometimes the initial setup can be tough. But once it's done, Symfony's security system is both flexible and (hopefully) fun to work with.

Since there's a lot to talk about, this chapter is organized into a few big sections:

1. Initial `security.yml` setup (*authentication*);
2. Denying access to your app (*authorization*);
3. Fetching the current User object.

These are followed by a number of small (but still captivating) sections, like logging out and encoding user passwords.

1) Initial security.yml Setup (Authentication)

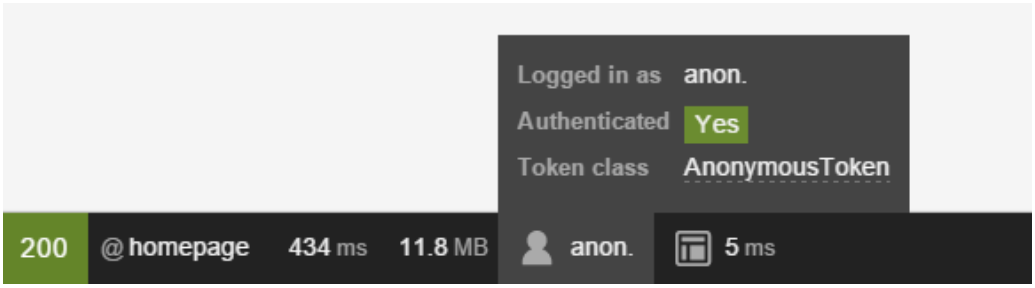
The security system is configured in `app/config/security.yml`. The default configuration looks like this:

Listing 15-1
```yaml
# app/config/security.yml
security:
    providers:
        in_memory:
            memory: ~

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

        default:
            anonymous: ~
```

The `firewalls` key is the *heart* of your security configuration. The `dev` firewall isn't important, it just makes sure that Symfony's development tools - which live under URLs like `/_profiler` and `/_wdt` aren't blocked by your security.

You can also match a request against other details of the request (e.g. host). For more information and examples read *How to Restrict Firewalls to a Specific Request*.

All other URLs will be handled by the `default` firewall (no `pattern` key means it matches *all* URLs). You can think of the firewall like your security system, and so it usually makes sense to have just one main firewall. But this does *not* mean that every URL requires authentication - the `anonymous` key takes care of this. In fact, if you go to the homepage right now, you'll have access and you'll see that you're "authenticated" as `anon.`. Don't be fooled by the "Yes" next to Authenticated, you're just an anonymous user:



You'll learn later how to deny access to certain URLs or controllers.

Security is *highly* configurable and there's a *Security Configuration Reference* that shows all of the options with some extra explanation.

A) Configuring how your Users will Authenticate

The main job of a firewall is to configure *how* your users will authenticate. Will they use a login form? HTTP basic authentication? An API token? All of the above?

Let's start with HTTP basic authentication (the old-school prompt) and work up from there. To activate this, add the `http_basic` key under your firewall:

Listing 15-2
```yaml
# app/config/security.yml
security:
    # ...

    firewalls:
        # ...
        default:
            anonymous: ~
            http_basic: ~
```

Simple! To try this, you need to require the user to be logged in to see a page. To make things interesting, create a new page at `/admin`. For example, if you use annotations, create something like this:

Listing 15-3
```php
// src/AppBundle/Controller/DefaultController.php
// ...

use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Response;

class DefaultController extends Controller
{
```

```
    /**
     * @Route("/admin")
     */
    public function adminAction()
    {
        return new Response('<html><body>Admin page!</body></html>');
    }
}
```

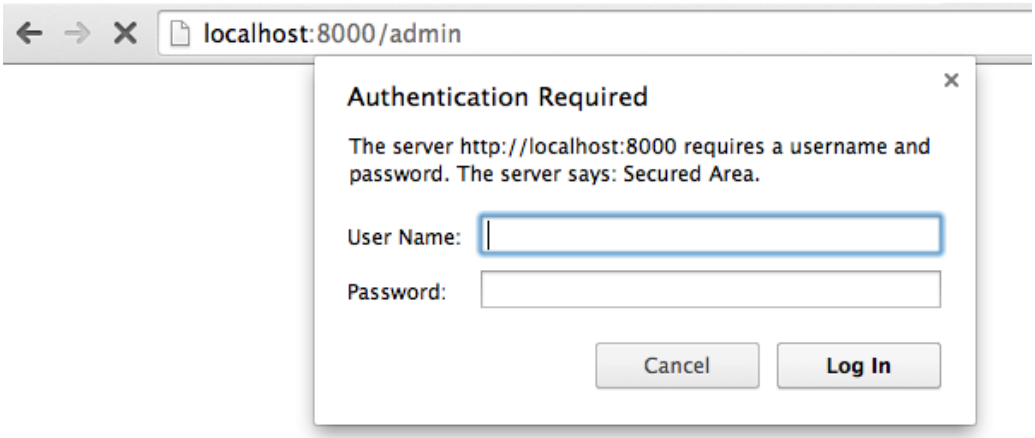
Next, add an `access_control` entry to `security.yml` that requires the user to be logged in to access this URL:

Listing 15-4

```
# app/config/security.yml
security:
    # ...
    firewalls:
        # ...
        default:
            # ...

    access_control:
        # require ROLE_ADMIN for /admin*
        - { path: ^/admin, roles: ROLE_ADMIN }
```

You'll learn more about this `ROLE_ADMIN` thing and denying access later in the 2) Denying Access, Roles and other Authorization section.

Great! Now, if you go to `/admin`, you'll see the HTTP basic auth prompt:



But who can you login as? Where do users come from?

Want to use a traditional login form? Great! See *How to Build a Traditional Login Form*. What other methods are supported? See the *Configuration Reference* or *build your own*.

If your application logs users in via a third-party service such as Google, Facebook or Twitter, check out the *HWIOAuthBundle*[1] community bundle.

B) Configuring how Users are Loaded

When you type in your username, Symfony needs to load that user's information from somewhere. This is called a "user provider", and you're in charge of configuring it. Symfony has a built-in way to *load users from the database*, or you can *create your own user provider*.

The easiest (but most limited) way, is to configure Symfony to load hardcoded users directly from the `security.yml` file itself. This is called an "in memory" provider, but it's better to think of it as an "in configuration" provider:

Listing 15-5
```yaml
# app/config/security.yml
security:
    providers:
        in_memory:
            memory:
                users:
                    ryan:
                        password: ryanpass
                        roles: 'ROLE_USER'
                    admin:
                        password: kitten
                        roles: 'ROLE_ADMIN'
    # ...
```

Like with `firewalls`, you can have multiple `providers`, but you'll probably only need one. If you *do* have multiple, you can configure which *one* provider to use for your firewall under its `provider` key (e.g. `provider: in_memory`).

> *See How to Use multiple User Providers for all the details about multiple providers setup.*

Try to login using username `admin` and password `kitten`. You should see an error!

> No encoder has been configured for account "Symfony\Component\Security\Core\User\User"

To fix this, add an `encoders` key:

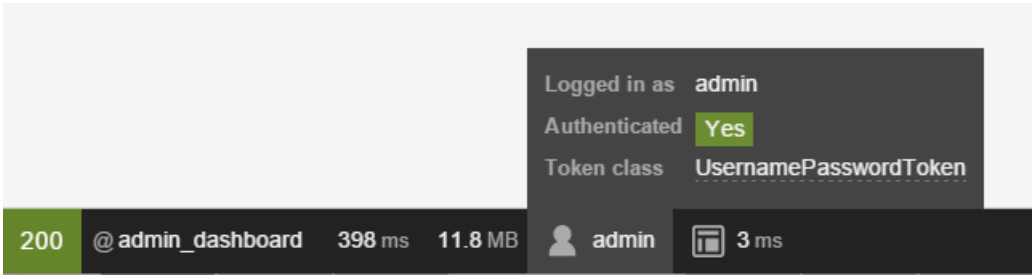
Listing 15-6
```yaml
# app/config/security.yml
security:
    # ...

    encoders:
        Symfony\Component\Security\Core\User\User: plaintext
    # ...
```

User providers load user information and put it into a `User` object. If you *load users from the database* or *some other source*, you'll use your own custom User class. But when you use the "in memory" provider, it gives you a `Symfony\Component\Security\Core\User\User` object.

Whatever your User class is, you need to tell Symfony what algorithm was used to encode the passwords. In this case, the passwords are just plaintext, but in a second, you'll change this to use `bcrypt`.

If you refresh now, you'll be logged in! The web debug toolbar even tells you who you are and what roles you have:

1. https://github.com/hwi/HWIOAuthBundle



Because this URL requires `ROLE_ADMIN`, if you had logged in as `ryan`, this would deny you access. More on that later (Securing URL patterns (access_control)).

Loading Users from the Database

If you'd like to load your users via the Doctrine ORM, that's easy! See *How to Load Security Users from the Database (the Entity Provider)* for all the details.

C) Encoding the User's Password

Whether your users are stored in `security.yml`, in a database or somewhere else, you'll want to encode their passwords. The best algorithm to use is `bcrypt`:

Listing 15-7
```yaml
# app/config/security.yml
security:
    # ...

    encoders:
        Symfony\Component\Security\Core\User\User:
            algorithm: bcrypt
            cost: 12
```

Of course, your users' passwords now need to be encoded with this exact algorithm. For hardcoded users, you can use the built-in command:

Listing 15-8
```
$ php bin/console security:encode-password
```

It will give you something like this:

Listing 15-9
```yaml
# app/config/security.yml
security:
    # ...

    providers:
        in_memory:
            memory:
                users:
                    ryan:
                        password: $2a$12$LCYOMefVIEc3TYPHV9SNnuzOfyr2p/AXIGoQJEDs4am4JwhNz/jli
                        roles: 'ROLE_USER'
                    admin:
                        password: $2a$12$cyTWeE9kpq1PjqKFiWUZFuCRPwVyAZwm4XzMZ1qPUFl7/flCM3VOG
                        roles: 'ROLE_ADMIN'
```

Everything will now work exactly like before. But if you have dynamic users (e.g. from a database), how can you programmatically encode the password before inserting them into the database? Don't worry, see Dynamically Encoding a Password for details.

Supported algorithms for this method depend on your PHP version, but include the algorithms returned by the PHP function *hash_algos*[2] as well as a few others (e.g. bcrypt). See the `encoders` key in the *Security Reference Section* for examples.

It's also possible to use different hashing algorithms on a user-by-user basis. See *How to Choose the Password Encoder Algorithm Dynamically* for more details.

D) Configuration Done!

Congratulations! You now have a working authentication system that uses HTTP basic auth and loads users right from the `security.yml` file.

Your next steps depend on your setup:

- Configure a different way for your users to login, like a login form or *something completely custom*;
- Load users from a different source, like the *database* or *some other source*;
- Learn how to deny access, load the User object and deal with roles in the Authorization section.

2) Denying Access, Roles and other Authorization

Users can now login to your app using `http_basic` or some other method. Great! Now, you need to learn how to deny access and work with the User object. This is called **authorization**, and its job is to decide if a user can access some resource (a URL, a model object, a method call, ...).

The process of authorization has two different sides:

1. The user receives a specific set of roles when logging in (e.g. `ROLE_ADMIN`).
2. You add code so that a resource (e.g. URL, controller) requires a specific "attribute" (most commonly a role like `ROLE_ADMIN`) in order to be accessed.

In addition to roles (e.g. `ROLE_ADMIN`), you can protect a resource using other attributes/strings (e.g. `EDIT`) and use voters or Symfony's ACL system to give these meaning. This might come in handy if you need to check if user A can "EDIT" some object B (e.g. a Product with id 5). See Access Control Lists (ACLs): Securing individual Database Objects.

Roles

When a user logs in, they receive a set of roles (e.g. `ROLE_ADMIN`). In the example above, these are hardcoded into `security.yml`. If you're loading users from the database, these are probably stored on a column in your table.

All roles you assign to a user **must** begin with the `ROLE_` prefix. Otherwise, they won't be handled by Symfony's security system in the normal way (i.e. unless you're doing something advanced, assigning a role like `FOO` to a user and then checking for `FOO` as described below will not work).

Roles are simple, and are basically strings that you invent and use as needed. 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.

2. http://php.net/manual/en/function.hash-algos.php

Make sure every user has at least *one* role, or your user will look like they're not authenticated. A common convention is to give *every* user `ROLE_USER`.

You can also specify a role hierarchy where some roles automatically mean that you also have other roles.

Add Code to Deny Access

There are **two** ways to deny access to something:

1. access_control in security.yml allows you to protect URL patterns (e.g. `/admin/*`). This is easy, but less flexible;
2. in your code via the security.authorization_checker service.

Securing URL patterns (access_control)

The most basic way to secure part of your application is to secure an entire URL pattern. You saw this earlier, where anything matching the regular expression `^/admin` requires the `ROLE_ADMIN` role:

Listing 15-10

```yaml
# app/config/security.yml
security:
    # ...

    firewalls:
        # ...
        default:
            # ...

    access_control:
        # require ROLE_ADMIN for /admin*
        - { path: ^/admin, roles: ROLE_ADMIN }
```

This is great for securing entire sections, but you'll also probably want to secure your individual controllers as well.

You can define as many URL patterns as you need - each is a regular expression. **BUT**, only **one** will be matched. Symfony will look at each starting at the top, and stop as soon as it finds one `access_control` entry that matches the URL.

Listing 15-11

```yaml
# app/config/security.yml
security:
    # ...

    access_control:
        - { path: ^/admin/users, roles: ROLE_SUPER_ADMIN }
        - { path: ^/admin, roles: ROLE_ADMIN }
```

Prepending the path with `^` means that only URLs *beginning* with the pattern are matched. For example, a path of simply `/admin` (without the `^`) would match `/admin/foo` but would also match URLs like `/foo/admin`.

Understanding how `access_control` Works

The `access_control` section is very powerful, but it can also be dangerous (because it involves security) if you don't understand *how* it works. In addition to the URL, the `access_control` can match on IP address, host name and HTTP methods. It can also be used to redirect a user to the `https` version of a URL pattern.

To learn about all of this, see *How Does the Security access_control Work?*.

Securing Controllers and other Code

You can easily deny access from inside a controller:

Listing 15-12
```php
// ...

public function helloAction($name)
{
    // The second parameter is used to specify on what object the role is tested.
    $this->denyAccessUnlessGranted('ROLE_ADMIN', null, 'Unable to access this page!');

    // Old way :
    // if (false === $this->get('security.authorization_checker')->isGranted('ROLE_ADMIN')) {
    //     throw $this->createAccessDeniedException('Unable to access this page!');
    // }

    // ...
}
```

In both cases, a special *AccessDeniedException*[3] is thrown, which ultimately triggers a 403 HTTP response inside Symfony.

That's it! If the user isn't logged in yet, they will be asked to login (e.g. redirected to the login page). If they *are* logged in, but do *not* have the `ROLE_ADMIN` role, they'll be shown the 403 access denied page (which you can customize). If they are logged in and have the correct roles, the code will be executed.

Thanks to the SensioFrameworkExtraBundle, you can also secure your controller using annotations:

Listing 15-13
```php
// ...
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Security;

/**
 * @Security("has_role('ROLE_ADMIN')")
 */
public function helloAction($name)
{
    // ...
}
```

For more information, see the *FrameworkExtraBundle documentation*[4].

Access Control in Templates

If you want to check if the current user has a role inside a template, use the built-in `is_granted()` helper function:

Listing 15-14
```twig
{% if is_granted('ROLE_ADMIN') %}
    <a href="...">Delete</a>
{% endif %}
```

Securing other Services

Anything in Symfony can be protected by doing something similar to the code used to secure a controller. For example, suppose you have a service (i.e. a PHP class) whose job is to send emails. You can restrict use of this class - no matter where it's being used from - to only certain users.

For more information see *How to Secure any Service or Method in your Application*.

3. http://api.symfony.com/3.0/Symfony/Component/Security/Core/Exception/AccessDeniedException.html
4. https://symfony.com/doc/current/bundles/SensioFrameworkExtraBundle/index.html

Checking to see if a User is Logged In (IS_AUTHENTICATED_FULLY)

So far, you've checked access based on roles - those strings that start with `ROLE_` and are assigned to users. But if you *only* want to check if a user is logged in (you don't care about roles), then you can use `IS_AUTHENTICATED_FULLY`:

Listing 15-15
```
// ...

public function helloAction($name)
{
    if (!$this->get('security.authorization_checker')->isGranted('IS_AUTHENTICATED_FULLY')) {
        throw $this->createAccessDeniedException();
    }

    // ...
}
```

You can of course also use this in `access_control`.

`IS_AUTHENTICATED_FULLY` isn't a role, but it kind of acts like one, and every user that has successfully logged in will have this. In fact, there are three special attributes like this:

- `IS_AUTHENTICATED_REMEMBERED`: *All* logged in users have this, even if they are logged in because of a "remember me cookie". Even if you don't use the *remember me functionality*, you can use this to check if the user is logged in.
- `IS_AUTHENTICATED_FULLY`: This is similar to `IS_AUTHENTICATED_REMEMBERED`, but stronger. Users who are logged in only because of a "remember me cookie" will have `IS_AUTHENTICATED_REMEMBERED` but will not have `IS_AUTHENTICATED_FULLY`.
- `IS_AUTHENTICATED_ANONYMOUSLY`: *All* users (even anonymous ones) have this - this is useful when *whitelisting* URLs to guarantee access - some details are in *How Does the Security access_control Work?*.

You can also use expressions inside your templates:

Listing 15-16
```
{% if is_granted(expression(
    '"ROLE_ADMIN" in roles or (user and user.isSuperAdmin())'
)) %}
    <a href="...">Delete</a>
{% endif %}
```

For more details on expressions and security, see Security: Complex Access Controls with Expressions.

Access Control Lists (ACLs): Securing individual Database Objects

Imagine you are designing a blog where users can comment on your posts. You also want a user to be able to edit their own comments, but not those of other users. Also, as the admin user, you yourself want to be able to edit *all* comments.

To accomplish this you have 2 options:

- *Voters* allow you to write own business logic (e.g. the user can edit this post because they were the creator) to determine access. You'll probably want this option - it's flexible enough to solve the above situation.
- *ACLs* allow you to create a database structure where you can assign *any* arbitrary user *any* access (e.g. EDIT, VIEW) to *any* object in your system. Use this if you need an admin user to be able to grant customized access across your system via some admin interface.

In both cases, you'll still deny access using methods similar to what was shown above.

Retrieving the User Object

After authentication, the `User` object of the current user can be accessed via the `security.token_storage` service. From inside a controller, this will look like:

Listing 15-17
```php
public function indexAction()
{
    if (!$this->get('security.authorization_checker')->isGranted('IS_AUTHENTICATED_FULLY')) {
        throw $this->createAccessDeniedException();
    }

    $user = $this->getUser();

    // the above is a shortcut for this
    $user = $this->get('security.token_storage')->getToken()->getUser();
}
```

The user will be an object and the class of that object will depend on your user provider.

Now you can call whatever methods are on *your* User object. For example, if your User object has a `getFirstName()` method, you could use that:

Listing 15-18
```php
use Symfony\Component\HttpFoundation\Response;
// ...

public function indexAction()
{
    // ...

    return new Response('Well hi there '.$user->getFirstName());
}
```

Always Check if the User is Logged In

It's important to check if the user is authenticated first. If they're not, `$user` will either be `null` or the string `anon.`. Wait, what? Yes, this is a quirk. If you're not logged in, the user is technically the string `anon.`, though the `getUser()` controller shortcut converts this to `null` for convenience.

The point is this: always check to see if the user is logged in before using the User object, and use the `isGranted` method (or access_control) to do this:

Listing 15-19
```php
// yay! Use this to see if the user is logged in
if (!$this->get('security.authorization_checker')->isGranted('IS_AUTHENTICATED_FULLY')) {
    throw $this->createAccessDeniedException();
}

// boo :(. Never check for the User object to see if they're logged in
if ($this->getUser()) {

}
```

Retrieving the User in a Template

In a Twig Template this object can be accessed via the app.user key:

Listing 15-20
```twig
{% if is_granted('IS_AUTHENTICATED_FULLY') %}
    <p>Username: {{ app.user.username }}</p>
{% endif %}
```

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 15-21
```yaml
# app/config/security.yml
security:
    # ...

    firewalls:
        secured_area:
            # ...
            logout:
                path:   /logout
                target: /
```

Next, you'll need to create a route for this URL (but not a controller):

Listing 15-22
```yaml
# app/config/routing.yml
logout:
    path: /logout
```

And that's it! By sending a user to `/logout` (or whatever you configure the `path` to be), Symfony will un-authenticate the current user.

Once the user has been logged out, they will be redirected to whatever path is defined by the `target` parameter above (e.g. the `homepage`).

If you need to do something more interesting after logging out, you can specify a logout success handler by adding a `success_handler` key and pointing it to a service id of a class that implements *`LogoutSuccessHandlerInterface`*[5]. See *Security Configuration Reference*.

Notice that when using http-basic authenticated firewalls, there is no real way to log out : the only way to *log out* is to have the browser stop sending your name and password on every request. Clearing your browser cache or restarting your browser usually helps. Some web developer tools might be helpful here too.

Dynamically Encoding a Password

For historical reasons, Symfony uses the term *"password encoding"* when it should really refer to *"password hashing"*. The "encoders" are in fact *cryptographic hash functions*[6].

If, for example, you're storing users in the database, you'll need to encode the users' passwords before inserting them. No matter what algorithm you configure for your user object, the hashed password can always be determined in the following way from a controller:

5. http://api.symfony.com/3.0/Symfony/Component/Security/Http/Logout/LogoutSuccessHandlerInterface.html
6. https://en.wikipedia.org/wiki/Cryptographic_hash_function

Listing 15-23
```php
// whatever *your* User object is
$user = new AppBundle\Entity\User();
$plainPassword = 'ryanpass';
$encoder = $this->container->get('security.password_encoder');
$encoded = $encoder->encodePassword($user, $plainPassword);

$user->setPassword($encoded);
```

In order for this to work, just make sure that you have the encoder for your user class (e.g. `AppBundle\Entity\User`) configured under the `encoders` key in `app/config/security.yml`.

The `$encoder` object also has an `isPasswordValid` method, which takes the `User` object as the first argument and the plain password to check as the second argument.

When you allow a user to submit a plaintext password (e.g. registration form, change password form), you *must* have validation that guarantees that the password is 4096 characters or fewer. Read more details in How to implement a simple Registration Form.

Hierarchical Roles

Instead of associating many roles to users, you can define role inheritance rules by creating a role hierarchy:

Listing 15-24
```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`).

Stateless Authentication

By default, Symfony 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 Symfony):

Listing 15-25
```yaml
# app/config/security.yml
security:
    # ...

    firewalls:
        main:
            http_basic: ~
            stateless:  true
```

If you use a form login, Symfony will create a cookie even if you set `stateless` to `true`.

Checking for Known Security Vulnerabilities in Dependencies

When using lots of dependencies in your Symfony projects, some of them may contain security vulnerabilities. That's why Symfony includes a command called `security:check` that checks your `composer.lock` file to find any known security vulnerability in your installed dependencies:

Listing 15-26
```
$ php bin/console security:check
```

A good security practice is to execute this command regularly to be able to update or replace compromised dependencies as soon as possible. Internally, this command uses the public *security advisories database*[7] published by the FriendsOfPHP organization.

The `security:check` command terminates with a non-zero exit code if any of your dependencies is affected by a known security vulnerability. Therefore, you can easily integrate it in your build process.

To enable the `security:check` command, make sure the *SensioDistributionBundle*[8] is installed.

Listing 15-27
```
$ composer require 'sensio/distribution-bundle'
```

Final Words

Woh! Nice work! You now know more than the basics of security. The hardest parts are when you have custom requirements: like a custom authentication strategy (e.g. API tokens), complex authorization logic and many other things (because security is complex!).

Fortunately, there are a lot of *Security Cookbook Articles* aimed at describing many of these situations. Also, see the *Security Reference Section*. Many of the options don't have specific details, but seeing the full possible configuration tree may be useful.

Good luck!

Learn More from the Cookbook

- *Forcing HTTP/HTTPS*
- *Impersonating a User*
- *How to Use Voters to Check User Permissions*
- *Access Control Lists (ACLs)*
- *How to Add "Remember Me" Login Functionality*
- *How to Use multiple User Providers*

7. https://github.com/FriendsOfPHP/security-advisories
8. https://packagist.org/packages/sensio/distribution-bundle

Chapter 16
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. Symfony 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, you'll see how the Symfony cache system works and why this is the best possible approach.

The Symfony cache system is different because it relies on the simplicity and power of the HTTP cache as defined in the *HTTP specification*[1]. Instead of reinventing a caching methodology, Symfony 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 Symfony cache system.

For the purposes of learning how to cache with Symfony, the subject is covered in four steps:

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. Symfony provides its own reverse proxy, but any reverse proxy can be used.
2. HTTP cache headers are used to communicate with the gateway cache and any other caches between your application and the client. Symfony provides sensible defaults and a powerful interface for interacting with the cache headers.
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).

1. `http://www.w3.org/Protocols/rfc2616/rfc2616.html`

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, Ryan Tomayko's article *Things Caches Do*[2] is *highly* recommended . Another in-depth resource is Mark Nottingham's *Cache Tutorial*[3].

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*[4], *Squid in reverse proxy mode*[5], and the Symfony reverse proxy.

Types of Caches

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 when caching responses containing content that is specific to exactly one user (e.g. account information) is discussed.

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.

2. http://2ndscale.com/writings/things-caches-do
3. http://www.mnot.net/cache_docs/
4. https://www.varnish-cache.org/
5. http://wiki.squid-cache.org/SquidFaq/ReverseProxy

Symfony Reverse Proxy

Symfony 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 Symfony 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 16-1
```php
// web/app.php
use Symfony\Component\HttpFoundation\Request;

// ...
$kernel = new AppKernel('prod', false);
$kernel->loadClassCache();
// wrap the default AppKernel with the AppCache one
$kernel = new AppCache($kernel);

$request = Request::createFromGlobals();

$response = $kernel->handle($request);
$response->send();

$kernel->terminate($request, $response);
```

The caching kernel will immediately act as a reverse proxy - caching responses from your application and returning them to the client.

If you're using the framework.http_method_override option to read the HTTP method from a `_method` parameter, see the above link for a tweak you need to make.

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 16-2
```php
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()`*[6] method:

Listing 16-3
```php
// 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,
        );
    }
}
```

6. http://api.symfony.com/3.0/Symfony/Bundle/FrameworkBundle/HttpCache/HttpCache.html#method_getOptions

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`, Symfony automatically adds an `X-Symfony-Cache` header to the response containing useful information about cache hits and misses.

Changing from one Reverse Proxy to another

The Symfony 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 it is highly recommended you 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 Symfony reverse proxy and upgrade later to Varnish when your traffic increases.

For more information on using Varnish with Symfony, see the *How to use Varnish* cookbook chapter.

The performance of the Symfony 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. HTTP caching is the part of that language that allows clients and servers to exchange information related to caching.

HTTP specifies four response cache headers that are looked at here:

- `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, Validation and Invalidation 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:

Listing 16-4
```
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 16-5
```
// ...

use Symfony\Component\HttpFoundation\Response;

$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);
```

If you need to set cache headers for many different controller actions, you might want to look into the *FOSHttpCacheBundle*[7]. It provides a way to define cache headers based on the URL pattern and other request properties.

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 Symfony 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 means 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.

Symfony 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, Symfony adds the `private` directive automatically (except when `s-maxage` is set).

HTTP Expiration, Validation and Invalidation

The HTTP specification defines two caching models:

7. `http://foshttpcachebundle.readthedocs.org/`

- With the *expiration model*[8], 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*[9] 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 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. To achieve long caching times but still provide updated content immediately, *cache invalidation* is sometimes used.

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 your work. Unfortunately, the actual specification document - *RFC 2616*[10] - can be difficult to read.

There is an ongoing effort (*HTTP Bis*[11]) 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*[12] and P6 - Caching: Browser and intermediary caches).

As a web developer, you are strongly urged 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 16-6
```
$date = new DateTime();
$date->modify('+600 seconds');

$response->setExpires($date);
```

The resulting HTTP header will look like this:

Listing 16-7
```
1  Expires: Thu, 01 Mar 2011 16:00:00 GMT
```

8. http://tools.ietf.org/html/rfc2616#section-13.2
9. http://tools.ietf.org/html/rfc2616#section-13.3
10. http://tools.ietf.org/html/rfc2616
11. http://tools.ietf.org/wg/httpbis/
12. http://tools.ietf.org/html/draft-ietf-httpbis-p4-conditional

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 on the local clock to evaluate the `Expires` header making the lifetime calculation vulnerable to clock skew. Another limitation of the `Expires` 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 16-8
```
// 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 16-9
```
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 if the cached response is still valid or if it needs to be regenerated. 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 only save CPU if you're able to determine that the cached response is still valid by doing *less* work than generating the whole page again (see below for an implementation example).

The 304 status code means "Not Modified". It's important because with this status code the response does *not* contain the actual content being requested. Instead, the response is simply a light-weight set of directions that tells the 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.

To see a simple implementation, generate the ETag as the md5 of the content:

Listing 16-10
```php
// src/AppBundle/Controller/DefaultController.php
namespace AppBundle\Controller;

use Symfony\Component\HttpFoundation\Request;

class DefaultController extends Controller
{
    public function homepageAction(Request $request)
    {
        $response = $this->render('static/homepage.html.twig');
        $response->setETag(md5($response->getContent()));
        $response->setPublic(); // make sure the response is public/cacheable
        $response->isNotModified($request);

        return $response;
    }
}
```

The *`isNotModified()`*[13] method compares the `If-None-Match` sent with the `Request` with the `ETag` header set on the `Response`. If the two match, the method automatically sets the `Response` status code to 304.

The cache sets the `If-None-Match` header on the request to the `ETag` of the original cached response before sending the request back to the app. This is how the cache and server communicate with each other and decide whether or not the resource has been updated since it was cached.

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, you'll see how validation can be used more intelligently to determine the validity of a cache without doing so much work.

Symfony also supports weak ETags by passing `true` as the second argument to the *`setEtag()`*[14] 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 16-11
```php
// src/AppBundle/Controller/ArticleController.php
namespace AppBundle\Controller;

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

13. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Response.html#method_isNotModified
14. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Response.html#method_setEtag

```php
use Symfony\Component\HttpFoundation\Request;
use AppBundle\Entity\Article;

class ArticleController extends Controller
{
    public function showAction(Article $article, Request $request)
    {
        $author = $article->getAuthor();

        $articleDate = new \DateTime($article->getUpdatedAt());
        $authorDate = new \DateTime($author->getUpdatedAt());

        $date = $authorDate > $articleDate ? $authorDate : $articleDate;

        $response = new Response();
        $response->setLastModified($date);
        // Set response as public. Otherwise it will be private by default.
        $response->setPublic();

        if ($response->isNotModified($request)) {
            return $response;
        }

        // ... do more work to populate the response with the full content

        return $response;
    }
}
```

The *`isNotModified()`*[15] 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 cache sets the `If-Modified-Since` header on the request to the `Last-Modified` of the original cached response before sending the request back to the app. This is how the cache and server communicate with each other and decide whether or not the resource has been updated since it was cached.

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 16-12

```php
// src/AppBundle/Controller/ArticleController.php
namespace AppBundle\Controller;

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

class ArticleController extends Controller
{
    public function showAction($articleSlug, Request $request)
    {
        // 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 an ETag and/or a Last-Modified header
```

15. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Response.html#method_isNotModified

```
        $response = new Response();
        $response->setETag($article->computeETag());
        $response->setLastModified($article->getPublishedAt());

        // Set response as public. Otherwise it will be private by default.
        $response->setPublic();

        // Check that the Response is not modified for the given Request
        if ($response->isNotModified($request)) {
            // return the 304 Response immediately
            return $response;
        }

        // do more work here - like retrieving more data
        $comments = ...;

        // or render a template with the $response you've already started
        return $this->render('article/show.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()`*[16]).

Varying the Response

So far, it's been 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, you 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 16-13
```
Vary: Accept-Encoding, User-Agent
```

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 16-14
```
// 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.

16. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Response.html#method_setNotModified

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.

You can also define HTTP caching headers for expiration and validation by using annotations. See the *FrameworkExtraBundle documentation*[17].

More Response Methods

The Response class provides many more methods related to the cache. Here are the most useful ones:

Listing 16-15
```
// 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()`*[18] method:

Listing 16-16
```
// Set cache settings in one call
$response->setCache(array(
    'etag'          => $etag,
    'last_modified' => $date,
    'max_age'       => 10,
    's_maxage'      => 10,
    'public'        => true,
    // 'private'    => true,
));
```

Cache Invalidation

> "There are only two hard things in Computer Science: cache invalidation and naming things." -- Phil Karlton

Once an URL is cached by a gateway cache, the cache will not ask the application for that content anymore. This allows the cache to provide fast responses and reduces the load on your application. However, you risk delivering outdated content. A way out of this dilemma is to use long cache lifetimes, but to actively notify the gateway cache when content changes. Reverse proxies usually provide a channel to receive such notifications, typically through special HTTP requests.

While cache invalidation is powerful, avoid it when possible. If you fail to invalidate something, outdated caches will be served for a potentially long time. Instead, use short cache lifetimes or use the validation model, and adjust your controllers to perform efficient validation checks as explained in Optimizing your Code with Validation.

Furthermore, since invalidation is a topic specific to each type of reverse proxy, using this concept will tie you to a specific reverse proxy or need additional efforts to support different proxies.

17. https://symfony.com/doc/current/bundles/SensioFrameworkExtraBundle/annotations/cache.html
18. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Response.html#method_setCache

Sometimes, however, you need that extra performance you can get when explicitly invalidating. For invalidation, your application needs to detect when content changes and tell the cache to remove the URLs which contain that data from its cache.

If you want to use cache invalidation, have a look at the *FOSHttpCacheBundle*[19]. This bundle provides services to help with various cache invalidation concepts and also documents the configuration for a couple of common caching proxies.

If one content corresponds to one URL, the PURGE model works well. You send a request to the cache proxy with the HTTP method PURGE (using the word "PURGE" is a convention, technically this can be any string) instead of GET and make the cache proxy detect this and remove the data from the cache instead of going to the application to get a response.

Here is how you can configure the Symfony reverse proxy to support the PURGE HTTP method:

Listing 16-17
```php
// app/AppCache.php

use Symfony\Bundle\FrameworkBundle\HttpCache\HttpCache;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;
// ...

class AppCache extends HttpCache
{
    protected function invalidate(Request $request, $catch = false)
    {
        if ('PURGE' !== $request->getMethod()) {
            return parent::invalidate($request, $catch);
        }

        if ('127.0.0.1' !== $request->getClientIp()) {
            return new Response(
                'Invalid HTTP method',
                Response::HTTP_BAD_REQUEST
            );
        }

        $response = new Response();
        if ($this->getStore()->purge($request->getUri())) {
            $response->setStatusCode(200, 'Purged');
        } else {
            $response->setStatusCode(404, 'Not found');
        }

        return $response;
    }
}
```

You must protect the PURGE HTTP method somehow to avoid random people purging your cached data.

Purge instructs the cache to drop a resource in *all its variants* (according to the Vary header, see above). An alternative to purging is **refreshing** a content. Refreshing means that the caching proxy is instructed to discard its local cache and fetch the content again. This way, the new content is already available in the cache. The drawback of refreshing is that variants are not invalidated.

In many applications, the same content bit is used on various pages with different URLs. More flexible concepts exist for those cases:

- **Banning** invalidates responses matching regular expressions on the URL or other criteria;

19. http://foshttpcachebundle.readthedocs.org/

- **Cache tagging** lets you add a tag for each content used in a response so that you can invalidate all URLs containing a certain content.

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, Symfony provides a solution for these cases, based on a technology called *ESI*[20], or Edge Side Includes. Akamai 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 Symfony, `include`, as this is the only useful one outside of Akamai context:

Listing 16-18
```
<!DOCTYPE html>
<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, Symfony makes the process of including them almost effortless.

Using ESI in Symfony

First, to use ESI, be sure to enable it in your application configuration:

Listing 16-19
```
# app/config/config.yml
framework:
    # ...
    esi: { enabled: true }
```

Now, suppose you have a page that is relatively static, except for a news ticker at the bottom of the content. With ESI, you can cache the news ticker independent of the rest of the page.

Listing 16-20
```
// src/AppBundle/Controller/DefaultController.php

// ...
```

20. http://www.w3.org/TR/esi-lang

```
class DefaultController extends Controller
{
    public function aboutAction()
    {
        $response = $this->render('static/about.html.twig');
        // set the shared max age - which also marks the response as public
        $response->setSharedMaxAge(600);

        return $response;
    }
}
```

In this example, the full-page cache has a lifetime of ten minutes. Next, 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), Symfony uses the standard `render` helper to configure ESI tags:

Listing 16-21
```
{# app/Resources/views/static/about.html.twig #}

{# you can use a controller reference #}
{{ render_esi(controller('AppBundle:News:latest', { 'maxPerPage': 5 })) }}

{# ... or a URL #}
{{ render_esi(url('latest_news', { 'maxPerPage': 5 })) }}
```

By using the `esi` renderer (via the `render_esi` Twig function), you tell Symfony 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.

As you'll see below, the `maxPerPage` variable you pass is available as an argument to your controller (i.e. `$maxPerPage`). The variables passed through `render_esi` also become part of the cache key so that you have unique caches for each combination of variables and values.

When using the default `render` function (or setting the renderer to `inline`), Symfony merges the included page content into the main one before sending the response to the client. But if you use the `esi` renderer (i.e. call `render_esi`) *and* if Symfony 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, Symfony will just merge the included page content within the main one as it would have done if you had used `render`.

Symfony detects if a gateway cache supports ESI via another Akamai specification that is supported out of the box by the Symfony reverse proxy.

The embedded action can now specify its own caching rules, entirely independent of the master page.

Listing 16-22
```
// src/AppBundle/Controller/NewsController.php
namespace AppBundle\Controller;

// ...
class NewsController extends Controller
{
    public function latestAction($maxPerPage)
    {
        // ...
        $response->setSharedMaxAge(60);
```

```
11
12          return $response;
13      }
14  }
```

With ESI, the full page cache will be valid for 600 seconds, but the news component cache will only last for 60 seconds.

When using a controller reference, the ESI tag should reference the embedded action as an accessible URL so the gateway cache can fetch it independently of the rest of the page. Symfony takes care of generating a unique URL for any controller reference and it is able to route them properly thanks to the *`FragmentListener`*[21] that must be enabled in your configuration:

Listing 16-23
```yaml
# app/config/config.yml
framework:
    # ...
    fragments: { path: /_fragment }
```

One great advantage of the ESI renderer is that you can make your application as dynamic as needed and at the same time, hit the application as little as possible.

The fragment listener only responds to signed requests. Requests are only signed when using the fragment renderer and the `render_esi` Twig function.

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_esi` 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.

Summary

Symfony was designed to follow the proven rules of the road: HTTP. Caching is no exception. Mastering the Symfony cache system means becoming familiar with the HTTP cache models and using them effectively. This means that, instead of relying only on Symfony 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*

21. http://api.symfony.com/3.0/Symfony/Component/HttpKernel/EventListener/FragmentListener.html

Chapter 17
Translations

The term "internationalization" (often abbreviated *i18n*[1]) refers to the process of abstracting strings and other locale-specific pieces out of your application 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 17-1
```php
// text will *always* print out in English
dump('Hello World');
die();

// text can be translated into the end-user's language or
// default to English
dump($translator->trans('Hello World'));
die();
```

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). The *ISO 639-1*[2] *language* code, an underscore (_), then the *ISO 3166-1 alpha-2*[3] *country* code (e.g. `fr_FR` for French/France) is recommended.

In this chapter, you'll learn how to use the Translation component in the Symfony Framework. You can read the *Translation component documentation* to learn even more. Overall, the process has several steps:

1. Enable and configure Symfony's translation service;
2. Abstract strings (i.e. "messages") by wrapping them in calls to the `Translator` ("Basic Translation");
3. Create translation resources/files for each supported locale that translate each message in the application;
4. Determine, set and manage the user's locale for the request and optionally *on the user's entire session*.

1. https://en.wikipedia.org/wiki/Internationalization_and_localization
2. https://en.wikipedia.org/wiki/List_of_ISO_639-1_codes
3. https://en.wikipedia.org/wiki/ISO_3166-1#Current_codes

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:

Listing 17-2
```yaml
# app/config/config.yml
framework:
    translator: { fallbacks: [en] }
```

See Fallback Translation Locales for details on the `fallbacks` key and what Symfony does when it doesn't find a translation.

The locale used in translations is the one stored on the request. This is typically set via a `_locale` attribute on your routes (see The Locale and the URL).

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 you're translating a simple message from inside a controller:

Listing 17-3
```php
// ...
use Symfony\Component\HttpFoundation\Response;

public function indexAction()
{
    $translated = $this->get('translator')->trans('Symfony is great');

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

When this code is executed, Symfony will attempt to translate the message "Symfony is great" based on the `locale` of the user. For this to work, you need to tell Symfony how to translate the message via a "translation resource", which is usually a file that contains a collection of translations for a given locale. This "dictionary" of translations can be created in several different formats, XLIFF being the recommended format:

Listing 17-4
```xml
<!-- messages.fr.xlf -->
<?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="symfony_is_great">
                <source>Symfony is great</source>
                <target>J'aime Symfony</target>
            </trans-unit>
        </body>
    </file>
</xliff>
```

For information on where these files should be located, see Translation Resource/File Names and Locations.

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 Symfony`. You can also translate the message inside your templates.

4. http://api.symfony.com/3.0/Symfony/Component/Translation/Translator.html
5. http://api.symfony.com/3.0/Symfony/Component/Translation/Translator.html#method_trans

The Translation Process

To actually translate the message, Symfony uses a simple process:

- The `locale` of the current user, which is stored on the request is determined;
- A catalog (e.g. big collection) 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 they don't already exist. The end result is a large "dictionary" of translations.
- 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, Symfony 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 17-5
```
use Symfony\Component\HttpFoundation\Response;

public function indexAction($name)
{
    $translated = $this->get('translator')->trans('Hello '.$name);

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

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"*).

For details on how to handle this situation, see Message Placeholders in the components documentation. For how to do this in templates, see Twig Templates.

Pluralization

Another complication is when you have translations that may or may not be plural, based on some variable:

Listing 17-6
```
There is one apple.
There are 5 apples.
```

To handle this, use the *`transChoice()`*[6] method or the `transchoice` tag/filter in your template.

For much more information, see Pluralization in the Translation component documentation.

Translations in Templates

Most of the time, translation occurs in templates. Symfony provides native support for both Twig and PHP templates.

6. http://api.symfony.com/3.0/Symfony/Component/Translation/Translator.html#method_transChoice

Twig Templates

Symfony provides specialized Twig tags (`trans` and `transchoice`) to help with message translation of *static blocks of text*:

Listing 17-7
```
{% 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.

The `%var%` notation of placeholders is required when translating in Twig templates using the tag.

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 17-8
```
{% 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} %name%, there are no apples|{1} %name%, there is one apple|]1,Inf[ %name%, there are %count% apples
{% endtranschoice %}
```

The `trans` and `transchoice` filters can be used to translate *variable texts* and complex expressions:

Listing 17-9
```
{{ message|trans }}

{{ 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 translations using a filter. In other words, if you need to be sure that your translated message is *not* output escaped, you must apply the `raw` filter after the translation filter:

Listing 17-10
```
{# text translated between tags is never escaped #}
{% trans %}
    <h3>foo</h3>
{% endtrans %}

{% set message = '<h3>foo</h3>' %}

{# strings and variables translated via a filter are escaped by default #}
{{ message|trans|raw }}
{{ '<h3>bar</h3>'|trans|raw }}
```

You can set the translation domain for an entire Twig template with a single tag:

Listing 17-11
```
1  {% trans_default_domain "app" %}
```

Note that this only influences the current template, not any "included" template (in order to avoid side effects).

PHP Templates

The translator service is accessible in PHP templates through the `translator` helper:

Listing 17-12
```
1  <?php echo $view['translator']->trans('Symfony is great') ?>
2
3  <?php echo $view['translator']->transChoice(
4      '{0} There are no apples|{1} There is one apple|]1,Inf[ There are %count% apples',
5      10,
6      array('%count%' => 10)
7  ) ?>
```

Translation Resource/File Names and Locations

Symfony looks for message files (i.e. translations) in the following default locations:

- the `app/Resources/translations` directory;
- the `app/Resources/<bundle name>/translations` directory;
- the `Resources/translations/` directory inside of any bundle.

The locations are listed here with the highest priority first. That is, you can override the translation messages of a bundle in any of the top 2 directories.

The override mechanism works at a key level: only the overridden keys need to be listed in a higher priority message file. When a key is not found in a message file, the translator will automatically fall back to the lower priority message files.

The filename of the translation files is also important: each message file must be named according to the following path: `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 Symfony should load and parse the file (e.g. `xlf`, `php`, `yml`, etc).

The loader can be the name of any registered loader. By default, Symfony provides many loaders, including:

- `xlf`: 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. The recommended option is to use `xlf` for translations. For more options, see Loading Message Catalogs.

You can add other directories with the `paths` option in the configuration:

Listing 17-13
```yaml
# app/config/config.yml
framework:
    translator:
        paths:
            - '%kernel.root_dir%/../translations'
```

You can also store translations in a database, or any other storage by providing a custom class implementing the *`LoaderInterface`*[7] interface. See the translation.loader tag for more information.

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 resources:

Listing 17-14
```
$ php bin/console cache:clear
```

Fallback Translation Locales

Imagine that the user's locale is `fr_FR` and that you're translating the key `Symfony is great`. To find the French translation, Symfony actually checks translation resources for several locales:

1. First, Symfony looks for the translation in a `fr_FR` translation resource (e.g. `messages.fr_FR.xlf`);
2. If it wasn't found, Symfony looks for the translation in a `fr` translation resource (e.g. `messages.fr.xlf`);
3. If the translation still isn't found, Symfony uses the `fallbacks` configuration parameter, which defaults to `en` (see Configuration).

When Symfony doesn't find a translation in the given locale, it will add the missing translation to the log file. For details, see logging.

Handling the User's Locale

The locale of the current user is stored in the request and is accessible via the `request` object:

Listing 17-15
```php
use Symfony\Component\HttpFoundation\Request;

public function indexAction(Request $request)
{
    $locale = $request->getLocale();
}
```

To set the user's locale, you may want to create a custom event listener so that it's set before any other parts of the system (i.e. the translator) need it:

Listing 17-16
```php
public function onKernelRequest(GetResponseEvent $event)
{
    $request = $event->getRequest();

```

7. http://api.symfony.com/3.0/Symfony/Component/Translation/Loader/LoaderInterface.html

```
    // some logic to determine the $locale
    $request->setLocale($locale);
}
```

Read *Making the Locale "Sticky" during a User's Session* for more information on making the user's locale "sticky" to their session.

Setting the locale using `$request->setLocale()` in the controller is too late to affect the translator. Either set the locale via a listener (like above), the URL (see next) or call `setLocale()` directly on the `translator` service.

See the The Locale and the URL section below about setting the locale via routing.

The Locale and the URL

Since you can store the locale of the user in the session, it may be tempting to use the same URL to display a resource in 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 17-17
```
# app/config/routing.yml
contact:
    path:     /{_locale}/contact
    defaults: { _controller: AppBundle:Contact:index }
    requirements:
        _locale: en|fr|de
```

When using the special `_locale` parameter in a route, the matched locale will *automatically be set on the Request* and can be retrieved via the *`getLocale()`*[8] method. In other words, if a user visits the URI `/fr/contact`, the locale `fr` will automatically be set as the locale for the current request.

You can now use the locale to create routes to other translated pages in your application.

Read *How to Use Service Container Parameters in your Routes* to learn how to avoid hardcoding the `_locale` requirement in all your routes.

Setting a Default Locale

What if the user's locale hasn't been determined? You can guarantee that a locale is set on each user's request by defining a `default_locale` for the framework:

Listing 17-18
```
# app/config/config.yml
framework:
    default_locale: en
```

8. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Request.html#method_getLocale

Translating Constraint Messages

If you're using validation constraints with the Form component, then translating the error messages is easy: simply create a translation resource for the `validators` domain.

To start, suppose you've created a plain-old-PHP object that you need to use somewhere in your application:

Listing 17-19
```php
// src/AppBundle/Entity/Author.php
namespace AppBundle\Entity;

class Author
{
    public $name;
}
```

Add constraints through 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 17-20
```php
// src/AppBundle/Entity/Author.php
use Symfony\Component\Validator\Constraints as Assert;

class Author
{
    /**
     * @Assert\NotBlank(message = "author.name.not_blank")
     */
    public $name;
}
```

Create a translation file under the `validators` catalog for the constraint messages, typically in the `Resources/translations/` directory of the bundle.

Listing 17-21
```xml
<!-- validators.en.xlf -->
<?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="author.name.not_blank">
                <source>author.name.not_blank</source>
                <target>Please enter an author name.</target>
            </trans-unit>
        </body>
    </file>
</xliff>
```

Translating Database Content

The translation of database content should be handled by Doctrine through the *Translatable Extension*[9] or the *Translatable Behavior*[10] (PHP 5.4+). For more information, see the documentation for these libraries.

9. http://atlantic18.github.io/DoctrineExtensions/doc/translatable.html
10. https://github.com/KnpLabs/DoctrineBehaviors

Debugging Translations

When maintaining a bundle, you may use or remove the usage of a translation message without updating all message catalogues. The `debug:translation` command helps you to find these missing or unused translation messages for a given locale. It shows you a table with the result when translating the message in the given locale and the result when the fallback would be used. On top of that, it also shows you when the translation is the same as the fallback translation (this could indicate that the message was not correctly translated).

Thanks to the messages extractors, the command will detect the translation tag or filter usages in Twig templates:

Listing 17-22
```
{% trans %}Symfony2 is great{% endtrans %}

{{ 'Symfony2 is great'|trans }}

{{ 'Symfony2 is great'|transchoice(1) }}

{% transchoice 1 %}Symfony2 is great{% endtranschoice %}
```

It will also detect the following translator usages in PHP templates:

Listing 17-23
```
$view['translator']->trans("Symfony2 is great");

$view['translator']->transChoice('Symfony2 is great', 1);
```

The extractors are not able to inspect the messages translated outside templates which means that translator usages in form labels or inside your controllers won't be detected. Dynamic translations involving variables or expressions are not detected in templates, which means this example won't be analyzed:

Listing 17-24
```
{% set message = 'Symfony2 is great' %}
{{ message|trans }}
```

Suppose your application's default_locale is `fr` and you have configured `en` as the fallback locale (see Configuration and Fallback Translation Locales for how to configure these). And suppose you've already setup some translations for the `fr` locale inside an AcmeDemoBundle:

Listing 17-25
```
<!-- src/Acme/AcmeDemoBundle/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>
        </body>
    </file>
</xliff>
```

and for the `en` locale:

Listing 17-26
```
<!-- src/Acme/AcmeDemoBundle/Resources/translations/messages.en.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>Symfony2 is great</target>
            </trans-unit>
        </body>
    </file>
</xliff>
```

To inspect all messages in the `fr` locale for the AcmeDemoBundle, run:

Listing 17-27
```
$ php bin/console debug:translation fr AcmeDemoBundle
```

You will get this output:

```
+---------+-----------------+--------------------+-----------------------------+
| State(s) | Id              | Message Preview (fr) | Fallback Message Preview (en) |
+---------+-----------------+--------------------+-----------------------------+
| o       | Symfony2 is great | J'aime Symfony2    | Symfony2 is great           |
+---------+-----------------+--------------------+-----------------------------+

Legend:
 x Missing message
 o Unused message
 = Same as the fallback message
```

It indicates that the message `Symfony2 is great` is unused because it is translated, but you haven't used it anywhere yet.

Now, if you translate the message in one of your templates, you will get this output:

```
+---------+-----------------+--------------------+-----------------------------+
| State(s) | Id              | Message Preview (fr) | Fallback Message Preview (en) |
+---------+-----------------+--------------------+-----------------------------+
|         | Symfony2 is great | J'aime Symfony2    | Symfony2 is great           |
+---------+-----------------+--------------------+-----------------------------+

Legend:
 x Missing message
 o Unused message
 = Same as the fallback message
```

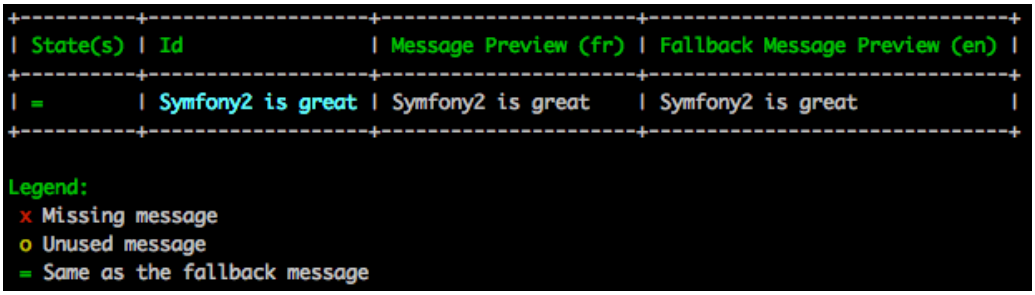
The state is empty which means the message is translated in the `fr` locale and used in one or more templates.

If you delete the message `Symfony2 is great` from your translation file for the `fr` locale and run the command, you will get:

```
+---------+-----------------+--------------------+-----------------------------+
| State(s) | Id              | Message Preview (fr) | Fallback Message Preview (en) |
+---------+-----------------+--------------------+-----------------------------+
| x =     | Symfony2 is great | Symfony2 is great  | Symfony2 is great           |
+---------+-----------------+--------------------+-----------------------------+

Legend:
 x Missing message
 o Unused message
 = Same as the fallback message
```

The state indicates the message is missing because it is not translated in the `fr` locale but it is still used in the template. Moreover, the message in the `fr` locale equals to the message in the `en` locale. This is a special case because the untranslated message id equals its translation in the `en` locale.

If you copy the content of the translation file in the `en` locale, to the translation file in the `fr` locale and run the command, you will get:



You can see that the translations of the message are identical in the `fr` and `en` locales which means this message was probably copied from French to English and maybe you forgot to translate it.

By default all domains are inspected, but it is possible to specify a single domain:

Listing 17-28
```
$ php bin/console debug:translation en AcmeDemoBundle --domain=messages
```

When bundles have a lot of messages, it is useful to display only the unused or only the missing messages, by using the `--only-unused` or `--only-missing` switches:

Listing 17-29
```
$ php bin/console debug:translation en AcmeDemoBundle --only-unused
$ php bin/console debug:translation en AcmeDemoBundle --only-missing
```

Summary

With the Symfony 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()*[11] or *transChoice()*[12] methods (learn about this in *Using the Translator*);
- Translate each message into multiple locales by creating translation message files. Symfony discovers and processes each file because its name follows a specific convention;
- Manage the user's locale, which is stored on the request, but can also be set on the user's session.

11. http://api.symfony.com/3.0/Symfony/Component/Translation/Translator.html#method_trans
12. http://api.symfony.com/3.0/Symfony/Component/Translation/Translator.html#method_transChoice

Chapter 18
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.

This chapter is about a special PHP object in Symfony 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. Since all core Symfony classes use the container, you'll learn how to extend, configure and use any object in Symfony. In large part, the service container is the biggest contributor to the speed and extensibility of Symfony.

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.

If you want to know a lot more after reading this chapter, check out the *DependencyInjection component documentation*.

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 Symfony 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 you have a simple PHP class that delivers email messages. Without a service container, you must manually create the object whenever you need it:

Listing 18-1
```php
use AppBundle\Mailer;

$mailer = new Mailer('sendmail');
$mailer->send('ryan@example.com', ...);
```

This is easy enough. The imaginary `Mailer` class allows you to configure the method used to deliver the email messages (e.g. `sendmail`, `smtp`, etc). But what if you wanted to use the mailer service somewhere else? You certainly don't want to repeat the mailer configuration *every* time you need to use the `Mailer` object. What if you needed to change the `transport` from `sendmail` to `smtp` everywhere in the application? You'd need to hunt down every place you 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, you 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 18-2
```yaml
# app/config/services.yml
services:
    app.mailer:
        class:        AppBundle\Mailer
        arguments:    [sendmail]
```

When Symfony 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`).

1. https://en.wikipedia.org/wiki/Service-oriented_architecture

An instance of the `AppBundle\Mailer` class is now available via the service container. The container is available in any traditional Symfony controller where you can access the services of the container via the `get()` shortcut method:

Listing 18-3
```php
class HelloController extends Controller
{
    // ...

    public function sendEmailAction()
    {
        // ...
        $mailer = $this->get('app.mailer');
        $mailer->send('ryan@foobar.net', ...);
    }
}
```

When you ask for the `app.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 a 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 you'll learn later how you can configure a service that has multiple instances in the "*How to Define Non Shared Services*" cookbook article.

In this example, the controller extends Symfony's base Controller, which gives you access to the service container itself. You can then use the `get` method to locate and retrieve the `app.mailer` service from the service container.

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 18-4
```yaml
# app/config/services.yml
parameters:
    app.mailer.transport:  sendmail

services:
    app.mailer:
        class:        AppBundle\Mailer
        arguments:    ['%app.mailer.transport%']
```

The end result is exactly the same as before - the difference is only in *how* you defined the service. By enclosing the `app.mailer.transport` string with percent (`%`) signs, the container knows to look for a parameter with that name. When the container is built, it looks up the value of each parameter and uses it in the service definition.

If you want to use a string that starts with an `@` sign as a parameter value (e.g. a very safe mailer password) in a YAML file, you need to escape it by adding another `@` sign (this only applies to the YAML format):

Listing 18-5
```yaml
# app/config/parameters.yml
parameters:
    # This will be parsed as string '@securepass'
    mailer_password: '@@securepass'
```

The percent sign inside a parameter or argument, as part of the string, must be escaped with another percent sign:

Listing 18-6
```xml
<argument type="string">http://symfony.com/?foo=%%s&amp;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 (this follows 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 can also contain array values. See Array Parameters.

Importing other Container Configuration Resources

In this section, service configuration files are referred to as *resources*. This is to highlight the fact that, while most configuration resources will be files (e.g. YAML, XML, PHP), Symfony 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 Symfony 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. The first method, commonly used to import other resources, is via the `imports` directive. The second method, using dependency injection extensions, is used by third-party bundles to load the configuration. Read on to learn more about both methods.

Importing Configuration with `imports`

So far, you've placed your `app.mailer` service container definition directly in the services configuration file (e.g. `app/config/services.yml`). If your application ends up having many services, this file becomes huge and hard to maintain. To avoid this, you can split your service configuration into multiple service files:

Listing 18-7
```yaml
# app/config/services/mailer.yml
parameters:
    app.mailer.transport: sendmail

services:
    app.mailer:
        class:        AppBundle\Mailer
        arguments:    ['%app.mailer.transport%']
```

The definition itself hasn't changed, only its location. To make the service container load the definitions in this resource file, use the `imports` key in any already loaded resource (e.g. `app/config/services.yml` or `app/config/config.yml`):

Listing 18-8
```yaml
# app/config/services.yml
imports:
    - { resource: services/mailer.yml }
```

The `resource` location, for files, is either a relative path from the current file or an absolute path.

Due to the way in which parameters are resolved, you cannot use them to build paths in imports dynamically. This means that something like the following doesn't work:

Listing 18-9
```yaml
# app/config/config.yml
imports:
    - { resource: '%kernel.root_dir%/parameters.yml' }
```

Importing Configuration via Container Extensions

Third-party bundle container configuration, including Symfony core services, are usually loaded using another method that's more flexible and easy to configure in your application.

Internally, each bundle defines its services like you've seen so far. However, these files aren't imported using the `import` directive. These bundles use a *dependency injection extension* to load the files. The extension also allows bundles to provide configuration to dynamically load some services.

Take the FrameworkBundle - the core Symfony Framework bundle - as an example. The presence of the following code in your application configuration invokes the service container extension inside the FrameworkBundle:

Listing 18-10
```yaml
# app/config/config.yml
framework:
    secret: xxxxxxxxxx
    form:   true
    # ...
```

When the resources are parsed, the container looks for an extension that can handle the `framework` directive. The extension in question, which lives in the FrameworkBundle, is invoked and the service configuration for the FrameworkBundle is loaded.

The settings under the `framework` directive (e.g. `form: true`) indicate that the extension should load all services related to the Form component. If form was disabled, these services wouldn't be loaded and Form integration would not be available.

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*.

> *If you want to use dependency injection extensions in your own shared bundles and provide user friendly configuration, take a look at the "How to Load Service Configuration inside a Bundle" cookbook recipe.*

Referencing (Injecting) Services

So far, the original `app.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.

As an example, suppose you 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 `app.mailer` service is already really good at delivering email messages, so you'll use it inside `NewsletterManager` to handle the actual delivery of the messages. This pretend class might look something like this:

Listing 18-11
```php
// src/AppBundle/Newsletter/NewsletterManager.php
namespace AppBundle\Newsletter;

use AppBundle\Mailer;

class NewsletterManager
{
    protected $mailer;

    public function __construct(Mailer $mailer)
    {
        $this->mailer = $mailer;
    }

    // ...
}
```

Without using the service container, you can create a new `NewsletterManager` fairly easily from inside a controller:

Listing 18-12
```php
use AppBundle\Newsletter\NewsletterManager;

// ...

public function sendNewsletterAction()
{
    $mailer = $this->get('app.mailer');
    $newsletter = new NewsletterManager($mailer);
    // ...
}
```

This approach is fine, but what if you decide later that the `NewsletterManager` class needs a second or third constructor argument? What if you decide to refactor your 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 you a much more appealing option:

Listing 18-13
```yaml
# app/config/services.yml
services:
    app.mailer:
        # ...

    app.newsletter_manager:
        class:     AppBundle\Newsletter\NewsletterManager
        arguments: ['@app.mailer']
```

In YAML, the special `@app.mailer` syntax tells the container to look for a service named `app.mailer` and to pass that object into the constructor of `NewsletterManager`. In this case, however, the specified service `app.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 `app.newsletter_manager` service needs the `app.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 classes.

Using the Expression Language

The service container also supports an "expression" that allows you to inject very specific values into a service.

For example, suppose you have a third service (not shown here), called `mailer_configuration`, which has a `getMailerMethod()` method on it, which will return a string like `sendmail` based on some configuration. Remember that the first argument to the `my_mailer` service is the simple string `sendmail`:

Listing 18-14
```yaml
# app/config/services.yml
services:
    app.mailer:
        class:        AppBundle\Mailer
        arguments:    [sendmail]
```

But instead of hardcoding this, how could we get this value from the `getMailerMethod()` of the new `mailer_configuration` service? One way is to use an expression:

Listing 18-15
```yaml
# app/config/config.yml
services:
    my_mailer:
        class:        AppBundle\Mailer
        arguments:    ["@=service('mailer_configuration').getMailerMethod()"]
```

To learn more about the expression language syntax, see *The Expression Syntax*.

In this context, you have access to 2 functions:

service
: Returns a given service (see the example above).

parameter
: Returns a specific parameter value (syntax is just like `service`).

You also have access to the *`ContainerBuilder`*[2] via a `container` variable. Here's another example:

Listing 18-16
```yaml
services:
    my_mailer:
        class:     AppBundle\Mailer
        arguments: ["@=container.hasParameter('some_param') ? parameter('some_param') : 'default_value'"]
```

Expressions can be used in `arguments`, `properties`, as arguments with `configurator` and as arguments to `calls` (method calls).

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 18-17

2. http://api.symfony.com/3.0/Symfony/Component/DependencyInjection/ContainerBuilder.html

```php
namespace AppBundle\Newsletter;

use AppBundle\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 18-18
```yaml
# app/config/services.yml
services:
    app.mailer:
        # ...

    app.newsletter_manager:
        class:     AppBundle\Newsletter\NewsletterManager
        calls:
            - [setMailer, ['@app.mailer']]
```

The approaches presented in this section are called "constructor injection" and "setter injection". The Symfony service container also supports "property injection".

Accessing the Request in a Service

Whenever you need to access the current request in a service, you can either add it as an argument to the methods that need the request or inject the `request_stack` service and access the `Request` by calling the *`getCurrentRequest()`*[3] method:

Listing 18-19
```php
namespace AppBundle\Newsletter;

use Symfony\Component\HttpFoundation\RequestStack;

class NewsletterManager
{
    protected $requestStack;

    public function __construct(RequestStack $requestStack)
    {
        $this->requestStack = $requestStack;
    }

    public function anyMethod()
    {
        $request = $this->requestStack->getCurrentRequest();
        // ... do something with the request
    }

    // ...
}
```

Now, just inject the `request_stack`, which behaves like any normal service:

3. http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/RequestStack.html#method_getCurrentRequest

Listing 18-20
```yaml
# src/AppBundle/Resources/config/services.yml
services:
    newsletter_manager:
        class:     AppBundle\Newsletter\NewsletterManager
        arguments: ["@request_stack"]
```

If you define a controller as a service then you can get the `Request` object without injecting the container by having it passed in as an argument of your action method. See The Request object as a Controller Argument for details.

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 `app.mailer` service *must* exist, otherwise an exception will be thrown. By modifying the `app.newsletter_manager` service definition, you can make this reference optional, there are two strategies for doing this.

Setting Missing Dependencies to null

You can use the `null` strategy to explicitly set the argument to `null` if the service does not exist:

Listing 18-21
```xml
<!-- app/config/services.xml -->
<?xml version="1.0" encoding="UTF-8" ?>
<container xmlns="http://symfony.com/schema/dic/services"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/dic/services
        http://symfony.com/schema/dic/services/services-1.0.xsd">

    <services>
        <service id="app.mailer">
        <!-- ... -->
        </service>

        <service id="app.newsletter_manager" class="AppBundle\Newsletter\NewsletterManager">
            <argument type="service" id="app.mailer" on-invalid="null" />
        </service>
    </services>
</container>
```

The "null" strategy is not currently supported by the YAML driver.

Ignoring Missing Dependencies

The behavior of ignoring missing dependencies is the same as the "null" behavior except when used within a method call, in which case the method call itself will be removed.

In the following example the container will inject a service using a method call if the service exists and remove the method call if it does not:

Listing 18-22
```yaml
# app/config/services.yml
services:
    app.newsletter_manager:
        class:     AppBundle\Newsletter\NewsletterManager
        arguments: ['@?app.mailer']
```

In YAML, the special `@?` syntax tells the service container that the dependency is optional. Of course, the `NewsletterManager` must also be rewritten to allow for an optional dependency:

Listing 18-23
```
public function __construct(Mailer $mailer = null)
{
    // ...
}
```

Core Symfony and Third-Party Bundle Services

Since Symfony 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, Symfony by default does not require that controllers must be defined as services. Furthermore, Symfony injects the entire service container into your controller. For example, to handle the storage of information on a user's session, Symfony provides a `session` service, which you can access inside a standard controller as follows:

Listing 18-24
```
1  public function indexAction($bar)
2  {
3      $session = $this->get('session');
4      $session->set('foo', $bar);
5
6      // ...
7  }
```

In Symfony, 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 through the request stack (`request_stack`).

You can take this a step further by using these services inside services that you've created for your application. Beginning by modifying the `NewsletterManager` to use the real Symfony `mailer` service (instead of the pretend `app.mailer`). Also pass the templating engine service to the `NewsletterManager` so that it can generate the email content via a template:

Listing 18-25
```
1  // src/AppBundle/Newsletter/NewsletterManager.php
2  namespace AppBundle\Newsletter;
3
4  use Symfony\Component\Templating\EngineInterface;
5
6  class NewsletterManager
7  {
8      protected $mailer;
9
10     protected $templating;
11
12     public function __construct(
13         \Swift_Mailer $mailer,
14         EngineInterface $templating
15     ) {
16         $this->mailer = $mailer;
17         $this->templating = $templating;
18     }
19
20     // ...
21 }
```

Configuring the service container is easy:

Listing 18-26
```
1  # app/config/services.yml
2  services:
3      app.newsletter_manager:
4          class:     AppBundle\Newsletter\NewsletterManager
5          arguments: ['@mailer', '@templating']
```

The `app.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 the `swiftmailer` entry appears in your application configuration. As was mentioned in Importing Configuration via Container Extensions, the `swiftmailer` key invokes the service extension from the SwiftmailerBundle, which registers the `mailer` service.

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 18-27
```yaml
# app/config/services.yml
services:
    foo.twig.extension:
        class: AppBundle\Extension\FooExtension
        public: false
        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 Symfony or other third-party bundles that your service should be registered or used in some special way by the bundle.

For a list of all the tags available in the core Symfony Framework, check out *The Dependency Injection Tags*. Each of these has a different effect on your service and many tags require additional arguments (beyond just the `name` parameter).

Debugging Services

You can find out what services are registered with the container using the console. To show all services and the class for each service, run:

Listing 18-28
```
$ php bin/console debug:container
```

By default, only public services are shown, but you can also view private services:

Listing 18-29
```
$ php bin/console debug:container --show-private
```

If a private service is only used as an argument to just *one* other service, it won't be displayed by the `debug:container` command, even when using the `--show-private` option. See Inline Private Services for more details.

You can get more detailed information about a particular service by specifying its id:

Listing 18-30
```
$ php bin/console debug:container app.mailer
```

Learn more

- *Introduction to Parameters*
- *Compiling the Container*
- *Working with Container Service Definitions*
- *Using a Factory to Create Services*
- *Managing Common Dependencies with Parent Services*
- *Working with Tagged Services*
- *How to Define Controllers as Services*
- *How to Work with Compiler Passes in Bundles*
- *Advanced Container Configuration*

Chapter 19
Performance

Symfony 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 of 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. As of PHP 5.5, PHP comes with *OPcache*[2] built-in. For older versions, the most widely used byte code cache is probably *APC*[3]

Using a byte code cache really has no downside, and Symfony 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. https://en.wikipedia.org/wiki/List_of_PHP_accelerators
2. http://php.net/manual/en/book.opcache.php
3. http://php.net/manual/en/book.apc.php

Use Composer's Class Map Functionality

By default, the Symfony Standard Edition uses Composer's autoloader in the *autoload.php*[4] 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 tell Composer to build a "class map" (i.e. a big array of the locations of all the classes). This can be done from the command line, and might become part of your deploy process:

Listing 19-1
```
$ composer dump-autoload --optimize
```

Internally, this builds the big class map array in `vendor/composer/autoload_classmap.php`.

Caching the Autoloader with APC

Another solution is to cache the location of each class after it's located the first time. Symfony comes with a class - *`ApcClassLoader`*[5] - that does exactly this. To use it, just adapt your front controller file. If you're using the Standard Distribution, this code should already be available as comments in this file:

Listing 19-2
```
// app.php

// ...
$loader = require __DIR__.'/../app/autoload.php';
include_once __DIR__.'/../var/bootstrap.php.cache';
// Enable APC for autoloading to improve performance.
// You should change the ApcClassLoader first argument to a unique prefix
// in order to prevent cache key conflicts with other applications
// also using APC.
/*
$apcLoader = new Symfony\Component\ClassLoader\ApcClassLoader(sha1(__FILE__), $loader);
$loader->unregister();
$apcLoader->register(true);
*/

// ...
```

For more details, see *Cache a Class Loader*.

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, Symfony 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 Symfony Standard Edition provides a script to generate a so-called *bootstrap file*[6], consisting of multiple classes definitions in a single file. By including this file (which

4. https://github.com/symfony/symfony-standard/blob/master/app/autoload.php
5. http://api.symfony.com/3.0/Symfony/Component/ClassLoader/ApcClassLoader.html
6. https://github.com/sensiolabs/SensioDistributionBundle/blob/master/Composer/ScriptHandler.php

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 Symfony 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 19-3
```
include_once __DIR__.'/../var/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 Symfony source or vendor libraries);
- when debugging, one will need to place break points inside the bootstrap file.

If you're using the Symfony Standard Edition, the bootstrap file is automatically rebuilt after updating the vendor libraries via the `composer install` command.

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 fewer 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.