Symfony

The Cookbook

for Symfony 2.0

generated on May 16, 2012

Contents at a Glance

Chapter 1

How to Create and store a Symfony2 Project in git

Though this entry is specifically about git, the same generic principles will apply if you're storing your project in Subversion.

Once you've read through *Creating Pages in Symfony2* and become familiar with using Symfony, you'll no-doubt be ready to start your own project. In this cookbook article, you'll learn the best way to start a new Symfony2 project that's stored using the *git*[1] source control management system.

Initial Project Setup

To get started, you'll need to download Symfony and initialize your local git repository:

1. Download the *Symfony2 Standard Edition*[2] without vendors.
2. Unzip/untar the distribution. It will create a folder called Symfony with your new project structure, config files, etc. Rename it to whatever you like.
3. Create a new file called `.gitignore` at the root of your new project (e.g. next to the `deps` file) and paste the following into it. Files matching these patterns will be ignored by git:

Listing 1-1

```
/web/bundles/
/app/bootstrap*
/app/cache/*
/app/logs/*
/vendor/
/app/config/parameters.ini
```

1. http://git-scm.com/
2. http://symfony.com/download

You may also want to create a .gitignore file that can be used system-wide, in which case, you can find more information here: *Github .gitignore*[3] This way you can exclude files/folders often used by your IDE for all of your projects.

4. Copy `app/config/parameters.ini` to `app/config/parameters.ini.dist`. The `parameters.ini` file is ignored by git (see above) so that machine-specific settings like database passwords aren't committed. By creating the `parameters.ini.dist` file, new developers can quickly clone the project, copy this file to `parameters.ini`, customize it, and start developing.
5. Initialize your git repository:

Listing 1-2
```
$ git init
```

6. Add all of the initial files to git:

Listing 1-3
```
$ git add .
```

7. Create an initial commit with your started project:

Listing 1-4
```
$ git commit -m "Initial commit"
```

8. Finally, download all of the third-party vendor libraries:

Listing 1-5
```
$ php bin/vendors install
```

At this point, you have a fully-functional Symfony2 project that's correctly committed to git. You can immediately begin development, committing the new changes to your git repository.

After execution of the command:

Listing 1-6
```
$ php bin/vendors install
```

your project will contain complete the git history of all the bundles and libraries defined in the `deps` file. It can be as much as 100 MB! If you save the current versions of all your dependencies with the command:

Listing 1-7
```
$ php bin/vendors lock
```

then you can remove the git history directories with the following command:

Listing 1-8
```
$ find vendor -name .git -type d | xargs rm -rf
```

The command removes all `.git` directories contained inside the `vendor` directory.

If you want to update bundles defined in `deps` file after this, you will have to reinstall them:

Listing 1-9
```
$ php bin/vendors install --reinstall
```

You can continue to follow along with the *Creating Pages in Symfony2* chapter to learn more about how to configure and develop inside your application.

3. `http://help.github.com/ignore-files/`

The Symfony2 Standard Edition comes with some example functionality. To remove the sample code, follow the instructions on the *Standard Edition Readme*[4].

Managing Vendor Libraries with bin/vendors and deps

How does it work ?

Every Symfony project uses a group of third-party "vendor" libraries. One way or another the goal is to download these files into your `vendor/` directory and, ideally, to give you some sane way to manage the exact version you need for each.

By default, these libraries are downloaded by running a `php bin/vendors install` "downloader" script. This script reads from the `deps` file at the root of your project. This is an ini-formatted script, which holds a list of each of the external libraries you need, the directory each should be downloaded to, and (optionally) the version to be downloaded. The `bin/vendors` script uses `git` to downloaded these, solely because these external libraries themselves tend to be stored via git. The `bin/vendors` script also reads the `deps.lock` file, which allows you to pin each library to an exact git commit hash.

It's important to realize that these vendor libraries are *not* actually part of *your* repository. Instead, they're simply un-tracked files that are downloaded into the `vendor/` directory by the `bin/vendors` script. But since all the information needed to download these files is saved in `deps` and `deps.lock` (which *are* stored) in our repository), any other developer can use our project, run `php bin/vendors install`, and download the exact same set of vendor libraries. This means that you're controlling exactly what each vendor library looks like, without needing to actually commit them to *your* repository.

So, whenever a developer uses your project, he/she should run the `php bin/vendors install` script to ensure that all of the needed vendor libraries are downloaded.

Upgrading Symfony

Since Symfony is just a group of third-party libraries and third-party libraries are entirely controlled through `deps` and `deps.lock`, upgrading Symfony means simply upgrading each of these files to match their state in the latest Symfony Standard Edition.

Of course, if you've added new entries to `deps` or `deps.lock`, be sure to replace only the original parts (i.e. be sure not to also delete any of your custom entries).

There is also a `php bin/vendors update` command, but this has nothing to do with upgrading your project and you will normally not need to use it. This command is used to freeze the versions of all of your vendor libraries by updating them to the version specified in `deps` and recording it into the `deps.lock` file.

Hacking vendor libraries

Sometimes, you want a specific branch, tag, or commit of a library to be downloaded or upgraded. You can set that directly to the `deps` file :

Listing 1-10

```
[AcmeAwesomeBundle]
    git=http://github.com/johndoe/Acme/AwesomeBundle.git
    target=/bundles/Acme/AwesomeBundle
    version=the-awesome-version
```

4. https://github.com/symfony/symfony-standard/blob/master/README.md

- The `git` option sets the URL of the library. It can use various protocols, like `http://` as well as `git://`.
- The `target` option specifies where the repository will live : plain Symfony bundles should go under the `vendor/bundles/Acme` directory, other third-party libraries usually go to `vendor/my-awesome-library-name`. The target directory defaults to this last option when not specified.
- **The `version` option allows you to set a specific revision. You can use a tag**

  (`version=origin/0.42`) or a branch name (`refs/remotes/origin/awesome-branch`). It defaults to `origin/HEAD`.

Updating workflow

When you execute the `php bin/vendors install`, for every library, the script first checks if the install directory exists.

If it does not (and ONLY if it does not), it runs a `git clone`.

Then, it does a `git fetch origin` and a `git reset --hard the-awesome-version`.

This means that the repository will only be cloned once. If you want to perform any change of the git remote, you MUST delete the entire target directory, not only its content.

Vendors and Submodules

Instead of using the `deps`, `bin/vendors` system for managing your vendor libraries, you may instead choose to use native *git submodules*[5]. There is nothing wrong with this approach, though the `deps` system is the official way to solve this problem and git submodules can be difficult to work with at times.

Storing your Project on a Remote Server

You now have a fully-functional Symfony2 project stored in git. However, in most cases, you'll also want to store your project on a remote server both for backup purposes, and so that other developers can collaborate on the project.

The easiest way to store your project on a remote server is via *GitHub*[6]. Public repositories are free, however you will need to pay a monthly fee to host private repositories.

Alternatively, you can store your git repository on any server by creating a *barebones repository*[7] and then pushing to it. One library that helps manage this is *Gitolite*[8].

5. `http://book.git-scm.com/5_submodules.html`
6. `https://github.com/`
7. `http://progit.org/book/ch4-4.html`
8. `https://github.com/sitaramc/gitolite`

Chapter 2

How to Create and store a Symfony2 Project in Subversion

This entry is specifically about Subversion, and based on principles found in *How to Create and store a Symfony2 Project in git*.

Once you've read through *Creating Pages in Symfony2* and become familiar with using Symfony, you'll no-doubt be ready to start your own project. The preferred method to manage Symfony2 projects is using *git*[1] but some prefer to use *Subversion*[2] which is totally fine!. In this cookbook article, you'll learn how to manage your project using *svn*[3] in a similar manner you would do with *git*[4].

This is **a** method to tracking your Symfony2 project in a Subversion repository. There are several ways to do and this one is simply one that works.

The Subversion Repository

For this article we will suppose that your repository layout follows the widespread standard structure:

Listing 2-1

```
myproject/
    branches/
    tags/
    trunk/
```

1. http://git-scm.com/
2. http://subversion.apache.org/
3. http://subversion.apache.org/
4. http://git-scm.com/

Most subversion hosting should follow this standard practice. This is the recommended layout in *Version Control with Subversion*[5] and the layout used by most free hosting (see *Subversion hosting solutions*).

Initial Project Setup

To get started, you'll need to download Symfony2 and get the basic Subversion setup:

1. Download the *Symfony2 Standard Edition*[6] with or without vendors.
2. Unzip/untar the distribution. It will create a folder called Symfony with your new project structure, config files, etc. Rename it to whatever you like.
3. Checkout the Subversion repository that will host this project. Let's say it is hosted on *Google code*[7] and called `myproject`:

Listing 2-2

```
$ svn checkout http://myproject.googlecode.com/svn/trunk myproject
```

4. Copy the Symfony2 project files in the subversion folder:

Listing 2-3

```
$ mv Symfony/* myproject/
```

5. Let's now set the ignore rules. Not everything *should* be stored in your subversion repository. Some files (like the cache) are generated and others (like the database configuration) are meant to be customized on each machine. This makes use of the `svn:ignore` property, so that we can ignore specific files.

Listing 2-4

```
$ cd myproject/
$ svn add --depth=empty app app/cache app/logs app/config web

$ svn propset svn:ignore "vendor" .
$ svn propset svn:ignore "bootstrap*" app/
$ svn propset svn:ignore "parameters.ini" app/config/
$ svn propset svn:ignore "*" app/cache/
$ svn propset svn:ignore "*" app/logs/

$ svn propset svn:ignore "bundles" web

$ svn ci -m "commit basic symfony ignore list (vendor, app/bootstrap*, app/config/
parameters.ini, app/cache/*, app/logs/*, web/bundles)"
```

6. The rest of the files can now be added and committed to the project:

Listing 2-5

```
$ svn add --force .
$ svn ci -m "add basic Symfony Standard 2.X.Y"
```

7. Copy `app/config/parameters.ini` to `app/config/parameters.ini.dist`. The `parameters.ini` file is ignored by svn (see above) so that machine-specific settings like database passwords aren't committed. By creating the `parameters.ini.dist` file, new developers can quickly clone the project, copy this file to `parameters.ini`, customize it, and start developing.
8. Finally, download all of the third-party vendor libraries:

5. `http://svnbook.red-bean.com/`
6. `http://symfony.com/download`
7. `http://code.google.com/hosting/`

```
$ php bin/vendors install
```
Listing 2-6

git[8] has to be installed to run `bin/vendors`, this is the protocol used to fetch vendor libraries. This only means that `git` is used as a tool to basically help download the libraries in the `vendor/` directory.

At this point, you have a fully-functional Symfony2 project stored in your Subversion repository. The development can start with commits in the Subversion repository.

You can continue to follow along with the *Creating Pages in Symfony2* chapter to learn more about how to configure and develop inside your application.

The Symfony2 Standard Edition comes with some example functionality. To remove the sample code, follow the instructions on the *Standard Edition Readme*[9].

Managing Vendor Libraries with bin/vendors and deps

How does it work ?

Every Symfony project uses a group of third-party "vendor" libraries. One way or another the goal is to download these files into your `vendor/` directory and, ideally, to give you some sane way to manage the exact version you need for each.

By default, these libraries are downloaded by running a `php bin/vendors install` "downloader" script. This script reads from the `deps` file at the root of your project. This is an ini-formatted script, which holds a list of each of the external libraries you need, the directory each should be downloaded to, and (optionally) the version to be downloaded. The `bin/vendors` script uses `git` to downloaded these, solely because these external libraries themselves tend to be stored via git. The `bin/vendors` script also reads the `deps.lock` file, which allows you to pin each library to an exact git commit hash.

It's important to realize that these vendor libraries are *not* actually part of *your* repository. Instead, they're simply un-tracked files that are downloaded into the `vendor/` directory by the `bin/vendors` script. But since all the information needed to download these files is saved in `deps` and `deps.lock` (which *are* stored) in our repository), any other developer can use our project, run `php bin/vendors install`, and download the exact same set of vendor libraries. This means that you're controlling exactly what each vendor library looks like, without needing to actually commit them to *your* repository.

So, whenever a developer uses your project, he/she should run the `php bin/vendors install` script to ensure that all of the needed vendor libraries are downloaded.

Upgrading Symfony

Since Symfony is just a group of third-party libraries and third-party libraries are entirely controlled through `deps` and `deps.lock`, upgrading Symfony means simply upgrading each of these files to match their state in the latest Symfony Standard Edition.

Of course, if you've added new entries to `deps` or `deps.lock`, be sure to replace only the original parts (i.e. be sure not to also delete any of your custom entries).

8. http://git-scm.com/
9. https://github.com/symfony/symfony-standard/blob/master/README.md

There is also a `php bin/vendors update` command, but this has nothing to do with upgrading your project and you will normally not need to use it. This command is used to freeze the versions of all of your vendor libraries by updating them to the version specified in `deps` and recording it into the `deps.lock` file.

Hacking vendor libraries

Sometimes, you want a specific branch, tag, or commit of a library to be downloaded or upgraded. You can set that directly to the `deps` file :

Listing 2-7

```
[AcmeAwesomeBundle]
    git=http://github.com/johndoe/Acme/AwesomeBundle.git
    target=/bundles/Acme/AwesomeBundle
    version=the-awesome-version
```

- The `git` option sets the URL of the library. It can use various protocols, like `http://` as well as `git://`.
- The `target` option specifies where the repository will live : plain Symfony bundles should go under the `vendor/bundles/Acme` directory, other third-party libraries usually go to `vendor/my-awesome-library-name`. The target directory defaults to this last option when not specified.
- **The `version` option allows you to set a specific revision. You can use a tag**

  (`version=origin/0.42`) or a branch name (`refs/remotes/origin/awesome-branch`). It defaults to `origin/HEAD`.

Updating workflow

When you execute the `php bin/vendors install`, for every library, the script first checks if the install directory exists.

If it does not (and ONLY if it does not), it runs a `git clone`.

Then, it does a `git fetch origin` and a `git reset --hard the-awesome-version`.

This means that the repository will only be cloned once. If you want to perform any change of the git remote, you MUST delete the entire target directory, not only its content.

Subversion hosting solutions

The biggest difference between *git*[10] and *svn*[11] is that Subversion *needs* a central repository to work. You then have several solutions:

- Self hosting: create your own repository and access it either through the filesystem or the network. To help in this task you can read Version Control with Subversion.
- Third party hosting: there are a lot of serious free hosting solutions available like *GitHub*[12], *Google code*[13], *SourceForge*[14] or *Gna*[15]. Some of them offer git hosting as well.

10. http://git-scm.com/
11. http://subversion.apache.org/
12. http://github.com/
13. http://code.google.com/hosting/
14. http://sourceforge.net/
15. http://gna.org/

Chapter 3

How to customize Error Pages

When any exception is thrown in Symfony2, the exception is caught inside the `Kernel` class and eventually forwarded to a special controller, `TwigBundle:Exception:show` for handling. This controller, which lives inside the core `TwigBundle`, determines which error template to display and the status code that should be set for the given exception.

Error pages can be customized in two different ways, depending on how much control you need:

1. Customize the error templates of the different error pages (explained below);
2. Replace the default exception controller `TwigBundle::Exception:show` with your own controller and handle it however you want (see *exception_controller in the Twig reference*);

The customization of exception handling is actually much more powerful than what's written here. An internal event, `kernel.exception`, is thrown which allows complete control over exception handling. For more information, see *kernel.exception Event*.

All of the error templates live inside `TwigBundle`. To override the templates, we simply rely on the standard method for overriding templates that live inside a bundle. For more information, see *Overriding Bundle Templates*.

For example, to override the default error template that's shown to the end-user, create a new template located at `app/Resources/TwigBundle/views/Exception/error.html.twig`:

Listing 3-1

```
<!DOCTYPE html>
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <title>An Error Occurred: {{ status_text }}</title>
</head>
<body>
    <h1>Oops! An Error Occurred</h1>
    <h2>The server returned a "{{ status_code }} {{ status_text }}".</h2>
</body>
</html>
```

If you're not familiar with Twig, don't worry. Twig is a simple, powerful and optional templating engine that integrates with `Symfony2`. For more information about Twig see *Creating and using Templates*.

In addition to the standard HTML error page, Symfony provides a default error page for many of the most common response formats, including JSON (`error.json.twig`), XML, (`error.xml.twig`), and even Javascript (`error.js.twig`), to name a few. To override any of these templates, just create a new file with the same name in the `app/Resources/TwigBundle/views/Exception` directory. This is the standard way of overriding any template that lives inside a bundle.

Customizing the 404 Page and other Error Pages

You can also customize specific error templates according to the HTTP status code. For instance, create a `app/Resources/TwigBundle/views/Exception/error404.html.twig` template to display a special page for 404 (page not found) errors.

Symfony uses the following algorithm to determine which template to use:

- First, it looks for a template for the given format and status code (like `error404.json.twig`);
- If it does not exist, it looks for a template for the given format (like `error.json.twig`);
- If it does not exist, it falls back to the HTML template (like `error.html.twig`).

To see the full list of default error templates, see the `Resources/views/Exception` directory of the `TwigBundle`. In a standard Symfony2 installation, the `TwigBundle` can be found at `vendor/symfony/src/Symfony/Bundle/TwigBundle`. Often, the easiest way to customize an error page is to copy it from the `TwigBundle` into `app/Resources/TwigBundle/views/Exception` and then modify it.

The debug-friendly exception pages shown to the developer can even be customized in the same way by creating templates such as `exception.html.twig` for the standard HTML exception page or `exception.json.twig` for the JSON exception page.

Chapter 4

How to define Controllers as Services

In the book, you've learned how easily a controller can be used when it extends the base *`Controller`*[1] class. While this works fine, controllers can also be specified as services.

To refer to a controller that's defined as a service, use the single colon (:) notation. For example, suppose we've defined a service called `my_controller` and we want to forward to a method called `indexAction()` inside the service:

```
$this->forward('my_controller:indexAction', array('foo' => $bar));
```

Listing 4-1

You need to use the same notation when defining the route `_controller` value:

```
my_controller:
    pattern:   /
    defaults:  { _controller: my_controller:indexAction }
```

Listing 4-2

To use a controller in this way, it must be defined in the service container configuration. For more information, see the *Service Container* chapter.

When using a controller defined as a service, it will most likely not extend the base `Controller` class. Instead of relying on its shortcut methods, you'll interact directly with the services that you need. Fortunately, this is usually pretty easy and the base `Controller` class itself is a great source on how to perform many common tasks.

Specifying a controller as a service takes a little bit more work. The primary advantage is that the entire controller or any services passed to the controller can be modified via the service container configuration. This is especially useful when developing an open-source bundle or any bundle that will be used in many different projects. So, even if you don't specify your controllers as services, you'll likely see this done in some open-source Symfony2 bundles.

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

Chapter 5

How to force routes to always use HTTPS or HTTP

Sometimes, you want to secure some routes and be sure that they are always accessed via the HTTPS protocol. The Routing component allows you to enforce the URI scheme via the `_scheme` requirement:

Listing 5-1

```
secure:
    pattern:  /secure
    defaults: { _controller: AcmeDemoBundle:Main:secure }
    requirements:
        _scheme:  https
```

The above configuration forces the `secure` route to always use HTTPS.

When generating the `secure` URL, and if the current scheme is HTTP, Symfony will automatically generate an absolute URL with HTTPS as the scheme:

Listing 5-2

```
# If the current scheme is HTTPS
{{ path('secure') }}
# generates /secure

# If the current scheme is HTTP
{{ path('secure') }}
# generates https://example.com/secure
```

The requirement is also enforced for incoming requests. If you try to access the `/secure` path with HTTP, you will automatically be redirected to the same URL, but with the HTTPS scheme.

The above example uses `https` for the `_scheme`, but you can also force a URL to always use `http`.

The Security component provides another way to enforce HTTP or HTTPs via the `requires_channel` setting. This alternative method is better suited to secure an "area" of your website (all URLs under `/admin`) or when you want to secure URLs defined in a third party bundle.

Chapter 6

How to allow a "/" character in a route parameter

Sometimes, you need to compose URLs with parameters that can contain a slash `/`. For example, take the classic `/hello/{name}` route. By default, `/hello/Fabien` will match this route but not `/hello/Fabien/Kris`. This is because Symfony uses this character as separator between route parts.

This guide covers how you can modify a route so that `/hello/Fabien/Kris` matches the `/hello/{name}` route, where `{name}` equals `Fabien/Kris`.

Configure the Route

By default, the symfony routing components requires that the parameters match the following regex pattern: `[^/]+`. This means that all characters are allowed except `/`.

You must explicitly allow `/` to be part of your parameter by specifying a more permissive regex pattern.

Listing 6-1

```
_hello:
    pattern: /hello/{name}
    defaults: { _controller: AcmeDemoBundle:Demo:hello }
    requirements:
        name: ".+"
```

That's it! Now, the `{name}` parameter can contain the `/` character.

Chapter 7

How to Use Assetic for Asset Management

Assetic combines two major ideas: assets and filters. The assets are files such as CSS, JavaScript and image files. The filters are things that can be applied to these files before they are served to the browser. This allows a separation between the asset files stored in the application and the files actually presented to the user.

Without Assetic, you just serve the files that are stored in the application directly:

Listing 7-1

```
<script src="{{ asset('js/script.js') }}" type="text/javascript" />
```

But *with* Assetic, you can manipulate these assets however you want (or load them from anywhere) before serving them. These means you can:

- Minify and combine all of your CSS and JS files
- Run all (or just some) of your CSS or JS files through some sort of compiler, such as LESS, SASS or CoffeeScript
- Run image optimizations on your images

Assets

Using Assetic provides many advantages over directly serving the files. The files do not need to be stored where they are served from and can be drawn from various sources such as from within a bundle:

Listing 7-2

```
{% javascripts
    '@AcmeFooBundle/Resources/public/js/*'
%}
<script type="text/javascript" src="{{ asset_url }}"></script>
{% endjavascripts %}
```

To bring in CSS stylesheets, you can use the same methodologies seen in this entry, except with the *stylesheets* tag:

Listing 7-3

```
{% stylesheets
    '@AcmeFooBundle/Resources/public/css/*'
%}
<link rel="stylesheet" href="{{ asset_url }}" />
{% endstylesheets %}
```

In this example, all of the files in the `Resources/public/js/` directory of the `AcmeFooBundle` will be loaded and served from a different location. The actual rendered tag might simply look like:

Listing 7-4

```
<script src="/app_dev.php/js/abcd123.js"></script>
```

This is a key point: once you let Assetic handle your assets, the files are served from a different location. This *can* cause problems with CSS files that reference images by their relative path. However, this can be fixed by using the `cssrewrite` filter, which updates paths in CSS files to reflect their new location.

Combining Assets

You can also combine several files into one. This helps to reduce the number of HTTP requests, which is great for front end performance. It also allows you to maintain the files more easily by splitting them into manageable parts. This can help with re-usability as you can easily split project-specific files from those which can be used in other applications, but still serve them as a single file:

Listing 7-5

```
{% javascripts
    '@AcmeFooBundle/Resources/public/js/*'
    '@AcmeBarBundle/Resources/public/js/form.js'
    '@AcmeBarBundle/Resources/public/js/calendar.js'
%}
<script src="{{ asset_url }}"></script>
{% endjavascripts %}
```

In the *dev* environment, each file is still served individually, so that you can debug problems more easily. However, in the *prod* environment, this will be rendered as a single *script* tag.

If you're new to Assetic and try to use your application in the `prod` environment (by using the `app.php` controller), you'll likely see that all of your CSS and JS breaks. Don't worry! This is on purpose. For details on using Assetic in the *prod* environment, see *Dumping Asset Files*.

And combining files doesn't only apply to *your* files. You can also use Assetic to combine third party assets, such as jQuery, with your own into a single file:

Listing 7-6

```
{% javascripts
    '@AcmeFooBundle/Resources/public/js/thirdparty/jquery.js'
    '@AcmeFooBundle/Resources/public/js/*'
%}
<script src="{{ asset_url }}"></script>
{% endjavascripts %}
```

Filters

Once they're managed by Assetic, you can apply filters to your assets before they are served. This includes filters that compress the output of your assets for smaller file sizes (and better front-end optimization). Other filters can compile JavaScript file from CoffeeScript files and process SASS into CSS. In fact, Assetic has a long list of available filters.

Many of the filters do not do the work directly, but use existing third-party libraries to do the heavy-lifting. This means that you'll often need to install a third-party library to use a filter. The great advantage of using Assetic to invoke these libraries (as opposed to using them directly) is that instead of having to run them manually after you work on the files, Assetic will take care of this for you and remove this step altogether from your development and deployment processes.

To use a filter, you first need to specify it in the Assetic configuration. Adding a filter here doesn't mean it's being used - it just means that it's available to use (we'll use the filter below).

For example to use the JavaScript YUI Compressor the following config should be added:

Listing 7-7

```
# app/config/config.yml
assetic:
    filters:
        yui_js:
            jar: "%kernel.root_dir%/Resources/java/yuicompressor.jar"
```

Now, to actually *use* the filter on a group of JavaScript files, add it into your template:

Listing 7-8

```
{% javascripts
    '@AcmeFooBundle/Resources/public/js/*'
    filter='yui_js'
%}
<script src="{{ asset_url }}"></script>
{% endjavascripts %}
```

A more detailed guide about configuring and using Assetic filters as well as details of Assetic's debug mode can be found in *How to Minify JavaScripts and Stylesheets with YUI Compressor*.

Controlling the URL used

If you wish to, you can control the URLs that Assetic produces. This is done from the template and is relative to the public document root:

Listing 7-9

```
{% javascripts
    '@AcmeFooBundle/Resources/public/js/*'
    output='js/compiled/main.js'
%}
<script src="{{ asset_url }}"></script>
{% endjavascripts %}
```

Symfony also contains a method for cache *busting*, where the final URL generated by Assetic contains a query parameter that can be incremented via configuration on each deployment. For more information, see the *assets_version* configuration option.

Dumping Asset Files

In the `dev` environment, Assetic generates paths to CSS and JavaScript files that don't physically exist on your computer. But they render nonetheless because an internal Symfony controller opens the files and serves back the content (after running any filters).

This kind of dynamic serving of processed assets is great because it means that you can immediately see the new state of any asset files you change. It's also bad, because it can be quite slow. If you're using a lot of filters, it might be downright frustrating.

Fortunately, Assetic provides a way to dump your assets to real files, instead of being generated dynamically.

Dumping Asset Files in the `prod` environment

In the `prod` environment, your JS and CSS files are represented by a single tag each. In other words, instead of seeing each JavaScript file you're including in your source, you'll likely just see something like this:

Listing 7-10

```
<script src="/app_dev.php/js/abcd123.js"></script>
```

Moreover, that file does **not** actually exist, nor is it dynamically rendered by Symfony (as the asset files are in the `dev` environment). This is on purpose - letting Symfony generate these files dynamically in a production environment is just too slow.

Instead, each time you use your app in the `prod` environment (and therefore, each time you deploy), you should run the following task:

Listing 7-11

```
php app/console assetic:dump --env=prod --no-debug
```

This will physically generate and write each file that you need (e.g. `/js/abcd123.js`). If you update any of your assets, you'll need to run this again to regenerate the file.

Dumping Asset Files in the `dev` environment

By default, each asset path generated in the `dev` environment is handled dynamically by Symfony. This has no disadvantage (you can see your changes immediately), except that assets can load noticeably slow. If you feel like your assets are loading too slowly, follow this guide.

First, tell Symfony to stop trying to process these files dynamically. Make the following change in your `config_dev.yml` file:

Listing 7-12

```
# app/config/config_dev.yml
assetic:
    use_controller: false
```

Next, since Symfony is no longer generating these assets for you, you'll need to dump them manually. To do so, run the following:

Listing 7-13

```
php app/console assetic:dump
```

This physically writes all of the asset files you need for your `dev` environment. The big disadvantage is that you need to run this each time you update an asset. Fortunately, by passing the `--watch` option, the command will automatically regenerate assets *as they change*:

Listing 7-14

```
php app/console assetic:dump --watch
```

Since running this command in the `dev` environment may generate a bunch of files, it's usually a good idea to point your generated assets files to some isolated directory (e.g. `/js/compiled`), to keep things organized:

Listing 7-15

```
{% javascripts
    '@AcmeFooBundle/Resources/public/js/*'
    output='js/compiled/main.js'
%}
<script src="{{ asset_url }}"></script>
{% endjavascripts %}
```

Chapter 8

How to Minify JavaScripts and Stylesheets with YUI Compressor

Yahoo! provides an excellent utility for minifying JavaScripts and stylesheets so they travel over the wire faster, the *YUI Compressor*[1]. Thanks to Assetic, you can take advantage of this tool very easily.

Download the YUI Compressor JAR

The YUI Compressor is written in Java and distributed as a JAR. *Download the JAR*[2] from the Yahoo! site and save it to `app/Resources/java/yuicompressor.jar`.

Configure the YUI Filters

Now you need to configure two Assetic filters in your application, one for minifying JavaScripts with the YUI Compressor and one for minifying stylesheets:

Listing 8-1

```
# app/config/config.yml
assetic:
    filters:
        yui_css:
            jar: "%kernel.root_dir%/Resources/java/yuicompressor.jar"
        yui_js:
            jar: "%kernel.root_dir%/Resources/java/yuicompressor.jar"
```

You now have access to two new Assetic filters in your application: `yui_css` and `yui_js`. These will use the YUI Compressor to minify stylesheets and JavaScripts, respectively.

1. http://developer.yahoo.com/yui/compressor/
2. http://yuilibrary.com/downloads/#yuicompressor

Minify your Assets

You have YUI Compressor configured now, but nothing is going to happen until you apply one of these filters to an asset. Since your assets are a part of the view layer, this work is done in your templates:

Listing 8-2
```
{% javascripts '@AcmeFooBundle/Resources/public/js/*' filter='yui_js' %}
<script src="{{ asset_url }}"></script>
{% endjavascripts %}
```

The above example assumes that you have a bundle called `AcmeFooBundle` and your JavaScript files are in the `Resources/public/js` directory under your bundle. This isn't important however - you can include your Javascript files no matter where they are.

With the addition of the `yui_js` filter to the asset tags above, you should now see minified JavaScripts coming over the wire much faster. The same process can be repeated to minify your stylesheets.

Listing 8-3
```
{% stylesheets '@AcmeFooBundle/Resources/public/css/*' filter='yui_css' %}
<link rel="stylesheet" type="text/css" media="screen" href="{{ asset_url }}" />
{% endstylesheets %}
```

Disable Minification in Debug Mode

Minified JavaScripts and Stylesheets are very difficult to read, let alone debug. Because of this, Assetic lets you disable a certain filter when your application is in debug mode. You can do this be prefixing the filter name in your template with a question mark: `?`. This tells Assetic to only apply this filter when debug mode is off.

Listing 8-4
```
{% javascripts '@AcmeFooBundle/Resources/public/js/*' filter='?yui_js' %}
<script src="{{ asset_url }}"></script>
{% endjavascripts %}
```

Instead of adding the filter to the asset tags, you can also globally enable it by adding the apply-to attribute to the filter configuration, for example in the yui_js filter `apply_to: "\.js$"`. To only have the filter applied in production, add this to the config_prod file rather than the common config file. For details on applying filters by file extension, see *Filtering based on a File Extension*.

Chapter 9

How to Use Assetic For Image Optimization with Twig Functions

Amongst its many filters, Assetic has four filters which can be used for on-the-fly image optimization. This allows you to get the benefits of smaller file sizes without having to use an image editor to process each image. The results are cached and can be dumped for production so there is no performance hit for your end users.

Using Jpegoptim

Jpegoptim[1] is a utility for optimizing JPEG files. To use it with Assetic, add the following to the Assetic config:

Listing 9-1

```
# app/config/config.yml
assetic:
    filters:
        jpegoptim:
            bin: path/to/jpegoptim
```

Notice that to use jpegoptim, you must have it already installed on your system. The `bin` option points to the location of the compiled binary.

It can now be used from a template:

Listing 9-2

```
{% image '@AcmeFooBundle/Resources/public/images/example.jpg'
    filter='jpegoptim' output='/images/example.jpg'
%}
<img src="{{ asset_url }}" alt="Example"/>
{% endimage %}
```

1. http://www.kokkonen.net/tjko/projects.html

Removing all EXIF Data

By default, running this filter only removes some of the meta information stored in the file. Any EXIF data and comments are not removed, but you can remove these by using the `strip_all` option:

Listing 9-3
```
# app/config/config.yml
assetic:
    filters:
        jpegoptim:
            bin: path/to/jpegoptim
            strip_all: true
```

Lowering Maximum Quality

The quality level of the JPEG is not affected by default. You can gain further file size reductions by setting the max quality setting lower than the current level of the images. This will of course be at the expense of image quality:

Listing 9-4
```
# app/config/config.yml
assetic:
    filters:
        jpegoptim:
            bin: path/to/jpegoptim
            max: 70
```

Shorter syntax: Twig Function

If you're using Twig, it's possible to achieve all of this with a shorter syntax by enabling and using a special Twig function. Start by adding the following config:

Listing 9-5
```
# app/config/config.yml
assetic:
    filters:
        jpegoptim:
            bin: path/to/jpegoptim
    twig:
        functions:
            jpegoptim: ~
```

The Twig template can now be changed to the following:

Listing 9-6
```
<img src="{{ jpegoptim('@AcmeFooBundle/Resources/public/images/example.jpg') }}"
     alt="Example"/>
```

You can specify the output directory in the config in the following way:

Listing 9-7
```
# app/config/config.yml
assetic:
    filters:
        jpegoptim:
            bin: path/to/jpegoptim
    twig:
        functions:
            jpegoptim: { output: images/*.jpg }
```

Chapter 10

How to Apply an Assetic Filter to a Specific File Extension

Assetic filters can be applied to individual files, groups of files or even, as you'll see here, files that have a specific extension. To show you how to handle each option, let's suppose that you want to use Assetic's CoffeeScript filter, which compiles CoffeeScript files into Javascript.

The main configuration is just the paths to coffee and node. These default respectively to `/usr/bin/coffee` and `/usr/bin/node`:

Listing 10-1

```
# app/config/config.yml
assetic:
    filters:
        coffee:
            bin: /usr/bin/coffee
            node: /usr/bin/node
```

Filter a Single File

You can now serve up a single CoffeeScript file as JavaScript from within your templates:

Listing 10-2

```
{% javascripts '@AcmeFooBundle/Resources/public/js/example.coffee'
    filter='coffee'
%}
<script src="{{ asset_url }}" type="text/javascript"></script>
{% endjavascripts %}
```

This is all that's needed to compile this CoffeeScript file and server it as the compiled JavaScript.

Filter Multiple Files

You can also combine multiple CoffeeScript files into a single output file:

Listing 10-3

```
{% javascripts '@AcmeFooBundle/Resources/public/js/example.coffee'
               '@AcmeFooBundle/Resources/public/js/another.coffee'
    filter='coffee'
%}
<script src="{{ asset_url }}" type="text/javascript"></script>
{% endjavascripts %}
```

Both the files will now be served up as a single file compiled into regular JavaScript.

Filtering based on a File Extension

One of the great advantages of using Assetic is reducing the number of asset files to lower HTTP requests. In order to make full use of this, it would be good to combine *all* your JavaScript and CoffeeScript files together since they will ultimately all be served as JavaScript. Unfortunately just adding the JavaScript files to the files to be combined as above will not work as the regular JavaScript files will not survive the CoffeeScript compilation.

This problem can be avoided by using the `apply_to` option in the config, which allows you to specify that a filter should always be applied to particular file extensions. In this case you can specify that the Coffee filter is applied to all `.coffee` files:

Listing 10-4

```
# app/config/config.yml
assetic:
    filters:
        coffee:
            bin: /usr/bin/coffee
            node: /usr/bin/node
            apply_to: "\.coffee$"
```

With this, you no longer need to specify the `coffee` filter in the template. You can also list regular JavaScript files, all of which will be combined and rendered as a single JavaScript file (with only the `.coffee` files being run through the CoffeeScript filter):

Listing 10-5

```
{% javascripts '@AcmeFooBundle/Resources/public/js/example.coffee'
               '@AcmeFooBundle/Resources/public/js/another.coffee'
               '@AcmeFooBundle/Resources/public/js/regular.js'
%}
<script src="{{ asset_url }}" type="text/javascript"></script>
{% endjavascripts %}
```

Chapter 11

How to handle File Uploads with Doctrine

Handling file uploads with Doctrine entities is no different than handling any other file upload. In other words, you're free to move the file in your controller after handling a form submission. For examples of how to do this, see the *file type reference* page.

If you choose to, you can also integrate the file upload into your entity lifecycle (i.e. creation, update and removal). In this case, as your entity is created, updated, and removed from Doctrine, the file uploading and removal processing will take place automatically (without needing to do anything in your controller);

To make this work, you'll need to take care of a number of details, which will be covered in this cookbook entry.

Basic Setup

First, create a simple Doctrine Entity class to work with:

Listing 11-1

```
// src/Acme/DemoBundle/Entity/Document.php
namespace Acme\DemoBundle\Entity;

use Doctrine\ORM\Mapping as ORM;
use Symfony\Component\Validator\Constraints as Assert;

/**
 * @ORM\Entity
 */
class Document
{
    /**
     * @ORM\Id
     * @ORM\Column(type="integer")
     * @ORM\GeneratedValue(strategy="AUTO")
     */
    public $id;

    /**
     * @ORM\Column(type="string", length=255)
     * @Assert\NotBlank
     */
    public $name;

    /**
     * @ORM\Column(type="string", length=255, nullable=true)
     */
    public $path;

    public function getAbsolutePath()
    {
        return null === $this->path ? null : $this->getUploadRootDir().'/'.$this->path;
    }

    public function getWebPath()
    {
        return null === $this->path ? null : $this->getUploadDir().'/'.$this->path;
    }

    protected function getUploadRootDir()
    {
        // the absolute directory path where uploaded documents should be saved
        return __DIR__.'/../../../../web/'.$this->getUploadDir();
    }

    protected function getUploadDir()
    {
        // get rid of the __DIR__ so it doesn't screw when displaying uploaded doc/image in
the view.
        return 'uploads/documents';
    }
}
```

The `Document` entity has a name and it is associated with a file. The `path` property stores the relative path to the file and is persisted to the database. The `getAbsolutePath()` is a convenience method that returns the absolute path to the file while the `getWebPath()` is a convenience method that returns the web path, which can be used in a template to link to the uploaded file.

If you have not done so already, you should probably read the *file* type documentation first to understand how the basic upload process works.

If you're using annotations to specify your validation rules (as shown in this example), be sure that you've enabled validation by annotation (see *validation configuration*).

To handle the actual file upload in the form, use a "virtual" `file` field. For example, if you're building your form directly in a controller, it might look like this:

Listing 11-2
```
public function uploadAction()
{
    // ...

    $form = $this->createFormBuilder($document)
        ->add('name')
        ->add('file')
        ->getForm()
    ;
    // ...
}
```

Next, create this property on your `Document` class and add some validation rules:

Listing 11-3

```
// src/Acme/DemoBundle/Entity/Document.php

// ...
class Document
{
    /**
     * @Assert\File(maxSize="6000000")
     */
    public $file;

    // ...
}
```

As you are using the `File` constraint, Symfony2 will automatically guess that the form field is a file upload input. That's why you did not have to set it explicitly when creating the form above (`->add('file')`).

The following controller shows you how to handle the entire process:

Listing 11-4

```
use Acme\DemoBundle\Entity\Document;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Template;
// ...

/**
 * @Template()
 */
public function uploadAction()
{
    $document = new Document();
    $form = $this->createFormBuilder($document)
        ->add('name')
        ->add('file')
        ->getForm()
    ;

    if ($this->getRequest()->getMethod() === 'POST') {
        $form->bindRequest($this->getRequest());
        if ($form->isValid()) {
            $em = $this->getDoctrine()->getEntityManager();

            $em->persist($document);
            $em->flush();

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

    return array('form' => $form->createView());
}
```

When writing the template, don't forget to set the `enctype` attribute:

Listing 11-5

```
<h1>Upload File</h1>

<form action="#" method="post" {{ form_enctype(form) }}>
    {{ form_widget(form) }}

    <input type="submit" value="Upload Document" />
</form>
```

The previous controller will automatically persist the `Document` entity with the submitted name, but it will do nothing about the file and the `path` property will be blank.

An easy way to handle the file upload is to move it just before the entity is persisted and then set the `path` property accordingly. Start by calling a new `upload()` method on the `Document` class, which you'll create in a moment to handle the file upload:

Listing 11-6

```
if ($form->isValid()) {
    $em = $this->getDoctrine()->getEntityManager();

    $document->upload();

    $em->persist($document);
    $em->flush();

    $this->redirect('...');
}
```

The `upload()` method will take advantage of the *`UploadedFile`*[1] object, which is what's returned after a `file` field is submitted:

Listing 11-7

```
public function upload()
{
    // the file property can be empty if the field is not required
    if (null === $this->file) {
        return;
    }

    // we use the original file name here but you should
    // sanitize it at least to avoid any security issues

    // move takes the target directory and then the target filename to move to
    $this->file->move($this->getUploadRootDir(), $this->file->getClientOriginalName());

    // set the path property to the filename where you've saved the file
    $this->path = $this->file->getClientOriginalName();

    // clean up the file property as you won't need it anymore
    $this->file = null;
}
```

1. `http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/File/UploadedFile.html`

Using Lifecycle Callbacks

Even if this implementation works, it suffers from a major flaw: What if there is a problem when the entity is persisted? The file would have already moved to its final location even though the entity's `path` property didn't persist correctly.

To avoid these issues, you should change the implementation so that the database operation and the moving of the file become atomic: if there is a problem persisting the entity or if the file cannot be moved, then *nothing* should happen.

To do this, you need to move the file right as Doctrine persists the entity to the database. This can be accomplished by hooking into an entity lifecycle callback:

Listing 11-8

```
/**
 * @ORM\Entity
 * @ORM\HasLifecycleCallbacks
 */
class Document
{
}
```

Next, refactor the `Document` class to take advantage of these callbacks:

Listing 11-9

```
use Symfony\Component\HttpFoundation\File\UploadedFile;

/**
 * @ORM\Entity
 * @ORM\HasLifecycleCallbacks
 */
class Document
{
    /**
     * @ORM\PrePersist()
     * @ORM\PreUpdate()
     */
    public function preUpload()
    {
        if (null !== $this->file) {
            // do whatever you want to generate a unique name
            $this->path = uniqid().'.'.$this->file->guessExtension();
        }
    }

    /**
     * @ORM\PostPersist()
     * @ORM\PostUpdate()
     */
    public function upload()
    {
        if (null === $this->file) {
            return;
        }

        // if there is an error when moving the file, an exception will
        // be automatically thrown by move(). This will properly prevent
        // the entity from being persisted to the database on error
        $this->file->move($this->getUploadRootDir(), $this->path);

        unset($this->file);
    }
    /**
     * @ORM\PostRemove()
     */
    public function removeUpload()
    {
        if ($file = $this->getAbsolutePath()) {
            unlink($file);
        }
    }
}
```

The class now does everything you need: it generates a unique filename before persisting, moves the file after persisting, and removes the file if the entity is ever deleted.

Now that the moving of the file is handled atomically by the entity, the call to `$document->upload()` should be removed from the controller:

Listing 11-10

```
if ($form->isValid()) {
    $em = $this->getDoctrine()->getEntityManager();

    $em->persist($document);
    $em->flush();

    $this->redirect('...');
}
```

The `@ORM\PrePersist()` and `@ORM\PostPersist()` event callbacks are triggered before and after the entity is persisted to the database. On the other hand, the `@ORM\PreUpdate()` and `@ORM\PostUpdate()` event callbacks are called when the entity is updated.

The `PreUpdate` and `PostUpdate` callbacks are only triggered if there is a change in one of the entity's field that are persisted. This means that, by default, if you modify only the `$file` property, these events will not be triggered, as the property itself is not directly persisted via Doctrine. One solution would be to use an `updated` field that's persisted to Doctrine, and to modify it manually when changing the file.

Using the `id` as the filename

If you want to use the `id` as the name of the file, the implementation is slightly different as you need to save the extension under the `path` property, instead of the actual filename:

Listing 11-11

```
use Symfony\Component\HttpFoundation\File\UploadedFile;

/**
 * @ORM\Entity
 * @ORM\HasLifecycleCallbacks
 */
class Document
{
    // a property used temporarily while deleting
    private $filenameForRemove;

    /**
     * @ORM\PrePersist()
     * @ORM\PreUpdate()
     */
    public function preUpload()
    {
        if (null !== $this->file) {
            $this->path = $this->file->guessExtension();
        }
    }

    /**
     * @ORM\PostPersist()
     * @ORM\PostUpdate()
     */
    public function upload()
    {
        if (null === $this->file) {
            return;
        }

        // you must throw an exception here if the file cannot be moved
        // so that the entity is not persisted to the database
        // which the UploadedFile move() method does
        $this->file->move($this->getUploadRootDir(),
$this->id.'.'.$this->file->guessExtension());

        unset($this->file);
    }

    /**
     * @ORM\PreRemove()
     */
    public function storeFilenameForRemove()
    {
        $this->filenameForRemove = $this->getAbsolutePath();
    }

    /**
     * @ORM\PostRemove()
     */
    public function removeUpload()
    {
        if ($this->filenameForRemove) {
            unlink($this->filenameForRemove);
        }
    }

    public function getAbsolutePath()
    {
        return null === $this->path ? null :
$this->getUploadRootDir().'/'.$this->id.'.'.$this->path;
    }
}
```

You'll notice in this case that you need to do a little bit more work in order to remove the file. Before it's removed, you must store the file path (since it depends on the id). Then, once the object has been fully removed from the database, you can safely delete the file (in `PostRemove`).

Chapter 12

Doctrine Extensions: Timestampable, Sluggable, Translatable, etc.

Doctrine2 is very flexible, and the community has already created a series of useful Doctrine extensions to help you with common entity-related tasks.

One library in particular - the *DoctrineExtensions*[1] library - provides integration functionality for *Sluggable*[2], *Translatable*[3], *Timestampable*[4], *Loggable*[5], *Tree*[6] and *Sortable*[7] behaviors.

The usage for each of these extensions is explained in that repository.

However, to install/activate each extension you must register and activate an *Event Listener*. To do this, you have two options:

1. Use the *StofDoctrineExtensionsBundle*[8], which integrates the above library.
2. Implement this services directly by following the documentation for integration with Symfony2: *Install Gedmo Doctrine2 extensions in Symfony2*[9]

1. `https://github.com/l3pp4rd/DoctrineExtensions`
2. `https://github.com/l3pp4rd/DoctrineExtensions/blob/master/doc/sluggable.md`
3. `https://github.com/l3pp4rd/DoctrineExtensions/blob/master/doc/translatable.md`
4. `https://github.com/l3pp4rd/DoctrineExtensions/blob/master/doc/timestampable.md`
5. `https://github.com/l3pp4rd/DoctrineExtensions/blob/master/doc/loggable.md`
6. `https://github.com/l3pp4rd/DoctrineExtensions/blob/master/doc/tree.md`
7. `https://github.com/l3pp4rd/DoctrineExtensions/blob/master/doc/sortable.md`
8. `https://github.com/stof/StofDoctrineExtensionsBundle`
9. `https://github.com/l3pp4rd/DoctrineExtensions/blob/master/doc/symfony2.md`

Chapter 13

Registering Event Listeners and Subscribers

Doctrine packages a rich event system that fires events when almost anything happens inside the system. For you, this means that you can create arbitrary *services* and tell Doctrine to notify those objects whenever a certain action (e.g. `prePersist`) happens within Doctrine. This could be useful, for example, to create an independent search index whenever an object in your database is saved.

Doctrine defines two types of objects that can listen to Doctrine events: listeners and subscribers. Both are very similar, but listeners are a bit more straightforward. For more, see *The Event System*[1] on Doctrine's website.

Configuring the Listener/Subscriber

To register a service to act as an event listener or subscriber you just have to *tag* it with the appropriate name. Depending on your use-case, you can hook a listener into every DBAL connection and ORM entity manager or just into one specific DBAL connection and all the entity managers that use this connection.

Listing 13-1

```
doctrine:
    dbal:
        default_connection: default
        connections:
            default:
                driver: pdo_sqlite
                memory: true

services:
    my.listener:
        class: Acme\SearchBundle\Listener\SearchIndexer
        tags:
            - { name: doctrine.event_listener, event: postPersist }
    my.listener2:
        class: Acme\SearchBundle\Listener\SearchIndexer2
        tags:
            - { name: doctrine.event_listener, event: postPersist, connection: default }
    my.subscriber:
```

1. http://docs.doctrine-project.org/projects/doctrine-orm/en/2.1/reference/events.html

```
        class: Acme\SearchBundle\Listener\SearchIndexerSubscriber
        tags:
            - { name: doctrine.event_subscriber, connection: default }
```

Creating the Listener Class

In the previous example, a service `my.listener` was configured as a Doctrine listener on the event `postPersist`. That class behind that service must have a `postPersist` method, which will be called when the event is thrown:

Listing 13-2

```php
// src/Acme/SearchBundle/Listener/SearchIndexer.php
namespace Acme\SearchBundle\Listener;

use Doctrine\ORM\Event\LifecycleEventArgs;
use Acme\StoreBundle\Entity\Product;

class SearchIndexer
{
    public function postPersist(LifecycleEventArgs $args)
    {
        $entity = $args->getEntity();
        $entityManager = $args->getEntityManager();

        // perhaps you only want to act on some "Product" entity
        if ($entity instanceof Product) {
            // do something with the Product
        }
    }
}
```

In each event, you have access to a `LifecycleEventArgs` object, which gives you access to both the entity object of the event and the entity manager itself.

One important thing to notice is that a listener will be listening for *all* entities in your application. So, if you're interested in only handling a specific type of entity (e.g. a `Product` entity but not a `BlogPost` entity), you should check for the class name of the entity in your method (as shown above).

Chapter 14

How to use Doctrine's DBAL Layer

This article is about Doctrine DBAL's layer. Typically, you'll work with the higher level Doctrine ORM layer, which simply uses the DBAL behind the scenes to actually communicate with the database. To read more about the Doctrine ORM, see "*Databases and Doctrine*".

The *Doctrine*[1] Database Abstraction Layer (DBAL) is an abstraction layer that sits on top of *PDO*[2] and offers an intuitive and flexible API for communicating with the most popular relational databases. In other words, the DBAL library makes it easy to execute queries and perform other database actions.

Read the official Doctrine *DBAL Documentation*[3] to learn all the details and capabilities of Doctrine's DBAL library.

To get started, configure the database connection parameters:

Listing 14-1

```
# app/config/config.yml
doctrine:
    dbal:
        driver:   pdo_mysql
        dbname:   Symfony2
        user:     root
        password: null
        charset:  UTF8
```

For full DBAL configuration options, see *Doctrine DBAL Configuration*.

You can then access the Doctrine DBAL connection by accessing the `database_connection` service:

Listing 14-2

```
class UserController extends Controller
{
    public function indexAction()
```

1. `http://www.doctrine-project.org`
2. `http://www.php.net/pdo`
3. `http://docs.doctrine-project.org/projects/doctrine-dbal/en/latest/index.html`

```
    {
        $conn = $this->get('database_connection');
        $users = $conn->fetchAll('SELECT * FROM users');

        // ...
    }
}
```

Registering Custom Mapping Types

You can register custom mapping types through Symfony's configuration. They will be added to all configured connections. For more information on custom mapping types, read Doctrine's *Custom Mapping Types*[4] section of their documentation.

Listing 14-3

```
# app/config/config.yml
doctrine:
    dbal:
        types:
            custom_first: Acme\HelloBundle\Type\CustomFirst
            custom_second: Acme\HelloBundle\Type\CustomSecond
```

Registering Custom Mapping Types in the SchemaTool

The SchemaTool is used to inspect the database to compare the schema. To achieve this task, it needs to know which mapping type needs to be used for each database types. Registering new ones can be done through the configuration.

Let's map the ENUM type (not suppoorted by DBAL by default) to a the `string` mapping type:

Listing 14-4

```
# app/config/config.yml
doctrine:
    dbal:
        connections:
            default:
                // Other connections parameters
                mapping_types:
                    enum: string
```

4. http://docs.doctrine-project.org/projects/doctrine-dbal/en/latest/reference/types.html#custom-mapping-types

Chapter 15

How to generate Entities from an Existing Database

When starting work on a brand new project that uses a database, two different situations comes naturally. In most cases, the database model is designed and built from scratch. Sometimes, however, you'll start with an existing and probably unchangeable database model. Fortunately, Doctrine comes with a bunch of tools to help generate model classes from your existing database.

As the *Doctrine tools documentation*[1] says, reverse engineering is a one-time process to get started on a project. Doctrine is able to convert approximately 70-80% of the necessary mapping information based on fields, indexes and foreign key constraints. Doctrine can't discover inverse associations, inheritance types, entities with foreign keys as primary keys or semantical operations on associations such as cascade or lifecycle events. Some additional work on the generated entities will be necessary afterwards to design each to fit your domain model specificities.

This tutorial assumes you're using a simple blog application with the following two tables: `blog_post` and `blog_comment`. A comment record is linked to a post record thanks to a foreign key constraint.

Listing 15-1

```
CREATE TABLE `blog_post` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT,
  `title` varchar(100) COLLATE utf8_unicode_ci NOT NULL,
  `content` longtext COLLATE utf8_unicode_ci NOT NULL,
  `created_at` datetime NOT NULL,
  PRIMARY KEY (`id`),
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8 COLLATE=utf8_unicode_ci;

CREATE TABLE `blog_comment` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT,
  `post_id` bigint(20) NOT NULL,
  `author` varchar(20) COLLATE utf8_unicode_ci NOT NULL,
  `content` longtext COLLATE utf8_unicode_ci NOT NULL,
  `created_at` datetime NOT NULL,
  PRIMARY KEY (`id`),
```

1. `http://docs.doctrine-project.org/projects/doctrine-orm/en/2.1/reference/tools.html#reverse-engineering`

```
  KEY `blog_comment_post_id_idx` (`post_id`),
  CONSTRAINT `blog_post_id` FOREIGN KEY (`post_id`) REFERENCES `blog_post` (`id`) ON DELETE
CASCADE
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8 COLLATE=utf8_unicode_ci;
```

Before diving into the recipe, be sure your database connection parameters are correctly setup in the `app/config/parameters.ini` file (or wherever your database configuration is kept) and that you have initialized a bundle that will host your future entity class. In this tutorial, we will assume that an `AcmeBlogBundle` exists and is located under the `src/Acme/BlogBundle` folder.

The first step towards building entity classes from an existing database is to ask Doctrine to introspect the database and generate the corresponding metadata files. Metadata files describe the entity class to generate based on tables fields.

Listing 15-2

```
php app/console doctrine:mapping:convert xml ./src/Acme/BlogBundle/Resources/config/doctrine/
metadata/orm --from-database --force
```

This command line tool asks Doctrine to introspect the database and generate the XML metadata files under the `src/Acme/BlogBundle/Resources/config/doctrine/metadata/orm` folder of your bundle.

It's also possible to generate metadata class in YAML format by changing the first argument to *yml*.

The generated `BlogPost.dcm.xml` metadata file looks as follows:

Listing 15-3

```
<?xml version="1.0" encoding="utf-8"?>
<doctrine-mapping>
  <entity name="BlogPost" table="blog_post">
    <change-tracking-policy>DEFERRED_IMPLICIT</change-tracking-policy>
    <id name="id" type="bigint" column="id">
      <generator strategy="IDENTITY"/>
    </id>
    <field name="title" type="string" column="title" length="100"/>
    <field name="content" type="text" column="content"/>
    <field name="isPublished" type="boolean" column="is_published"/>
    <field name="createdAt" type="datetime" column="created_at"/>
    <field name="updatedAt" type="datetime" column="updated_at"/>
    <field name="slug" type="string" column="slug" length="255"/>
    <lifecycle-callbacks/>
  </entity>
</doctrine-mapping>
```

Once the metadata files are generated, you can ask Doctrine to import the schema and build related entity classes by executing the following two commands.

Listing 15-4

```
php app/console doctrine:mapping:import AcmeBlogBundle annotation
php app/console doctrine:generate:entities AcmeBlogBundle
```

The first command generates entity classes with an annotations mapping, but you can of course change the `annotation` argument to `xml` or `yml`. The newly created `BlogComment` entity class looks as follow:

Listing 15-5

```
<?php

// src/Acme/BlogBundle/Entity/BlogComment.php
namespace Acme\BlogBundle\Entity;

use Doctrine\ORM\Mapping as ORM;
```

```
/**
 * Acme\BlogBundle\Entity\BlogComment
 *
 * @ORM\Table(name="blog_comment")
 * @ORM\Entity
 */
class BlogComment
{
    /**
     * @var bigint $id
     *
     * @ORM\Column(name="id", type="bigint", nullable=false)
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="IDENTITY")
     */
    private $id;

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

    /**
     * @var text $content
     *
     * @ORM\Column(name="content", type="text", nullable=false)
     */
    private $content;

    /**
     * @var datetime $createdAt
     *
     * @ORM\Column(name="created_at", type="datetime", nullable=false)
     */
    private $createdAt;

    /**
     * @var BlogPost
     *
     * @ORM\ManyToOne(targetEntity="BlogPost")
     * @ORM\JoinColumn(name="post_id", referencedColumnName="id")
     */
    private $post;
}
```

As you can see, Doctrine converts all table fields to pure private and annotated class properties. The most impressive thing is that it also discovered the relationship with the `BlogPost` entity class based on the foreign key constraint. Consequently, you can find a private `$post` property mapped with a `BlogPost` entity in the `BlogComment` entity class.

The last command generated all getters and setters for your two `BlogPost` and `BlogComment` entity class properties. The generated entities are now ready to be used. Have fun!

Chapter 16

How to work with Multiple Entity Managers

You can use multiple entity managers in a Symfony2 application. This is necessary if you are using different databases or even vendors with entirely different sets of entities. In other words, one entity manager that connects to one database will handle some entities while another entity manager that connects to another database might handle the rest.

Using multiple entity managers is pretty easy, but more advanced and not usually required. Be sure you actually need multiple entity managers before adding in this layer of complexity.

The following configuration code shows how you can configure two entity managers:

Listing 16-1

```
doctrine:
    orm:
        default_entity_manager:   default
        entity_managers:
            default:
                connection:       default
                mappings:
                    AcmeDemoBundle: ~
                    AcmeStoreBundle: ~
            customer:
                connection:       customer
                mappings:
                    AcmeCustomerBundle: ~
```

In this case, you've defined two entity managers and called them `default` and `customer`. The `default` entity manager manages entities in the `AcmeDemoBundle` and `AcmeStoreBundle`, while the `customer` entity manager manages entities in the `AcmeCustomerBundle`.

When working with multiple entity managers, you should be explicit about which entity manager you want. If you *do* omit the entity manager's name when asking for it, the default entity manager (i.e. `default`) is returned:

Listing 16-2

```
class UserController extends Controller
{
    public function indexAction()
    {
        // both return the "default" em
        $em = $this->get('doctrine')->getEntityManager();
        $em = $this->get('doctrine')->getEntityManager('default');

        $customerEm =  $this->get('doctrine')->getEntityManager('customer');
    }
}
```

You can now use Doctrine just as you did before - using the `default` entity manager to persist and fetch entities that it manages and the `customer` entity manager to persist and fetch its entities.

Chapter 17
Registering Custom DQL Functions

Doctrine allows you to specify custom DQL functions. For more information on this topic, read Doctrine's cookbook article "*DQL User Defined Functions*[1]".

In Symfony, you can register your custom DQL functions as follows:

Listing 17-1
```
# app/config/config.yml
doctrine:
    orm:
        # ...
        entity_managers:
            default:
                # ...
                dql:
                    string_functions:
                        test_string: Acme\HelloBundle\DQL\StringFunction
                        second_string: Acme\HelloBundle\DQL\SecondStringFunction
                    numeric_functions:
                        test_numeric: Acme\HelloBundle\DQL\NumericFunction
                    datetime_functions:
                        test_datetime: Acme\HelloBundle\DQL\DatetimeFunction
```

1. http://docs.doctrine-project.org/projects/doctrine-orm/en/2.1/cookbook/dql-user-defined-functions.html

Chapter 18

How to customize Form Rendering

Symfony gives you a wide variety of ways to customize how a form is rendered. In this guide, you'll learn how to customize every possible part of your form with as little effort as possible whether you use Twig or PHP as your templating engine.

Form Rendering Basics

Recall that the label, error and HTML widget of a form field can easily be rendered by using the `form_row` Twig function or the `row` PHP helper method:

```
{{ form_row(form.age) }}
```

Listing 18-1

You can also render each of the three parts of the field individually:

```
<div>
    {{ form_label(form.age) }}
    {{ form_errors(form.age) }}
    {{ form_widget(form.age) }}
</div>
```

Listing 18-2

In both cases, the form label, errors and HTML widget are rendered by using a set of markup that ships standard with Symfony. For example, both of the above templates would render:

```
<div>
    <label for="form_age">Age</label>
    <ul>
        <li>This field is required</li>
    </ul>
    <input type="number" id="form_age" name="form[age]" />
</div>
```

Listing 18-3

To quickly prototype and test a form, you can render the entire form with just one line:

```
{{ form_widget(form) }}
```

Listing 18-4

The remainder of this recipe will explain how every part of the form's markup can be modified at several different levels. For more information about form rendering in general, see *Rendering a Form in a Template*.

What are Form Themes?

Symfony uses form fragments - a small piece of a template that renders just one part of a form - to render every part of a form - - field labels, errors, `input` text fields, `select` tags, etc

The fragments are defined as blocks in Twig and as template files in PHP.

A *theme* is nothing more than a set of fragments that you want to use when rendering a form. In other words, if you want to customize one portion of how a form is rendered, you'll import a *theme* which contains a customization of the appropriate form fragments.

Symfony comes with a default theme (*form_div_layout.html.twig*[1] in Twig and `FrameworkBundle:Form` in PHP) that defines each and every fragment needed to render every part of a form.

In the next section you will learn how to customize a theme by overriding some or all of its fragments.

For example, when the widget of a `integer` type field is rendered, an `input number` field is generated

Listing 18-5
```
{{ form_widget(form.age) }}
```

renders:

Listing 18-6
```
<input type="number" id="form_age" name="form[age]" required="required" value="33" />
```

Internally, Symfony uses the `integer_widget` fragment to render the field. This is because the field type is `integer` and you're rendering its `widget` (as opposed to its `label` or `errors`).

In Twig that would default to the block `integer_widget` from the *form_div_layout.html.twig*[2] template.

In PHP it would rather be the `integer_widget.html.php` file located in `FrameworkBundle/Resources/views/Form` folder.

The default implementation of the `integer_widget` fragment looks like this:

Listing 18-7
```
{% block integer_widget %}
    {% set type = type|default('number') %}
    {{ block('field_widget') }}
{% endblock integer_widget %}
```

As you can see, this fragment itself renders another fragment - `field_widget`:

Listing 18-8
```
{% block field_widget %}
    {% set type = type|default('text') %}
    <input type="{{ type }}" {{ block('widget_attributes') }} value="{{ value }}" />
{% endblock field_widget %}
```

The point is, the fragments dictate the HTML output of each part of a form. To customize the form output, you just need to identify and override the correct fragment. A set of these form fragment customizations is known as a form "theme". When rendering a form, you can choose which form theme(s) you want to apply.

In Twig a theme is a single template file and the fragments are the blocks defined in this file.

In PHP a theme is a folder and the the fragments are individual template files in this folder.

1. https://github.com/symfony/symfony/blob/master/src/Symfony/Bridge/Twig/Resources/views/Form/form_div_layout.html.twig
2. https://github.com/symfony/symfony/blob/master/src/Symfony/Bridge/Twig/Resources/views/Form/form_div_layout.html.twig

Knowing which block to customize

In this example, the customized fragment name is `integer_widget` because you want to override the HTML `widget` for all `integer` field types. If you need to customize textarea fields, you would customize `textarea_widget`.

As you can see, the fragment name is a combination of the field type and which part of the field is being rendered (e.g. `widget`, `label`, `errors`, `row`). As such, to customize how errors are rendered for just input `text` fields, you should customize the `text_errors` fragment.

More commonly, however, you'll want to customize how errors are displayed across *all* fields. You can do this by customizing the `field_errors` fragment. This takes advantage of field type inheritance. Specifically, since the `text` type extends from the `field` type, the form component will first look for the type-specific fragment (e.g. `text_errors`) before falling back to its parent fragment name if it doesn't exist (e.g. `field_errors`).

For more information on this topic, see *Form Fragment Naming*.

Form Theming

To see the power of form theming, suppose you want to wrap every input `number` field with a `div` tag. The key to doing this is to customize the `integer_widget` fragment.

Form Theming in Twig

When customizing the form field block in Twig, you have two options on *where* the customized form block can live:

Method	**Pros**	**Cons**
Inside the same template as the form	Quick and easy	Can't be reused in other templates
Inside a separate template	Can be reused by many templates	Requires an extra template to be created

Both methods have the same effect but are better in different situations.

Method 1: Inside the same Template as the Form

The easiest way to customize the `integer_widget` block is to customize it directly in the template that's actually rendering the form.

Listing 18-9

```
{% extends '::base.html.twig' %}

{% form_theme form _self %}

{% block integer_widget %}
    <div class="integer_widget">
        {% set type = type|default('number') %}
        {{ block('field_widget') }}
    </div>
{% endblock %}

{% block content %}
```

```
    {# render the form #}

    {{ form_row(form.age) }}
{% endblock %}
```

By using the special `{% form_theme form _self %}` tag, Twig looks inside the same template for any overridden form blocks. Assuming the `form.age` field is an `integer` type field, when its widget is rendered, the customized `integer_widget` block will be used.

The disadvantage of this method is that the customized form block can't be reused when rendering other forms in other templates. In other words, this method is most useful when making form customizations that are specific to a single form in your application. If you want to reuse a form customization across several (or all) forms in your application, read on to the next section.

Method 2: Inside a Separate Template

You can also choose to put the customized `integer_widget` form block in a separate template entirely. The code and end-result are the same, but you can now re-use the form customization across many templates:

Listing 18-10

```
{# src/Acme/DemoBundle/Resources/views/Form/fields.html.twig #}

{% block integer_widget %}
    <div class="integer_widget">
        {% set type = type|default('number') %}
        {{ block('field_widget') }}
    </div>
{% endblock %}
```

Now that you've created the customized form block, you need to tell Symfony to use it. Inside the template where you're actually rendering your form, tell Symfony to use the template via the `form_theme` tag:

Listing 18-11

```
{% form_theme form 'AcmeDemoBundle:Form:fields.html.twig' %}

{{ form_widget(form.age) }}
```

When the `form.age` widget is rendered, Symfony will use the `integer_widget` block from the new template and the `input` tag will be wrapped in the `div` element specified in the customized block.

Form Theming in PHP

When using PHP as a templating engine, the only method to customize a fragment is to create a new template file - this is similar to the second method used by Twig.

The template file must be named after the fragment. You must create a `integer_widget.html.php` file in order to customize the `integer_widget` fragment.

Listing 18-12

```
<!-- src/Acme/DemoBundle/Resources/views/Form/integer_widget.html.php -->

<div class="integer_widget">
    <?php echo $view['form']->renderBlock('field_widget', array('type' => isset($type) ? $type
: "number")) ?>
</div>
```

Now that you've created the customized form template, you need to tell Symfony to use it. Inside the template where you're actually rendering your form, tell Symfony to use the theme via the `setTheme` helper method:

Listing 18-13

```
<?php $view['form']->setTheme($form, array('AcmeDemoBundle:Form')) ;?>

<?php $view['form']->widget($form['age']) ?>
```

When the `form.age` widget is rendered, Symfony will use the customized `integer_widget.html.php` template and the `input` tag will be wrapped in the `div` element.

Referencing Base Form Blocks (Twig specific)

So far, to override a particular form block, the best method is to copy the default block from *form_div_layout.html.twig*[3], paste it into a different template, and the customize it. In many cases, you can avoid doing this by referencing the base block when customizing it.

This is easy to do, but varies slightly depending on if your form block customizations are in the same template as the form or a separate template.

Referencing Blocks from inside the same Template as the Form

Import the blocks by adding a `use` tag in the template where you're rendering the form:

Listing 18-14

```
{% use 'form_div_layout.html.twig' with integer_widget as base_integer_widget %}
```

Now, when the blocks from *form_div_layout.html.twig*[4] are imported, the `integer_widget` block is called `base_integer_widget`. This means that when you redefine the `integer_widget` block, you can reference the default markup via `base_integer_widget`:

Listing 18-15

```
{% block integer_widget %}
    <div class="integer_widget">
        {{ block('base_integer_widget') }}
    </div>
{% endblock %}
```

Referencing Base Blocks from an External Template

If your form customizations live inside an external template, you can reference the base block by using the `parent()` Twig function:

Listing 18-16

```
{# src/Acme/DemoBundle/Resources/views/Form/fields.html.twig #}

{% extends 'form_div_layout.html.twig' %}

{% block integer_widget %}
    <div class="integer_widget">
        {{ parent() }}
    </div>
{% endblock %}
```

3. https://github.com/symfony/symfony/blob/master/src/Symfony/Bridge/Twig/Resources/views/Form/form_div_layout.html.twig
4. https://github.com/symfony/symfony/blob/master/src/Symfony/Bridge/Twig/Resources/views/Form/form_div_layout.html.twig

It is not possible to reference the base block when using PHP as the templating engine. You have to manually copy the content from the base block to your new template file.

Making Application-wide Customizations

If you'd like a certain form customization to be global to your application, you can accomplish this by making the form customizations in an external template and then importing it inside your application configuration:

Twig

By using the following configuration, any customized form blocks inside the `AcmeDemoBundle:Form:fields.html.twig` template will be used globally when a form is rendered.

Listing 18-17

```yaml
# app/config/config.yml

twig:
    form:
        resources:
            - 'AcmeDemoBundle:Form:fields.html.twig'
    # ...
```

By default, Twig uses a *div* layout when rendering forms. Some people, however, may prefer to render forms in a *table* layout. Use the `form_table_layout.html.twig` resource to use such a layout:

Listing 18-18

```yaml
# app/config/config.yml

twig:
    form:
        resources: ['form_table_layout.html.twig']
    # ...
```

If you only want to make the change in one template, add the following line to your template file rather than adding the template as a resource:

Listing 18-19

```
{% form_theme form 'form_table_layout.html.twig' %}
```

Note that the `form` variable in the above code is the form view variable that you passed to your template.

PHP

By using the following configuration, any customized form fragments inside the `src/Acme/DemoBundle/Resources/views/Form` folder will be used globally when a form is rendered.

Listing 18-20

```yaml
# app/config/config.yml

framework:
    templating:
        form:
            resources:
                - 'AcmeDemoBundle:Form'
    # ...
```

By default, the PHP engine uses a *div* layout when rendering forms. Some people, however, may prefer to render forms in a *table* layout. Use the FrameworkBundle:FormTable resource to use such a layout:

Listing 18-21

```
# app/config/config.yml

framework:
    templating:
        form:
            resources:
                - 'FrameworkBundle:FormTable'
```

If you only want to make the change in one template, add the following line to your template file rather than adding the template as a resource:

Listing 18-22

```
<?php $view['form']->setTheme($form, array('FrameworkBundle:FormTable')); ?>
```

Note that the $form variable in the above code is the form view variable that you passed to your template.

How to customize an Individual field

So far, you've seen the different ways you can customize the widget output of all text field types. You can also customize individual fields. For example, suppose you have two text fields - first_name and last_name - but you only want to customize one of the fields. This can be accomplished by customizing a fragment whose name is a combination of the field id attribute and which part of the field is being customized. For example:

Listing 18-23

```
{% form_theme form _self %}

{% block _product_name_widget %}
    <div class="text_widget">
        {{ block('field_widget') }}
    </div>
{% endblock %}

{{ form_widget(form.name) }}
```

Here, the _product_name_widget fragment defines the template to use for the field whose *id* is product_name (and name is product[name]).

The product portion of the field is the form name, which may be set manually or generated automatically based on your form type name (e.g. ProductType equates to product). If you're not sure what your form name is, just view the source of your generated form.

You can also override the markup for an entire field row using the same method:

Listing 18-24

```
{% form_theme form _self %}

{% block _product_name_row %}
    <div class="name_row">
        {{ form_label(form) }}
        {{ form_errors(form) }}
        {{ form_widget(form) }}
    </div>
{% endblock %}
```

Other Common Customizations

So far, this recipe has shown you several different ways to customize a single piece of how a form is rendered. The key is to customize a specific fragment that corresponds to the portion of the form you want to control (see *naming form blocks*).

In the next sections, you'll see how you can make several common form customizations. To apply these customizations, use one of the methods described in the *Form Theming* section.

Customizing Error Output

The form component only handles *how* the validation errors are rendered, and not the actual validation error messages. The error messages themselves are determined by the validation constraints you apply to your objects. For more information, see the chapter on *validation*.

There are many different ways to customize how errors are rendered when a form is submitted with errors. The error messages for a field are rendered when you use the `form_errors` helper:

Listing 18-25

```
{{ form_errors(form.age) }}
```

By default, the errors are rendered inside an unordered list:

Listing 18-26

```
<ul>
    <li>This field is required</li>
</ul>
```

To override how errors are rendered for *all* fields, simply copy, paste and customize the `field_errors` fragment.

Listing 18-27

```
{% block field_errors %}
{% spaceless %}
    {% if errors|length > 0 %}
    <ul class="error_list">
        {% for error in errors %}
            <li>{{ error.messageTemplate|trans(error.messageParameters, 'validators') }}</li>
        {% endfor %}
    </ul>
    {% endif %}
{% endspaceless %}
{% endblock field_errors %}
```

See *Form Theming* for how to apply this customization.

You can also customize the error output for just one specific field type. For example, certain errors that are more global to your form (i.e. not specific to just one field) are rendered separately, usually at the top of your form:

Listing 18-28

```
{{ form_errors(form) }}
```

To customize *only* the markup used for these errors, follow the same directions as above, but now call the block `form_errors` (Twig) / the file `form_errors.html.php` (PHP). Now, when errors for the `form` type are rendered, your customized fragment will be used instead of the default `field_errors`.

Customizing the "Form Row"

When you can manage it, the easiest way to render a form field is via the `form_row` function, which renders the label, errors and HTML widget of a field. To customize the markup used for rendering *all* form field rows, override the `field_row` fragment. For example, suppose you want to add a class to the `div` element around each row:

Listing 18-29

```
{% block field_row %}
    <div class="form_row">
        {{ form_label(form) }}
        {{ form_errors(form) }}
        {{ form_widget(form) }}
    </div>
{% endblock field_row %}
```

See *Form Theming* for how to apply this customization.

Adding a "Required" Asterisk to Field Labels

If you want to denote all of your required fields with a required asterisk (*), you can do this by customizing the `field_label` fragment.

In Twig, if you're making the form customization inside the same template as your form, modify the `use` tag and add the following:

Listing 18-30

```
{% use 'form_div_layout.html.twig' with field_label as base_field_label %}

{% block field_label %}
    {{ block('base_field_label') }}

    {% if required %}
        <span class="required" title="This field is required">*</span>
    {% endif %}
{% endblock %}
```

In Twig, if you're making the form customization inside a separate template, use the following:

Listing 18-31

```
{% extends 'form_div_layout.html.twig' %}

{% block field_label %}
    {{ parent() }}

    {% if required %}
        <span class="required" title="This field is required">*</span>
    {% endif %}
{% endblock %}
```

When using PHP as a templating engine you have to copy the content from the original template:

Listing 18-32

```
<!-- field_label.html.php -->

<!-- original content -->
<label for="<?php echo $view->escape($id) ?>" <?php foreach($attr as $k => $v) {
printf('%s="%s" ', $view->escape($k), $view->escape($v)); } ?>><?php echo
$view->escape($view['translator']->trans($label)) ?></label>
```

```
<!-- customization -->
<?php if ($required) : ?>
    <span class="required" title="This field is required">*</span>
<?php endif ?>
```

See *Form Theming* for how to apply this customization.

Adding "help" messages

You can also customize your form widgets to have an optional "help" message.

In Twig, If you're making the form customization inside the same template as your form, modify the `use` tag and add the following:

Listing 18-33
```
{% use 'form_div_layout.html.twig' with field_widget as base_field_widget %}

{% block field_widget %}
    {{ block('base_field_widget') }}

    {% if help is defined %}
        <span class="help">{{ help }}</span>
    {% endif %}
{% endblock %}
```

In twig, If you're making the form customization inside a separate template, use the following:

Listing 18-34
```
{% extends 'form_div_layout.html.twig' %}

{% block field_widget %}
    {{ parent() }}

    {% if help is defined %}
        <span class="help">{{ help }}</span>
    {% endif %}
{% endblock %}
```

When using PHP as a templating engine you have to copy the content from the original template:

Listing 18-35
```
<!-- field_widget.html.php -->

<!-- Original content -->
<input
    type="<?php echo isset($type) ? $view->escape($type) : "text" ?>"
    value="<?php echo $view->escape($value) ?>"
    <?php echo $view['form']->renderBlock('attributes') ?>
/>

<!-- Customization -->
<?php if (isset($help)) : ?>
    <span class="help"><?php echo $view->escape($help) ?></span>
<?php endif ?>
```

To render a help message below a field, pass in a `help` variable:

Listing 18-36
```
{{ form_widget(form.title, { 'help': 'foobar' }) }}
```

See *Form Theming* for how to apply this customization.

Chapter 19

Using Data Transformers

You'll often find the need to transform the data the user entered in a form into something else for use in your program. You could easily do this manually in your controller, but what if you want to use this specific form in different places?

Say you have a one-to-one relation of Task to Issue, e.g. a Task optionally has an issue linked to it. Adding a listbox with all possible issues can eventually lead to a really long listbox in which it is impossible to find something. You'll rather want to add a textbox, in which the user can simply enter the number of the issue. In the controller you can convert this issue number to an actual task, and eventually add errors to the form if it was not found, but of course this is not really clean.

It would be better if this issue was automatically looked up and converted to an Issue object, for use in your action. This is where Data Transformers come into play.

First, create a custom form type which has a Data Transformer attached to it, which returns the Issue by number: the issue selector type. Eventually this will simply be a text field, as we configure the fields' parent to be a "text" field, in which you will enter the issue number. The field will display an error if a non existing number was entered:

Listing 19-1

```
// src/Acme/TaskBundle/Form/Type/IssueSelectorType.php
namespace Acme\TaskBundle\Form\Type;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilder;
use Acme\TaskBundle\Form\DataTransformer\IssueToNumberTransformer;
use Doctrine\Common\Persistence\ObjectManager;

class IssueSelectorType extends AbstractType
{
    /**
     * @var ObjectManager
     */
    private $om;

    /**
     * @param ObjectManager $om
     */
    public function __construct(ObjectManager $om)
    {
        $this->om = $om;
    }

    public function buildForm(FormBuilder $builder, array $options)
    {
        $transformer = new IssueToNumberTransformer($this->om);
        $builder->appendClientTransformer($transformer);
    }

    public function getDefaultOptions(array $options)
    {
        return array(
            'invalid_message' => 'The selected issue does not exist',
        );
    }

    public function getParent(array $options)
    {
        return 'text';
    }

    public function getName()
    {
        return 'issue_selector';
    }
}
```

You can also use transformers without creating a new custom form type by calling `appendClientTransformer` on any field builder:

Listing 19-2

```
use Acme\TaskBundle\Form\DataTransformer\IssueToNumberTransformer;

class TaskType extends AbstractType
{
    public function buildForm(FormBuilder $builder, array $options)
    {
        // ...

        // this assumes that the entity manager was passed in as an option
        $entityManager = $options['em'];
        $transformer = new IssueToNumberTransformer($entityManager);

        // use a normal text field, but transform the text into an issue object
        $builder
            ->add('issue', 'text')
            ->appendClientTransformer($transformer)
        ;
    }

    // ...
}
```

Next, we create the data transformer, which does the actual conversion:

Listing 19-3

```
// src/Acme/TaskBundle/Form/DataTransformer/IssueToNumberTransformer.php

namespace Acme\TaskBundle\Form\DataTransformer;

use Symfony\Component\Form\DataTransformerInterface;
use Symfony\Component\Form\Exception\TransformationFailedException;
use Doctrine\Common\Persistence\ObjectManager;
use Acme\TaskBundle\Entity\Issue;

class IssueToNumberTransformer implements DataTransformerInterface
{
    /**
     * @var ObjectManager
     */
    private $om;

    /**
     * @param ObjectManager $om
     */
    public function __construct(ObjectManager $om)
    {
        $this->om = $om;
    }

    /**
     * Transforms an object (issue) to a string (number).
     *
     * @param  Issue|null $issue
     * @return string
     */
    public function transform($issue)
    {
        if (null === $issue) {
            return "";
        }

        return $issue->getNumber();
    }

    /**
     * Transforms a string (number) to an object (issue).
     *
     * @param  string $number
     * @return Issue|null
     * @throws TransformationFailedException if object (issue) is not found.
     */
    public function reverseTransform($number)
    {
        if (!$number) {
            return null;
        }

        $issue = $this->om
            ->getRepository('AcmeTaskBundle:Issue')
            ->findOneBy(array('number' => $number))
        ;

        if (null === $issue) {
            throw new TransformationFailedException(sprintf(
                'An issue with number "%s" does not exist!',
                $number
            ));
        }

        return $issue;
    }
}
```

Finally, since we've decided to create a custom form type that uses the data transformer, register the Type in the service container, so that the entity manager can be automatically injected:

Listing 19-4

```
services:
    acme_demo.type.issue_selector:
        class: Acme\TaskBundle\Form\Type\IssueSelectorType
        arguments: ["@doctrine.orm.entity_manager"]
        tags:
            - { name: form.type, alias: issue_selector }
```

You can now add the type to your form by its alias as follows:

Listing 19-5

```
// src/Acme/TaskBundle/Form/Type/TaskType.php

namespace Acme\TaskBundle\Form\Type;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilder;

class TaskType extends AbstractType
{
    public function buildForm(FormBuilder $builder, array $options)
    {
        $builder
            ->add('task')
            ->add('dueDate', null, array('widget' => 'single_text'));
            ->add('issue', 'issue_selector')
        ;
    }

    public function getName()
    {
        return 'task';
    }
}
```

Now it will be very easy at any random place in your application to use this selector type to select an issue by number. No logic has to be added to your Controller at all.

If you want a new issue to be created when an unknown number is entered, you can instantiate it rather than throwing the TransformationFailedException, and even persist it to your entity manager if the task has no cascading options for the issue.

Chapter 20

How to Dynamically Generate Forms Using Form Events

Before jumping right into dynamic form generation, let's have a quick review of what a bare form class looks like:

Listing 20-1

```php
//src/Acme/DemoBundle/Form/ProductType.php
namespace Acme\DemoBundle\Form;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilder;

class ProductType extends AbstractType
{
    public function buildForm(FormBuilder $builder, array $options)
    {
        $builder->add('name');
        $builder->add('price');
    }

    public function getName()
    {
        return 'product';
    }
}
```

If this particular section of code isn't already familiar to you, you probably need to take a step back and first review the *Forms chapter* before proceeding.

Let's assume for a moment that this form utilizes an imaginary "Product" class that has only two relevant properties ("name" and "price"). The form generated from this class will look the exact same regardless of a new Product is being created or if an existing product is being edited (e.g. a product fetched from the database).

Suppose now, that you don't want the user to be able to change the *name* value once the object has been created. To do this, you can rely on Symfony's *Event Dispatcher* system to analyze the data on the object and modify the form based on the Product object's data. In this entry, you'll learn how to add this level of flexibility to your forms.

Adding An Event Subscriber To A Form Class

So, instead of directly adding that "name" widget via our ProductType form class, let's delegate the responsibility of creating that particular field to an Event Subscriber:

Listing 20-2

```
//src/Acme/DemoBundle/Form/ProductType.php
namespace Acme\DemoBundle\Form

use Symfony\Component\Form\AbstractType
use Symfony\Component\Form\FormBuilder;
use Acme\DemoBundle\Form\EventListener\AddNameFieldSubscriber;

class ProductType extends AbstractType
{
    public function buildForm(FormBuilder $builder, array $options)
    {
        $subscriber = new AddNameFieldSubscriber($builder->getFormFactory());
        $builder->addEventSubscriber($subscriber);
        $builder->add('price');
    }

    public function getName()
    {
        return 'product';
    }
}
```

The event subscriber is passed the FormFactory object in its constructor so that our new subscriber is capable of creating the form widget once it is notified of the dispatched event during form creation.

Inside the Event Subscriber Class

The goal is to create a "name" field *only* if the underlying Product object is new (e.g. hasn't been persisted to the database). Based on that, the subscriber might look like the following:

Listing 20-3

```
// src/Acme/DemoBundle/Form/EventListener/AddNameFieldSubscriber.php
namespace Acme\DemoBundle\Form\EventListener;

use Symfony\Component\Form\Event\DataEvent;
use Symfony\Component\Form\FormFactoryInterface;
use Symfony\Component\EventDispatcher\EventSubscriberInterface;
use Symfony\Component\Form\FormEvents;

class AddNameFieldSubscriber implements EventSubscriberInterface
{
    private $factory;

    public function __construct(FormFactoryInterface $factory)
    {
        $this->factory = $factory;
    }
    public static function getSubscribedEvents()
    {
        // Tells the dispatcher that we want to listen on the form.pre_set_data
        // event and that the preSetData method should be called.
        return array(FormEvents::PRE_SET_DATA => 'preSetData');
    }

    public function preSetData(DataEvent $event)
    {
        $data = $event->getData();
        $form = $event->getForm();

        // During form creation setData() is called with null as an argument
        // by the FormBuilder constructor. We're only concerned with when
        // setData is called with an actual Entity object in it (whether new,
        // or fetched with Doctrine). This if statement let's us skip right
        // over the null condition.
        if (null === $data) {
            return;
        }

        // check if the product object is "new"
        if (!$data->getId()) {
            $form->add($this->factory->createNamed('text', 'name'));
        }
    }
}
```

It is easy to misunderstand the purpose of the `if (null === $data)` segment of this event subscriber. To fully understand its role, you might consider also taking a look at the *Form class*[1] and paying special attention to where setData() is called at the end of the constructor, as well as the setData() method itself.

The `FormEvents::PRE_SET_DATA` line actually resolves to the string `form.pre_set_data`. The *FormEvents class*[2] serves an organizational purpose. It is a centralized location in which you can find all of the various form events available.

While this example could have used the `form.set_data` event or even the `form.post_set_data` events just as effectively, by using `form.pre_set_data` we guarantee that the data being retrieved from the `Event` object has in no way been modified by any other subscribers or listeners. This is because `form.pre_set_data` passes a *DataEvent*[3] object instead of the *FilterDataEvent*[4] object passed by the `form.set_data` event. *DataEvent*[5], unlike its child *FilterDataEvent*[6], lacks a setData() method.

You may view the full list of form events via the *FormEvents class*[7], found in the form bundle.

1. `https://github.com/symfony/symfony/blob/master/src/Symfony/Component/Form/Form.php`
2. `https://github.com/symfony/Form/blob/master/FormEvents.php`
3. `https://github.com/symfony/symfony/blob/master/src/Symfony/Component/Form/Event/DataEvent.php`
4. `https://github.com/symfony/symfony/blob/master/src/Symfony/Component/Form/Event/FilterDataEvent.php`
5. `https://github.com/symfony/symfony/blob/master/src/Symfony/Component/Form/Event/DataEvent.php`
6. `https://github.com/symfony/symfony/blob/master/src/Symfony/Component/Form/Event/FilterDataEvent.php`
7. `https://github.com/symfony/Form/blob/master/FormEvents.php`

Chapter 21

How to Embed a Collection of Forms

In this entry, you'll learn how to create a form that embeds a collection of many other forms. This could be useful, for example, if you had a `Task` class and you wanted to edit/create/remove many `Tag` objects related to that Task, right inside the same form.

In this entry, we'll loosely assume that you're using Doctrine as your database store. But if you're not using Doctrine (e.g. Propel or just a database connection), it's all very similar. There are only a few parts of this tutorial that really care about "persistence".

If you *are* using Doctrine, you'll need to add the Doctrine metadata, including the `ManyToMany` on the Task's `tags` property.

Let's start there: suppose that each `Task` belongs to multiple `Tags` objects. Start by creating a simple `Task` class:

Listing 21-1

```
// src/Acme/TaskBundle/Entity/Task.php
namespace Acme\TaskBundle\Entity;

use Doctrine\Common\Collections\ArrayCollection;

class Task
{
    protected $description;

    protected $tags;

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

    public function getDescription()
    {
        return $this->description;
    }

    public function setDescription($description)
    {
        $this->description = $description;
    }

    public function getTags()
    {
        return $this->tags;
    }

    public function setTags(ArrayCollection $tags)
    {
        $this->tags = $tags;
    }
}
```

The `ArrayCollection` is specific to Doctrine and is basically the same as using an `array` (but it must be an `ArrayCollection`) if you're using Doctrine.

Now, create a `Tag` class. As you saw above, a `Task` can have many `Tag` objects:

Listing 21-2

```
// src/Acme/TaskBundle/Entity/Tag.php
namespace Acme\TaskBundle\Entity;

class Tag
{
    public $name;
}
```

The `name` property is public here, but it can just as easily be protected or private (but then it would need `getName` and `setName` methods).

Now let's get to the forms. Create a form class so that a `Tag` object can be modified by the user:

Listing 21-3

```
// src/Acme/TaskBundle/Form/Type/TagType.php
namespace Acme\TaskBundle\Form\Type;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilder;

class TagType extends AbstractType
{
    public function buildForm(FormBuilder $builder, array $options)
    {
        $builder->add('name');
    }

    public function getDefaultOptions(array $options)
    {
        return array(
            'data_class' => 'Acme\TaskBundle\Entity\Tag',
        );
    }

    public function getName()
    {
        return 'tag';
    }
}
```

With this, we have enough to render a tag form by itself. But since the end goal is to allow the tags of a Task to be modified right inside the task form itself, create a form for the Task class.

Notice that we embed a collection of TagType forms using the *collection* field type:

Listing 21-4

```
// src/Acme/TaskBundle/Form/Type/TaskType.php
namespace Acme\TaskBundle\Form\Type;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilder;

class TaskType extends AbstractType
{
    public function buildForm(FormBuilder $builder, array $options)
    {
        $builder->add('description');

        $builder->add('tags', 'collection', array('type' => new TagType()));
    }

    public function getDefaultOptions(array $options)
    {
        return array(
            'data_class' => 'Acme\TaskBundle\Entity\Task',
        );
    }

    public function getName()
    {
        return 'task';
    }
}
```

In your controller, you'll now initialize a new instance of TaskType:

Listing 21-5

```
// src/Acme/TaskBundle/Controller/TaskController.php
namespace Acme\TaskBundle\Controller;

use Acme\TaskBundle\Entity\Task;
use Acme\TaskBundle\Entity\Tag;
use Acme\TaskBundle\Form\Type\TaskType;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class TaskController extends Controller
{
    public function newAction(Request $request)
    {
        $task = new Task();

        // dummy code - this is here just so that the Task has some tags
        // otherwise, this isn't an interesting example
        $tag1 = new Tag();
        $tag1->name = 'tag1';
        $task->getTags()->add($tag1);
        $tag2 = new Tag();
        $tag2->name = 'tag2';
        $task->getTags()->add($tag2);
        // end dummy code

        $form = $this->createForm(new TaskType(), $task);

        // process the form on POST
        if ('POST' === $request->getMethod()) {
            $form->bindRequest($request);
            if ($form->isValid()) {
                // maybe do some form processing, like saving the Task and Tag objects
            }
        }

        return $this->render('AcmeTaskBundle:Task:new.html.twig', array(
            'form' => $form->createView(),
        ));
    }
}
```

The corresponding template is now able to render both the `description` field for the task form as well as all the `TagType` forms for any tags that are already related to this `Task`. In the above controller, I added some dummy code so that you can see this in action (since a `Task` has zero tags when first created).

Listing 21-6

```
{# src/Acme/TaskBundle/Resources/views/Task/new.html.twig #}
{# ... #}

<form action="..." method="POST" {{ form_enctype(form) }}>
    {# render the task's only field: description #}
    {{ form_row(form.description) }}

    <h3>Tags</h3>
    <ul class="tags">
        {# iterate over each existing tag and render its only field: name #}
        {% for tag in form.tags %}
            <li>{{ form_row(tag.name) }}</li>
        {% endfor %}
    </ul>

    {{ form_rest(form) }}
    {# ... #}
</form>
```

When the user submits the form, the submitted data for the `Tags` fields are used to construct an ArrayCollection of `Tag` objects, which is then set on the `tag` field of the `Task` instance.

The `Tags` collection is accessible naturally via `$task->getTags()` and can be persisted to the database or used however you need.

So far, this works great, but this doesn't allow you to dynamically add new tags or delete existing tags. So, while editing existing tags will work great, your user can't actually add any new tags yet.

Allowing "new" tags with the "prototype"

Allowing the user to dynamically add new tags means that we'll need to use some JavaScript. Previously we added two tags to our form in the controller. Now we need to let the user add as many tag forms as he needs directly in the browser. This will be done through a bit of JavaScript.

The first thing we need to do is to let the form collection know that it will receive an unknown number of tags. So far we've added two tags and the form type expects to receive exactly two, otherwise an error will be thrown: `This form should not contain extra fields`. To make this flexible, we add the `allow_add` option to our collection field:

Listing 21-7

```php
// src/Acme/TaskBundle/Form/Type/TaskType.php
// ...

public function buildForm(FormBuilder $builder, array $options)
{
    $builder->add('description');

    $builder->add('tags', 'collection', array(
        'type' => new TagType(),
        'allow_add' => true,
        'by_reference' => false,
    ));
}
```

Note that we also added `'by_reference' => false`. Normally, the form framework would modify the tags on a *Task* object *without* actually ever calling *setTags*. By setting *by_reference* to *false*, *setTags* will be called. This will be important later as you'll see.

In addition to telling the field to accept any number of submitted objects, the `allow_add` also makes a "prototype" variable available to you. This "prototype" is a little "template" that contains all the HTML to be able to render any new "tag" forms. To render it, make the following change to your template:

Listing 21-8

```
<ul class="tags" data-prototype="{{ form_widget(form.tags.get('prototype')) | e }}">
    ...
</ul>
```

If you render your whole "tags" sub-form at once (e.g. `form_row(form.tags)`), then the prototype is automatically available on the outer `div` as the `data-prototype` attribute, similar to what you see above.

The `form.tags.get('prototype')` is form element that looks and feels just like the individual `form_widget(tag)` elements inside our `for` loop. This means that you can call `form_widget`, `form_row`, or `form_label` on it. You could even choose to render only one of its fields (e.g. the `name` field):

Listing 21-9

```
{{ form_widget(form.tags.get('prototype').name) | e }}
```

On the rendered page, the result will look something like this:

Listing 21-10

```
<ul class="tags" data-prototype="&lt;div&gt;&lt;label class=&quot;
required&quot;&gt;$$name$$&lt;/label&gt;&lt;div
id=&quot;task_tags_$$name$$&quot;&gt;&lt;div&gt;&lt;label
for=&quot;task_tags_$$name$$_name&quot; class=&quot; required&quot;&gt;Name&lt;/
label&gt;&lt;input type=&quot;text&quot; id=&quot;task_tags_$$name$$_name&quot;
name=&quot;task[tags][$$name$$][name]&quot; required=&quot;required&quot;
maxlength=&quot;255&quot; /&gt;&lt;/div&gt;&lt;/div&gt;&lt;/div&gt;">
```

The goal of this section will be to use JavaScript to read this attribute and dynamically add new tag forms when the user clicks a "Add a tag" link. To make things simple, we'll use jQuery and assume you have it included somewhere on your page.

Add a `script` tag somewhere on your page so we can start writing some JavaScript.

First, add a link to the bottom of the "tags" list via JavaScript. Second, bind to the "click" event of that link so we can add a new tag form (`addTagForm` will be show next):

Listing 21-11

```
// Get the div that holds the collection of tags
var collectionHolder = $('ul.tags');

// setup an "add a tag" link
var $addTagLink = $('<a href="#" class="add_tag_link">Add a tag</a>');
var $newLinkLi = $('<li></li>').append($addTagLink);

jQuery(document).ready(function() {
    // add the "add a tag" anchor and li to the tags ul
    collectionHolder.append($newLinkLi);

    $addTagLink.on('click', function(e) {
        // prevent the link from creating a "#" on the URL
        e.preventDefault();

        // add a new tag form (see next code block)
        addTagForm(collectionHolder, $newLinkLi);
    });
});
```

The `addTagForm` function's job will be to use the `data-prototype` attribute to dynamically add a new form when this link is clicked. The `data-prototype` HTML contains the tag `text` input element with a name of `task[tags][$$name$$][name]` and id of `task_tags_$$name$$_name`. The `$$name` is a little "placeholder", which we'll replace with a unique, incrementing number (e.g. `task[tags][3][name]`).

The actual code needed to make this all work can vary quite a bit, but here's one example:

Listing 21-12

```
function addTagForm(collectionHolder, $newLinkLi) {
    // Get the data-prototype we explained earlier
    var prototype = collectionHolder.attr('data-prototype');

    // Replace '$$name$$' in the prototype's HTML to
    // instead be a number based on the current collection's length.
    var newForm = prototype.replace(/\$\$name\$\$/g, collectionHolder.children().length);

    // Display the form in the page in an li, before the "Add a tag" link li
    var $newFormLi = $('<li></li>').append(newForm);
    $newLinkLi.before($newFormLi);
}
```

Now, each time a user clicks the `Add a tag` link, a new sub form will appear on the page. When we submit, any new tag forms will be converted into new `Tag` objects and added to the `tags` property of the `Task` object.

Doctrine: Cascading Relations and saving the "Inverse" side

To get the new tags to save in Doctrine, you need to consider a couple more things. First, unless you iterate over all of the new `Tag` objects and call `$em->persist($tag)` on each, you'll receive an error from Doctrine:

> A new entity was found through the relationship 'AcmeTaskBundleEntityTask#tags' that was not configured to cascade persist operations for entity...

To fix this, you may choose to "cascade" the persist operation automatically from the `Task` object to any related tags. To do this, add the `cascade` option to your `ManyToMany` metadata:

Listing 21-13

```
/**
 * @ORM\ManyToMany(targetEntity="Tag", cascade={"persist"})
 */
protected $tags;
```

A second potential issue deals with the *Owning Side and Inverse Side*[1] of Doctrine relationships. In this example, if the "owning" side of the relationship is "Task", then persistence will work fine as the tags are properly added to the Task. However, if the owning side is on "Tag", then you'll need to do a little bit more work to ensure that the correct side of the relationship is modified.

The trick is to make sure that the single "Task" is set on each "Tag". One easy way to do this is to add some extra logic to `setTags()`, which is called by the form framework since *by_reference* is set to `false`:

Listing 21-14

```
// src/Acme/TaskBundle/Entity/Task.php
// ...

public function setTags(ArrayCollection $tags)
{
    foreach ($tags as $tag) {
        $tag->addTask($this);
    }

    $this->tags = $tags;
}
```

Inside `Tag`, just make sure you have an `addTask` method:

Listing 21-15

```
// src/Acme/TaskBundle/Entity/Tag.php
// ...

public function addTask(Task $task)
{
    if (!$this->tasks->contains($task)) {
        $this->tasks->add($task);
    }
}
```

If you have a `OneToMany` relationship, then the workaround is similar, except that you can simply call `setTask` from inside `setTags`.

1. `http://docs.doctrine-project.org/en/latest/reference/unitofwork-associations.html`

Allowing tags to be removed

The next step is to allow the deletion of a particular item in the collection. The solution is similar to allowing tags to be added.

Start by adding the `allow_delete` option in the form Type:

Listing 21-16

```
// src/Acme/TaskBundle/Form/Type/TaskType.php
// ...

public function buildForm(FormBuilder $builder, array $options)
{
    $builder->add('description');

    $builder->add('tags', 'collection', array(
        'type' => new TagType(),
        'allow_add' => true,
        'allow_delete' => true,
        'by_reference' => false,
    ));
}
```

Templates Modifications

The `allow_delete` option has one consequence: if an item of a collection isn't sent on submission, the related data is removed from the collection on the server. The solution is thus to remove the form element from the DOM.

First, add a "delete this tag" link to each tag form:

Listing 21-17

```
jQuery(document).ready(function() {
    // add a delete link to all of the existing tag form li elements
    collectionHolder.find('li').each(function() {
        addTagFormDeleteLink($(this));
    });

    // ... the rest of the block from above
});

function addTagForm() {
    // ...

    // add a delete link to the new form
    addTagFormDeleteLink($newFormLi);
}
```

The `addTagFormDeleteLink` function will look something like this:

Listing 21-18

```
function addTagFormDeleteLink($tagFormLi) {
    var $removeFormA = $('<a href="#">delete this tag</a>');
    $tagFormLi.append($removeFormA);

    $removeFormA.on('click', function(e) {
        // prevent the link from creating a "#" on the URL
        e.preventDefault();

        // remove the li for the tag form
        $tagFormLi.remove();
    });
}
```

When a tag form is removed from the DOM and submitted, the removed `Tag` object will not be included in the collection passed to `setTags`. Depending on your persistence layer, this may or may not be enough to actually remove the relationship between the removed `Tag` and `Task` object.

Doctrine: Ensuring the database persistence

When removing objects in this way, you may need to do a little bit more work to ensure that the relationship between the Task and the removed Tag is properly removed.

In Doctrine, you have two side of the relationship: the owning side and the inverse side. Normally in this case you'll have a ManyToMany relation and the deleted tags will disappear and persist correctly (adding new tags also works effortlessly).

But if you have an `OneToMany` relation or a `ManyToMany` with a `mappedBy` on the Task entity (meaning Task is the "inverse" side), you'll need to do more work for the removed tags to persist correctly.

In this case, you can modify the controller to remove the relationship on the removed tag. This assumes that you have some `editAction` which is handling the "update" of your Task:

Listing 21-19

```php
// src/Acme/TaskBundle/Controller/TaskController.php
// ...

public function editAction($id, Request $request)
{
    $em = $this->getDoctrine()->getEntityManager();
    $task = $em->getRepository('AcmeTaskBundle:Task')->find($id);

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

    // Create an array of the current Tag objects in the database
    foreach ($task->getTags() as $tag) $originalTags[] = $tag;

    $editForm = $this->createForm(new TaskType(), $task);

       if ('POST' === $request->getMethod()) {
        $editForm->bindRequest($this->getRequest());

        if ($editForm->isValid()) {

            // filter $originalTags to contain tags no longer present
            foreach ($task->getTags() as $tag) {
                foreach ($originalTags as $key => $toDel) {
                    if ($toDel->getId() === $tag->getId()) {
                        unset($originalTags[$key]);
                    }
                }
            }

            // remove the relationship between the tag and the Task
            foreach ($originalTags as $tag) {
                // remove the Task from the Tag
                $tag->getTasks()->removeElement($task);

                // if it were a ManyToOne relationship, remove the relationship like this
                // $tag->setTask(null);

                $em->persist($tag);

                // if you wanted to delete the Tag entirely, you can also do that
                // $em->remove($tag);
            }

            $em->persist($task);
            $em->flush();

            // redirect back to some edit page
            return $this->redirect($this->generateUrl('task_edit', array('id' => $id)));
        }
    }

    // render some form template
}
```

As you can see, adding and removing the elements correctly can be tricky. Unless you have a ManyToMany relationship where Task is the "owning" side, you'll need to do extra work to make sure that the relationship is properly updated (whether you're adding new tags or removing existing tags) on each Tag object itself.

Chapter 22

How to Create a Custom Form Field Type

Symfony comes with a bunch of core field types available for building forms. However there are situations where we want to create a custom form field type for a specific purpose. This recipe assumes we need a field definition that holds a person's gender, based on the existing choice field. This section explains how the field is defined, how we can customize its layout and finally, how we can register it for use in our application.

Defining the Field Type

In order to create the custom field type, first we have to create the class representing the field. In our situation the class holding the field type will be called *GenderType* and the file will be stored in the default location for form fields, which is `<BundleName>\Form\Type`. Make sure the field extends *`AbstractType`*[1]:

Listing 22-1

```
# src/Acme/DemoBundle/Form/Type/GenderType.php
namespace Acme\DemoBundle\Form\Type;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilder;

class GenderType extends AbstractType
{
    public function getDefaultOptions(array $options)
    {
        return array(
            'choices' => array(
                'm' => 'Male',
                'f' => 'Female',
            )
        );
    }

    public function getParent(array $options)
    {
        return 'choice';
```

1. http://api.symfony.com/2.0/Symfony/Component/Form/AbstractType.html

```
    }

    public function getName()
    {
        return 'gender';
    }
}
```

The location of this file is not important - the `Form\Type` directory is just a convention.

Here, the return value of the `getParent` function indicates that we're extending the `choice` field type. This means that, by default, we inherit all of the logic and rendering of that field type. To see some of the logic, check out the *ChoiceType*[2] class. There are three methods that are particularly important:

- `buildForm()` - Each field type has a `buildForm` method, which is where you configure and build any field(s). Notice that this is the same method you use to setup *your* forms, and it works the same here.
- `buildView()` - This method is used to set any extra variables you'll need when rendering your field in a template. For example, in *ChoiceType*[3], a `multiple` variable is set and used in the template to set (or not set) the `multiple` attribute on the `select` field. See Creating a Template for the Field for more details.
- `getDefaultOptions()` - This defines options for your form type that can be used in `buildForm()` and `buildView()`. There are a lot of options common to all fields (see *FieldType*[4]), but you can create any others that you need here.

If you're creating a field that consists of many fields, then be sure to set your "parent" type as `form` or something that extends `form`. Also, if you need to modify the "view" of any of your child types from your parent type, use the `buildViewBottomUp()` method.

The `getName()` method returns an identifier which should be unique in your application. This is used in various places, such as when customizing how your form type will be rendered.

The goal of our field was to extend the choice type to enable selection of a gender. This is achieved by fixing the `choices` to a list of possible genders.

Creating a Template for the Field

Each field type is rendered by a template fragment, which is determined in part by the value of your `getName()` method. For more information, see *What are Form Themes?*.

In this case, since our parent field is `choice`, we don't *need* to do any work as our custom field type will automatically be rendered like a `choice` type. But for the sake of this example, let's suppose that when our field is "expanded" (i.e. radio buttons or checkboxes, instead of a select field), we want to always render it in a `ul` element. In your form theme template (see above link for details), create a `gender_widget` block to handle this:

Listing 22-2

```
{# src/Acme/DemoBundle/Resources/views/Form/fields.html.twig #}

{% block gender_widget %}
```

2. https://github.com/symfony/symfony/blob/master/src/Symfony/Component/Form/Extension/Core/Type/ChoiceType.php
3. https://github.com/symfony/symfony/blob/master/src/Symfony/Component/Form/Extension/Core/Type/ChoiceType.php
4. https://github.com/symfony/symfony/blob/master/src/Symfony/Component/Form/Extension/Core/Type/FieldType.php

```
{% spaceless %}
    {% if expanded %}
        <ul {{ block('widget_container_attributes') }}>
        {% for child in form %}
            <li>
                {{ form_widget(child) }}
                {{ form_label(child) }}
            </li>
        {% endfor %}
        </ul>
    {% else %}
        {# just let the choice widget render the select tag #}
        {{ block('choice_widget') }}
    {% endif %}
{% endspaceless %}
{% endblock %}
```

Make sure the correct widget prefix is used. In this example the name should be `gender_widget`, according to the value returned by `getName`. Further, the main config file should point to the custom form template so that it's used when rendering all forms.

Listing 22-3

```
# app/config/config.yml

twig:
    form:
        resources:
            - 'AcmeDemoBundle:Form:fields.html.twig'
```

Using the Field Type

You can now use your custom field type immediately, simply by creating a new instance of the type in one of your forms:

Listing 22-4

```
// src/Acme/DemoBundle/Form/Type/AuthorType.php
namespace Acme\DemoBundle\Form\Type;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilder;

class AuthorType extends AbstractType
{
    public function buildForm(FormBuilder $builder, array $options)
    {
        $builder->add('gender_code', new GenderType(), array(
            'empty_value' => 'Choose a gender',
        ));
    }
}
```

But this only works because the `GenderType()` is very simple. What if the gender codes were stored in configuration or in a database? The next section explains how more complex field types solve this problem.

Creating your Field Type as a Service

So far, this entry has assumed that you have a very simple custom field type. But if you need access to configuration, a database connection, or some other service, then you'll want to register your custom type as a service. For example, suppose that we're storing the gender parameters in configuration:

Listing 22-5

```yaml
# app/config/config.yml
parameters:
    genders:
        m: Male
        f: Female
```

To use the parameter, we'll define our custom field type as a service, injecting the `genders` parameter value as the first argument to its to-be-created `__construct` function:

Listing 22-6

```yaml
# src/Acme/DemoBundle/Resources/config/services.yml
services:
    form.type.gender:
        class: Acme\DemoBundle\Form\Type\GenderType
        arguments:
            - "%genders%"
        tags:
            - { name: form.type, alias: gender }
```

Make sure the services file is being imported. See *Importing Configuration with imports* for details.

Be sure that the `alias` attribute of the tag corresponds with the value returned by the `getName` method defined earlier. We'll see the importance of this in a moment when we use the custom field type. But first, add a `__construct` argument to `GenderType`, which receives the gender configuration:

Listing 22-7

```php
# src/Acme/DemoBundle/Form/Type/GenderType.php
namespace Acme\DemoBundle\Form\Type;
// ...

class GenderType extends AbstractType
{
    private $genderChoices;

    public function __construct(array $genderChoices)
    {
        $this->genderChoices = $genderChoices;
    }

    public function getDefaultOptions(array $options)
    {
        return array(
            'choices' => $this->genderChoices,
        );
    }

    // ...
}
```

Great! The `GenderType` is now fueled by the configuration parameters and registered as a service. And because we used the `form.type` alias in its configuration, using the field is now much easier:

Listing 22-8

```php
// src/Acme/DemoBundle/Form/Type/AuthorType.php
namespace Acme\DemoBundle\Form\Type;
// ...

class AuthorType extends AbstractType
{
    public function buildForm(FormBuilder $builder, array $options)
    {
        $builder->add('gender_code', 'gender', array(
            'empty_value' => 'Choose a gender',
        ));
    }
}
```

Notice that instead of instantiating a new instance, we can just refer to it by the alias used in our service configuration, `gender`. Have fun!

Chapter 23

How to use the Virtual Form Field Option

The `virtual` form field option can be very useful when you have some duplicated fields in different entities.

For example, imagine you have two entities, a `Company` and a `Customer`:

Listing 23-1

```
// src/Acme/HelloBundle/Entity/Company.php
namespace Acme\HelloBundle\Entity;

class Company
{
    private $name;
    private $website;

    private $address;
    private $zipcode;
    private $city;
    private $country;
}
```

Listing 23-2

```
// src/Acme/HelloBundle/Entity/Company.php
namespace Acme\HelloBundle\Entity;

class Customer
{
    private $firstName;
    private $lastName;

    private $address;
    private $zipcode;
    private $city;
    private $country;
}
```

Like you can see, each entity shares a few of the same fields: `address`, `zipcode`, `city`, `country`.

Now, you want to build two forms: one for a `Company` and the second for a `Customer`.

Start by creating a very simple `CompanyType` and `CustomerType`:

Listing 23-3

```php
// src/Acme/HelloBundle/Form/Type/CompanyType.php
namespace Acme\HelloBundle\Form\Type;

class CompanyType extends AbstractType
{
    public function buildForm(FormBuilder $builder, array $options)
    {
        $builder
            ->add('name', 'text')
            ->add('website', 'text')
        ;
    }
}
```

Listing 23-4

```php
// src/Acme/HelloBundle/Form/Type/CustomerType.php
namespace Acme\HelloBundle\Form\Type;

class CustomerType extends AbstractType
{
    public function buildForm(FormBuilder $builder, array $options)
    {
        $builder
            ->add('firstName', 'text')
            ->add('lastName', 'text')
        ;
    }
}
```

Now, we have to deal with the four duplicated fields. Here is a (simple) location form type:

Listing 23-5

```php
// src/Acme/HelloBundle/Form/Type/LocationType.php
namespace Acme\HelloBundle\Form\Type;

class LocationType extends AbstractType
{
    public function buildForm(FormBuilder $builder, array $options)
    {
        $builder
            ->add('address', 'textarea')
            ->add('zipcode', 'text')
            ->add('city', 'text')
            ->add('country', 'text')
        ;
    }

    public function getName()
    {
        return 'location';
    }
}
```

We don't *actually* have a location field in each of our entities, so we can't directly link our `LocationType` to our `CompanyType` or `CustomerType`. But we absolutely want to have a dedicated form type to deal with location (remember, DRY!).

The `virtual` form field option is the solution.

We can set the option `'virtual' => true` in the `getDefaultOptions` method of `LocationType` and directly start using it in the two original form types.

Look at the result:

```php
// CompanyType
public function buildForm(FormBuilder $builder, array $options)
{
    $builder->add('foo', new LocationType());
}
```

Listing 23-6

```php
// CustomerType
public function buildForm(FormBuilder $builder, array $options)
{
    $builder->add('bar', new LocationType());
}
```

Listing 23-7

With the virtual option set to false (default behavior), the Form Component expect each underlying object to have a `foo` (or `bar`) property that is either some object or array which contains the four location fields. Of course, we don't have this object/array in our entities and we don't want it!

With the virtual option set to true, the Form component skips the `foo` (or `bar`) property, and instead "gets" and "sets" the 4 location fields directly on the underlying object!

Instead of setting the `virtual` option inside `LocationType`, you can (just like with any options) also pass it in as an array option to the third argument of `$builder->add()`.

Chapter 24

How to create a Custom Validation Constraint

You can create a custom constraint by extending the base constraint class, *`Constraint`*[1]. Options for your constraint are represented as public properties on the constraint class. For example, the *Url* constraint includes the `message` and `protocols` properties:

Listing 24-1
```
namespace Symfony\Component\Validator\Constraints;

use Symfony\Component\Validator\Constraint;

/**
 * @Annotation
 */
class Protocol extends Constraint
{
    public $message = 'This value is not a valid protocol';
    public $protocols = array('http', 'https', 'ftp', 'ftps');
}
```

The `@Annotation` annotation is necessary for this new constraint in order to make it available for use in classes via annotations.

As you can see, a constraint class is fairly minimal. The actual validation is performed by a another "constraint validator" class. The constraint validator class is specified by the constraint's `validatedBy()` method, which includes some simple default logic:

Listing 24-2
```
// in the base Symfony\Component\Validator\Constraint class
public function validatedBy()
{
    return get_class($this).'Validator';
}
```

In other words, if you create a custom `Constraint` (e.g. `MyConstraint`), Symfony2 will automatically look for another class, `MyConstraintValidator` when actually performing the validation.

1. `http://api.symfony.com/2.0/Symfony/Component/Validator/Constraint.html`

The validator class is also simple, and only has one required method: `isValid`. Furthering our example, take a look at the `ProtocolValidator` as an example:

Listing 24-3

```
namespace Symfony\Component\Validator\Constraints;

use Symfony\Component\Validator\Constraint;
use Symfony\Component\Validator\ConstraintValidator;

class ProtocolValidator extends ConstraintValidator
{
    public function isValid($value, Constraint $constraint)
    {
        if (!in_array($value, $constraint->protocols)) {
            $this->setMessage($constraint->message, array('%protocols%' =>
$constraint->protocols));

            return false;
        }

        return true;
    }
}
```

Don't forget to call `setMessage` to construct an error message when the value is invalid.

Constraint Validators with Dependencies

If your constraint validator has dependencies, such as a database connection, it will need to be configured as a service in the dependency injection container. This service must include the `validator.constraint_validator` tag and an `alias` attribute:

Listing 24-4

```
services:
    validator.unique.your_validator_name:
        class: Fully\Qualified\Validator\Class\Name
        tags:
            - { name: validator.constraint_validator, alias: alias_name }
```

Your constraint class should now use this alias to reference the appropriate validator:

Listing 24-5

```
public function validatedBy()
{
    return 'alias_name';
}
```

As mentioned above, Symfony2 will automatically look for a class named after the constraint, with `Validator` appended. If your constraint validator is defined as a service, it's important that you override the `validatedBy()` method to return the alias used when defining your service, otherwise Symfony2 won't use the constraint validator service, and will instantiate the class instead, without any dependencies injected.

Class Constraint Validator

Beside validating a class property, a constraint can have a class scope by providing a target:

Listing 24-6

```
public function getTargets()
{
    return self::CLASS_CONSTRAINT;
}
```

With this, the validator `isValid()` method gets an object as its first argument:

Listing 24-7

```
class ProtocolClassValidator extends ConstraintValidator
{
    public function isValid($protocol, Constraint $constraint)
    {
        if ($protocol->getFoo() != $protocol->getBar()) {

            // bind error message on foo property
            $this->context->addViolationAtSubPath('foo', $constraint->getMessage(), array(),
null);

            return false;
        }

        return true;
    }
}
```

Chapter 25

How to Master and Create new Environments

Every application is the combination of code and a set of configuration that dictates how that code should function. The configuration may define the database being used, whether or not something should be cached, or how verbose logging should be. In Symfony2, the idea of "environments" is the idea that the same codebase can be run using multiple different configurations. For example, the `dev` environment should use configuration that makes development easy and friendly, while the `prod` environment should use a set of configuration optimized for speed.

Different Environments, Different Configuration Files

A typical Symfony2 application begins with three environments: `dev`, `prod`, and `test`. As discussed, each "environment" simply represents a way to execute the same codebase with different configuration. It should be no surprise then that each environment loads its own individual configuration file. If you're using the YAML configuration format, the following files are used:

- for the `dev` environment: `app/config/config_dev.yml`
- for the `prod` environment: `app/config/config_prod.yml`
- for the `test` environment: `app/config/config_test.yml`

This works via a simple standard that's used by default inside the `AppKernel` class:

Listing 25-1

```php
// app/AppKernel.php
// ...

class AppKernel extends Kernel
{
    // ...

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

As you can see, when Symfony2 is loaded, it uses the given environment to determine which configuration file to load. This accomplishes the goal of multiple environments in an elegant, powerful and transparent way.

Of course, in reality, each environment differs only somewhat from others. Generally, all environments will share a large base of common configuration. Opening the "dev" configuration file, you can see how this is accomplished easily and transparently:

Listing 25-2

```
imports:
    - { resource: config.yml }

# ...
```

To share common configuration, each environment's configuration file simply first imports from a central configuration file (`config.yml`). The remainder of the file can then deviate from the default configuration by overriding individual parameters. For example, by default, the `web_profiler` toolbar is disabled. However, in the `dev` environment, the toolbar is activated by modifying the default value in the `dev` configuration file:

Listing 25-3

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

web_profiler:
    toolbar: true
    # ...
```

Executing an Application in Different Environments

To execute the application in each environment, load up the application using either the `app.php` (for the `prod` environment) or the `app_dev.php` (for the `dev` environment) front controller:

Listing 25-4

```
http://localhost/app.php      -> *prod* environment
http://localhost/app_dev.php  -> *dev* environment
```

The given URLs assume that your web server is configured to use the `web/` directory of the application as its root. Read more in *Installing Symfony2*.

If you open up one of these files, you'll quickly see that the environment used by each is explicitly set:

Listing 25-5 Listing 25-6

```php
<?php

require_once __DIR__.'/../app/bootstrap_cache.php';
require_once __DIR__.'/../app/AppCache.php';

use Symfony\Component\HttpFoundation\Request;

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

As you can see, the `prod` key specifies that this environment will run in the `prod` environment. A Symfony2 application can be executed in any environment by using this code and changing the environment string.

The `test` environment is used when writing functional tests and is not accessible in the browser directly via a front controller. In other words, unlike the other environments, there is no `app_test.php` front controller file.

Debug Mode

Important, but unrelated to the topic of *environments* is the `false` key on line 8 of the front controller above. This specifies whether or not the application should run in "debug mode". Regardless of the environment, a Symfony2 application can be run with debug mode set to `true` or `false`. This affects many things in the application, such as whether or not errors should be displayed or if cache files are dynamically rebuilt on each request. Though not a requirement, debug mode is generally set to `true` for the `dev` and `test` environments and `false` for the `prod` environment.

Internally, the value of the debug mode becomes the `kernel.debug` parameter used inside the *service container*. If you look inside the application configuration file, you'll see the parameter used, for example, to turn logging on or off when using the Doctrine DBAL:

Listing 25-7

```
doctrine:
   dbal:
       logging:  "%kernel.debug%"
       # ...
```

Creating a New Environment

By default, a Symfony2 application has three environments that handle most cases. Of course, since an environment is nothing more than a string that corresponds to a set of configuration, creating a new environment is quite easy.

Suppose, for example, that before deployment, you need to benchmark your application. One way to benchmark the application is to use near-production settings, but with Symfony2's `web_profiler` enabled. This allows Symfony2 to record information about your application while benchmarking.

The best way to accomplish this is via a new environment called, for example, `benchmark`. Start by creating a new configuration file:

Listing 25-8

```
# app/config/config_benchmark.yml

imports:
    - { resource: config_prod.yml }

framework:
    profiler: { only_exceptions: false }
```

And with this simple addition, the application now supports a new environment called `benchmark`.

This new configuration file imports the configuration from the `prod` environment and modifies it. This guarantees that the new environment is identical to the `prod` environment, except for any changes explicitly made here.

Because you'll want this environment to be accessible via a browser, you should also create a front controller for it. Copy the `web/app.php` file to `web/app_benchmark.php` and edit the environment to be `benchmark`:

Listing 25-9

```
<?php

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

use Symfony\Component\HttpFoundation\Request;

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

The new environment is now accessible via:

Listing 25-10

```
http://localhost/app_benchmark.php
```

Some environments, like the `dev` environment, are never meant to be accessed on any deployed server by the general public. This is because certain environments, for debugging purposes, may give too much information about the application or underlying infrastructure. To be sure these environments aren't accessible, the front controller is usually protected from external IP addresses via the following code at the top of the controller:

Listing 25-11

```
if (!in_array(@$_SERVER['REMOTE_ADDR'], array('127.0.0.1', '::1'))) {
    die('You are not allowed to access this file. Check '.basename(__FILE__).'
for more information.');
}
```

Environments and the Cache Directory

Symfony2 takes advantage of caching in many ways: the application configuration, routing configuration, Twig templates and more are cached to PHP objects stored in files on the filesystem.

By default, these cached files are largely stored in the `app/cache` directory. However, each environment caches its own set of files:

Listing 25-12

```
app/cache/dev   - cache directory for the *dev* environment
app/cache/prod  - cache directory for the *prod* environment
```

Sometimes, when debugging, it may be helpful to inspect a cached file to understand how something is working. When doing so, remember to look in the directory of the environment you're using (most commonly `dev` while developing and debugging). While it can vary, the `app/cache/dev` directory includes the following:

- `appDevDebugProjectContainer.php` - the cached "service container" that represents the cached application configuration;
- `appdevUrlGenerator.php` - the PHP class generated from the routing configuration and used when generating URLs;
- `appdevUrlMatcher.php` - the PHP class used for route matching - look here to see the compiled regular expression logic used to match incoming URLs to different routes;
- `twig/` - this directory contains all the cached Twig templates.

Going Further

Read the article on *How to Set External Parameters in the Service Container*.

Chapter 26

How to Set External Parameters in the Service Container

In the chapter *How to Master and Create new Environments*, you learned how to manage your application configuration. At times, it may benefit your application to store certain credentials outside of your project code. Database configuration is one such example. The flexibility of the symfony service container allows you to easily do this.

Environment Variables

Symfony will grab any environment variable prefixed with `SYMFONY__` and set it as a parameter in the service container. Double underscores are replaced with a period, as a period is not a valid character in an environment variable name.

For example, if you're using Apache, environment variables can be set using the following `VirtualHost` configuration:

Listing 26-1

```
<VirtualHost *:80>
    ServerName      Symfony2
    DocumentRoot    "/path/to/symfony_2_app/web"
    DirectoryIndex  index.php index.html
    SetEnv          SYMFONY__DATABASE__USER user
    SetEnv          SYMFONY__DATABASE__PASSWORD secret

    <Directory "/path/to/symfony_2_app/web">
        AllowOverride All
        Allow from All
    </Directory>
</VirtualHost>
```

The example above is for an Apache configuration, using the *SetEnv*[1] directive. However, this will work for any web server which supports the setting of environment variables.

Also, in order for your console to work (which does not use Apache), you must export these as shell variables. On a Unix system, you can run the following:

Listing 26-2
```
export SYMFONY__DATABASE__USER=user
export SYMFONY__DATABASE__PASSWORD=secret
```

Now that you have declared an environment variable, it will be present in the PHP `$_SERVER` global variable. Symfony then automatically sets all `$_SERVER` variables prefixed with `SYMFONY__` as parameters in the service container.

You can now reference these parameters wherever you need them.

Listing 26-3
```
doctrine:
    dbal:
        driver    pdo_mysql
        dbname:   symfony2_project
        user:     %database.user%
        password: %database.password%
```

Constants

The container also has support for setting PHP constants as parameters. To take advantage of this feature, map the name of your constant to a parameter key, and define the type as `constant`.

Listing 26-4
```
<?xml version="1.0" encoding="UTF-8"?>

<container xmlns="http://symfony.com/schema/dic/services"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
>

    <parameters>
        <parameter key="global.constant.value"
type="constant">GLOBAL_CONSTANT</parameter>
        <parameter key="my_class.constant.value"
type="constant">My_Class::CONSTANT_NAME</parameter>
    </parameters>
</container>
```

This only works for XML configuration. If you're *not* using XML, simply import an XML file to take advantage of this functionality:

Listing 26-5
```
// app/config/config.yml
imports:
    - { resource: parameters.xml }
```

Miscellaneous Configuration

The `imports` directive can be used to pull in parameters stored elsewhere. Importing a PHP file gives you the flexibility to add whatever is needed in the container. The following imports a file named `parameters.php`.

1. http://httpd.apache.org/docs/current/env.html

```yaml
# app/config/config.yml
imports:
    - { resource: parameters.php }
```
Listing 26-6

A resource file can be one of many types. PHP, XML, YAML, INI, and closure resources are all supported by the `imports` directive.

In `parameters.php`, tell the service container the parameters that you wish to set. This is useful when important configuration is in a nonstandard format. The example below includes a Drupal database's configuration in the symfony service container.

```php
// app/config/parameters.php

include_once('/path/to/drupal/sites/default/settings.php');
$container->setParameter('drupal.database.url', $db_url);
```
Listing 26-7

Chapter 27

How to use PdoSessionStorage to store Sessions in the Database

The default session storage of Symfony2 writes the session information to file(s). Most medium to large websites use a database to store the session values instead of files, because databases are easier to use and scale in a multi-webserver environment.

Symfony2 has a built-in solution for database session storage called *PdoSessionStorage*[1]. To use it, you just need to change some parameters in `config.yml` (or the configuration format of your choice):

Listing 27-1

```
# app/config/config.yml
framework:
    session:
        # ...
        storage_id:     session.storage.pdo

parameters:
    pdo.db_options:
        db_table:    session
        db_id_col:   session_id
        db_data_col: session_value
        db_time_col: session_time

services:
    pdo:
        class: PDO
        arguments:
            dsn:      "mysql:dbname=mydatabase"
            user:     myuser
            password: mypassword

    session.storage.pdo:
        class:     Symfony\Component\HttpFoundation\SessionStorage\PdoSessionStorage
        arguments: [@pdo, %session.storage.options%, %pdo.db_options%]
```

1. http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/SessionStorage/PdoSessionStorage.html

- `db_table`: The name of the session table in your database
- `db_id_col`: The name of the id column in your session table (VARCHAR(255) or larger)
- `db_data_col`: The name of the value column in your session table (TEXT or CLOB)
- `db_time_col`: The name of the time column in your session table (INTEGER)

Sharing your Database Connection Information

With the given configuration, the database connection settings are defined for the session storage connection only. This is OK when you use a separate database for the session data.

But if you'd like to store the session data in the same database as the rest of your project's data, you can use the connection settings from the parameter.ini by referencing the database-related parameters defined there:

Listing 27-2

```
pdo:
    class: PDO
    arguments:
        - "mysql:dbname=%database_name%"
        - %database_user%
        - %database_password%
```

Example SQL Statements

MySQL

The SQL statement for creating the needed database table might look like the following (MySQL):

Listing 27-3

```
CREATE TABLE `session` (
    `session_id` varchar(255) NOT NULL,
    `session_value` text NOT NULL,
    `session_time` int(11) NOT NULL,
    PRIMARY KEY (`session_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

PostgreSQL

For PostgreSQL, the statement should look like this:

Listing 27-4

```
CREATE TABLE session (
    session_id character varying(255) NOT NULL,
    session_value text NOT NULL,
    session_time integer NOT NULL,
    CONSTRAINT session_pkey PRIMARY KEY (session_id)
);
```

Chapter 28

How to create an Event Listener

Symfony has various events and hooks that can be used to trigger custom behavior in your application. Those events are thrown by the HttpKernel component and can be viewed in the *KernelEvents*[1] class.

To hook into an event and add your own custom logic, you have to create a service that will act as an event listener on that event. In this entry, we will create a service that will act as an Exception Listener, allowing us to modify how exceptions are shown by our application. The `KernelEvents::EXCEPTION` event is just one of the core kernel events:

Listing 28-1

```php
// src/Acme/DemoBundle/Listener/AcmeExceptionListener.php
namespace Acme\DemoBundle\Listener;

use Symfony\Component\HttpKernel\Event\GetResponseForExceptionEvent;

class AcmeExceptionListener
{
    public function onKernelException(GetResponseForExceptionEvent $event)
    {
        // We get the exception object from the received event
        $exception = $event->getException();
        $message = 'My Error says: ' . $exception->getMessage();

        // Customize our response object to display our exception details
        $response->setContent($message);
        $response->setStatusCode($exception->getStatusCode());

        // Send our modified response object to the event
        $event->setResponse($response);
    }
}
```

Each event receives a slightly different type of `$event` object. For the `kernel.exception` event, it is *GetResponseForExceptionEvent*[2]. To see what type of object each event listener receives, see *KernelEvents*[3],

1. http://api.symfony.com/2.0/Symfony/Component/HttpKernel/KernelEvents.html

Now that the class is created, we just need to register it as a service and and notify Symfony that it is a "listener" on the `kernel.exception` event by using a special "tag":

Listing 28-2

```
services:
    kernel.listener.your_listener_name:
        class: Acme\DemoBundle\Listener\AcmeExceptionListener
        tags:
            - { name: kernel.event_listener, event: kernel.exception, method:
onKernelException }
```

There is an additional tag option `priority` that is optional and defaults to 0. This value can be from -255 to 255, and the listeners will be executed in the order of their priority. This is useful when you need to guarantee that one listener is executed before another.

2. http://api.symfony.com/2.0/Symfony/Component/HttpKernel/Event/GetResponseForExceptionEvent.html
3. http://api.symfony.com/2.0/Symfony/Component/HttpKernel/KernelEvents.html

Chapter 29

How to Use a Factory to Create Services

Symfony2's Service Container provides a powerful way of controlling the creation of objects, allowing you to specify arguments passed to the constructor as well as calling methods and setting parameters. Sometimes, however, this will not provide you with everything you need to construct your objects. For this situation, you can use a factory to create the object and tell the service container to call a method on the factory rather than directly instantiating the object.

Suppose you have a factory that configures and returns a new NewsletterManager object:

Listing 29-1

```
namespace Acme\HelloBundle\Newsletter;

class NewsletterFactory
{
    public function get()
    {
        $newsletterManager = new NewsletterManager();

        // ...

        return $newsletterManager;
    }
}
```

To make the `NewsletterManager` object available as a service, you can configure the service container to use the `NewsletterFactory` factory class:

Listing 29-2

```
# src/Acme/HelloBundle/Resources/config/services.yml
parameters:
    # ...
    newsletter_manager.class: Acme\HelloBundle\Newsletter\NewsletterManager
    newsletter_factory.class: Acme\HelloBundle\Newsletter\NewsletterFactory
services:
    newsletter_manager:
        class:          %newsletter_manager.class%
        factory_class:  %newsletter_factory.class%
        factory_method: get
```

When you specify the class to use for the factory (via `factory_class`) the method will be called statically. If the factory itself should be instantiated and the resulting object's method called (as in this example), configure the factory itself as a service:

Listing 29-3

```
# src/Acme/HelloBundle/Resources/config/services.yml
parameters:
    # ...
    newsletter_manager.class: Acme\HelloBundle\Newsletter\NewsletterManager
    newsletter_factory.class: Acme\HelloBundle\Newsletter\NewsletterFactory
services:
    newsletter_factory:
        class:            %newsletter_factory.class%
    newsletter_manager:
        class:            %newsletter_manager.class%
        factory_service:  newsletter_factory
        factory_method:   get
```

The factory service is specified by its id name and not a reference to the service itself. So, you do not need to use the @ syntax.

Passing Arguments to the Factory Method

If you need to pass arguments to the factory method, you can use the `arguments` options inside the service container. For example, suppose the `get` method in the previous example takes the `templating` service as an argument:

Listing 29-4

```
# src/Acme/HelloBundle/Resources/config/services.yml
parameters:
    # ...
    newsletter_manager.class: Acme\HelloBundle\Newsletter\NewsletterManager
    newsletter_factory.class: Acme\HelloBundle\Newsletter\NewsletterFactory
services:
    newsletter_factory:
        class:            %newsletter_factory.class%
    newsletter_manager:
        class:            %newsletter_manager.class%
        factory_service:  newsletter_factory
        factory_method:   get
        arguments:
            -             @templating
```

Chapter 30

How to Manage Common Dependencies with Parent Services

As you add more functionality to your application, you may well start to have related classes that share some of the same dependencies. For example you may have a Newsletter Manager which uses setter injection to set its dependencies:

Listing 30-1

```php
namespace Acme\HelloBundle\Mail;

use Acme\HelloBundle\Mailer;
use Acme\HelloBundle\EmailFormatter;

class NewsletterManager
{
    protected $mailer;
    protected $emailFormatter;

    public function setMailer(Mailer $mailer)
    {
        $this->mailer = $mailer;
    }

    public function setEmailFormatter(EmailFormatter $emailFormatter)
    {
        $this->emailFormatter = $emailFormatter;
    }
    // ...
}
```

and also a Greeting Card class which shares the same dependencies:

Listing 30-2

```php
namespace Acme\HelloBundle\Mail;

use Acme\HelloBundle\Mailer;
use Acme\HelloBundle\EmailFormatter;

class GreetingCardManager
```

```
{
    protected $mailer;
    protected $emailFormatter;

    public function setMailer(Mailer $mailer)
    {
        $this->mailer = $mailer;
    }

    public function setEmailFormatter(EmailFormatter $emailFormatter)
    {
        $this->emailFormatter = $emailFormatter;
    }
    // ...
}
```

The service config for these classes would look something like this:

Listing 30-3

```
# src/Acme/HelloBundle/Resources/config/services.yml
parameters:
    # ...
    newsletter_manager.class: Acme\HelloBundle\Mail\NewsletterManager
    greeting_card_manager.class: Acme\HelloBundle\Mail\GreetingCardManager
services:
    my_mailer:
        # ...
    my_email_formatter:
        # ...
    newsletter_manager:
        class:     %newsletter_manager.class%
        calls:
            - [ setMailer, [ @my_mailer ] ]
            - [ setEmailFormatter, [ @my_email_formatter] ]

    greeting_card_manager:
        class:     %greeting_card_manager.class%
        calls:
            - [ setMailer, [ @my_mailer ] ]
            - [ setEmailFormatter, [ @my_email_formatter] ]
```

There is a lot of repetition in both the classes and the configuration. This means that if you changed, for example, the `Mailer` of `EmailFormatter` classes to be injected via the constructor, you would need to update the config in two places. Likewise if you needed to make changes to the setter methods you would need to do this in both classes. The typical way to deal with the common methods of these related classes would be to extract them to a super class:

Listing 30-4

```
namespace Acme\HelloBundle\Mail;

use Acme\HelloBundle\Mailer;
use Acme\HelloBundle\EmailFormatter;

abstract class MailManager
{
    protected $mailer;
    protected $emailFormatter;

    public function setMailer(Mailer $mailer)
    {
        $this->mailer = $mailer;
    }
    public function setEmailFormatter(EmailFormatter $emailFormatter)
    {
        $this->emailFormatter = $emailFormatter;
    }
    // ...
}
```

The `NewsletterManager` and `GreetingCardManager` can then extend this super class:

Listing 30-5

```
namespace Acme\HelloBundle\Mail;

class NewsletterManager extends MailManager
{
    // ...
}
```

and:

Listing 30-6

```
namespace Acme\HelloBundle\Mail;

class GreetingCardManager extends MailManager
{
    // ...
}
```

In a similar fashion, the Symfony2 service container also supports extending services in the configuration so you can also reduce the repetition by specifying a parent for a service.

Listing 30-7

```
# src/Acme/HelloBundle/Resources/config/services.yml
parameters:
    # ...
    newsletter_manager.class: Acme\HelloBundle\Mail\NewsletterManager
    greeting_card_manager.class: Acme\HelloBundle\Mail\GreetingCardManager
    mail_manager.class: Acme\HelloBundle\Mail\MailManager
services:
    my_mailer:
        # ...
    my_email_formatter:
        # ...
    mail_manager:
        class:     %mail_manager.class%
        abstract:  true
        calls:
            - [ setMailer, [ @my_mailer ] ]
            - [ setEmailFormatter, [ @my_email_formatter] ]

    newsletter_manager:
        class:     %newsletter_manager.class%
        parent: mail_manager

    greeting_card_manager:
        class:     %greeting_card_manager.class%
        parent: mail_manager
```

In this context, having a `parent` service implies that the arguments and method calls of the parent service should be used for the child services. Specifically, the setter methods defined for the parent service will be called when the child services are instantiated.

If you remove the `parent` config key, the services will still be instantiated and they will still of course extend the `MailManager` class. The difference is that omitting the `parent` config key will mean that the `calls` defined on the `mail_manager` service will not be executed when the child services are instantiated.

The parent class is abstract as it should not be directly instantiated. Setting it to abstract in the config file as has been done above will mean that it can only be used as a parent service and cannot be used directly as a service to inject and will be removed at compile time. In other words, it exists merely as a "template" that other services can use.

Overriding Parent Dependencies

There may be times where you want to override what class is passed in for a dependency of one child service only. Fortunately, by adding the method call config for the child service, the dependencies set by the parent class will be overridden. So if you needed to pass a different dependency just to the `NewsletterManager` class, the config would look like this:

Listing 30-8

```
# src/Acme/HelloBundle/Resources/config/services.yml
parameters:
    # ...
    newsletter_manager.class: Acme\HelloBundle\Mail\NewsletterManager
    greeting_card_manager.class: Acme\HelloBundle\Mail\GreetingCardManager
    mail_manager.class: Acme\HelloBundle\Mail\MailManager
services:
    my_mailer:
        # ...
    my_alternative_mailer:
        # ...
    my_email_formatter:
        # ...
    mail_manager:
        class:     %mail_manager.class%
        abstract:  true
        calls:
            - [ setMailer, [ @my_mailer ] ]
            - [ setEmailFormatter, [ @my_email_formatter] ]

    newsletter_manager:
        class:     %newsletter_manager.class%
        parent: mail_manager
        calls:
            - [ setMailer, [ @my_alternative_mailer ] ]

    greeting_card_manager:
        class:     %greeting_card_manager.class%
        parent: mail_manager
```

The `GreetingCardManager` will receive the same dependencies as before, but the `NewsletterManager` will be passed the `my_alternative_mailer` instead of the `my_mailer` service.

Collections of Dependencies

It should be noted that the overridden setter method in the previous example is actually called twice - once per the parent definition and once per the child definition. In the previous example, that was fine, since the second `setMailer` call replaces mailer object set by the first call.

In some cases, however, this can be a problem. For example, if the overridden method call involves adding something to a collection, then two objects will be added to that collection. The following shows such a case, if the parent class looks like this:

Listing 30-9

```php
namespace Acme\HelloBundle\Mail;

use Acme\HelloBundle\Mailer;
use Acme\HelloBundle\EmailFormatter;

abstract class MailManager
{
    protected $filters;

    public function setFilter($filter)
    {
        $this->filters[] = $filter;
    }
    // ...
}
```

If you had the following config:

Listing 30-10

```yaml
# src/Acme/HelloBundle/Resources/config/services.yml
parameters:
    # ...
    newsletter_manager.class: Acme\HelloBundle\Mail\NewsletterManager
    mail_manager.class: Acme\HelloBundle\Mail\MailManager
services:
    my_filter:
        # ...
    another_filter:
        # ...
    mail_manager:
        class:     %mail_manager.class%
        abstract:  true
        calls:
            - [ setFilter, [ @my_filter ] ]

    newsletter_manager:
        class:     %newsletter_manager.class%
        parent: mail_manager
        calls:
            - [ setFilter, [ @another_filter ] ]
```

In this example, the `setFilter` of the `newsletter_manager` service will be called twice, resulting in the `$filters` array containing both `my_filter` and `another_filter` objects. This is great if you just want to add additional filters to the subclasses. If you want to replace the filters passed to the subclass, removing the parent setting from the config will prevent the base class from calling to `setFilter`.

Chapter 31

How to work with Scopes

This entry is all about scopes, a somewhat advanced topic related to the *Service Container*. If you've ever gotten an error mentioning "scopes" when creating services, or need to create a service that depends on the *request* service, then this entry is for you.

Understanding Scopes

The scope of a service controls how long an instance of a service is used by the container. The Dependency Injection component provides two generic scopes:

- *container* (the default one): The same instance is used each time you request it from this container.
- *prototype*: A new instance is created each time you request the service.

The FrameworkBundle also defines a third scope: *request*. This scope is tied to the request, meaning a new instance is created for each subrequest and is unavailable outside the request (for instance in the CLI).

Scopes add a constraint on the dependencies of a service: a service cannot depend on services from a narrower scope. For example, if you create a generic *my_foo* service, but try to inject the *request* component, you'll receive a `ScopeWideningInjectionException`[1] when compiling the container. Read the sidebar below for more details.

1. http://api.symfony.com/2.0/Symfony/Component/DependencyInjection/Exception/ScopeWideningInjectionException.html

Scopes and Dependencies

Imagine you've configured a *my_mailer* service. You haven't configured the scope of the service, so it defaults to *container*. In other words, everytime you ask the container for the *my_mailer* service, you get the same object back. This is usually how you want your services to work.

Imagine, however, that you need the *request* service in your *my_mailer* service, maybe because you're reading the URL of the current request. So, you add it as a constructor argument. Let's look at why this presents a problem:

- When requesting *my_mailer*, an instance of *my_mailer* (let's call it *MailerA*) is created and the *request* service (let's call it *RequestA*) is passed to it. Life is good!
- You've now made a subrequest in Symfony, which is a fancy way of saying that you've called, for example, the *{% render ... %}* Twig function, which executes another controller. Internally, the old *request* service (*RequestA*) is actually replaced by a new request instance (*RequestB*). This happens in the background, and it's totally normal.
- In your embedded controller, you once again ask for the *my_mailer* service. Since your service is in the *container* scope, the same instance (*MailerA*) is just re-used. But here's the problem: the *MailerA* instance still contains the old *RequestA* object, which is now **not** the correct request object to have (*RequestB* is now the current *request* service). This is subtle, but the mis-match could cause major problems, which is why it's not allowed.

  So, that's the reason *why* scopes exist, and how they can cause problems. Keep reading to find out the common solutions.

A service can of course depend on a service from a wider scope without any issue.

Setting the Scope in the Definition

The scope of a service is defined in the definition of the service:

Listing 31-1

```yaml
# src/Acme/HelloBundle/Resources/config/services.yml
services:
    greeting_card_manager:
        class: Acme\HelloBundle\Mail\GreetingCardManager
        scope: request
```

If you don't specify the scope, it defaults to *container*, which is what you want most of the time. Unless your service depends on another service that's scoped to a narrower scope (most commonly, the *request* service), you probably don't need to set the scope.

Using a Service from a narrower Scope

If your service depends on a scoped service, the best solution is to put it in the same scope (or a narrower one). Usually, this means putting your new service in the *request* scope.

But this is not always possible (for instance, a twig extension must be in the *container* scope as the Twig environment needs it as a dependency). In these cases, you should pass the entire container into your service and retrieve your dependency from the container each time we need it to be sure you have the right instance:

Listing 31-2

```php
namespace Acme\HelloBundle\Mail;

use Symfony\Component\DependencyInjection\ContainerInterface;

class Mailer
{
    protected $container;

    public function __construct(ContainerInterface $container)
    {
        $this->container = $container;
    }

    public function sendEmail()
    {
        $request = $this->container->get('request');
        // Do something using the request here
    }
}
```

Take care not to store the request in a property of the object for a future call of the service as it would be the same issue described in the first section (except that symfony cannot detect that you are wrong).

The service config for this class would look something like this:

Listing 31-3

```yaml
# src/Acme/HelloBundle/Resources/config/services.yml
parameters:
    # ...
    my_mailer.class: Acme\HelloBundle\Mail\Mailer
services:
    my_mailer:
        class:     %my_mailer.class%
        arguments:
            - "@service_container"
        # scope: container can be omitted as it is the default
```

Injecting the whole container into a service is generally not a good idea (only inject what you need). In some rare cases, it's necessary when you have a service in the `container` scope that needs a service in the `request` scope.

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 as a Controller Argument* for details.

Chapter 32

How to make your Services use Tags

Several of Symfony2's core services depend on tags to recognize which services should be loaded, notified of events, or handled in some other special way. For example, Twig uses the tag `twig.extension` to load extra extensions.

But you can also use tags in your own bundles. For example in case your service handles a collection of some kind, or implements a "chain", in which several alternative strategies are tried until one of them is successful. In this article I will use the example of a "transport chain", which is a collection of classes implementing `\Swift_Transport`. Using the chain, the Swift mailer may try several ways of transport, until one succeeds. This post focuses mainly on the dependency injection part of the story.

To begin with, define the `TransportChain` class:

Listing 32-1

```
namespace Acme\MailerBundle;

class TransportChain
{
    private $transports;

    public function __construct()
    {
        $this->transports = array();
    }

    public function addTransport(\Swift_Transport  $transport)
    {
        $this->transports[] = $transport;
    }
}
```

Then, define the chain as a service:

Listing 32-2

```
# src/Acme/MailerBundle/Resources/config/services.yml
parameters:
    acme_mailer.transport_chain.class: Acme\MailerBundle\TransportChain

services:
    acme_mailer.transport_chain:
        class: %acme_mailer.transport_chain.class%
```

Define Services with a Custom Tag

Now we want several of the `\Swift_Transport` classes to be instantiated and added to the chain automatically using the `addTransport()` method. As an example we add the following transports as services:

Listing 32-3

```
# src/Acme/MailerBundle/Resources/config/services.yml
services:
    acme_mailer.transport.smtp:
        class: \Swift_SmtpTransport
        arguments:
            - %mailer_host%
        tags:
            -  { name: acme_mailer.transport }
    acme_mailer.transport.sendmail:
        class: \Swift_SendmailTransport
        tags:
            -  { name: acme_mailer.transport }
```

Notice the tags named "acme_mailer.transport". We want the bundle to recognize these transports and add them to the chain all by itself. In order to achieve this, we need to add a `build()` method to the `AcmeMailerBundle` class:

Listing 32-4

```
namespace Acme\MailerBundle;

use Symfony\Component\HttpKernel\Bundle\Bundle;
use Symfony\Component\DependencyInjection\ContainerBuilder;

use Acme\MailerBundle\DependencyInjection\Compiler\TransportCompilerPass;

class AcmeMailerBundle extends Bundle
{
    public function build(ContainerBuilder $container)
    {
        parent::build($container);

        $container->addCompilerPass(new TransportCompilerPass());
    }
}
```

Create a `CompilerPass`

You will have spotted a reference to the not yet existing `TransportCompilerPass` class. This class will make sure that all services with a tag `acme_mailer.transport` will be added to the `TransportChain` class by calling the `addTransport()` method. The `TransportCompilerPass` should look like this:

Listing 32-5

```
namespace Acme\MailerBundle\DependencyInjection\Compiler;

use Symfony\Component\DependencyInjection\ContainerBuilder;
use Symfony\Component\DependencyInjection\Compiler\CompilerPassInterface;
use Symfony\Component\DependencyInjection\Reference;

class TransportCompilerPass implements CompilerPassInterface
{
    public function process(ContainerBuilder $container)
    {
        if (false === $container->hasDefinition('acme_mailer.transport_chain')) {
            return;
        }

        $definition = $container->getDefinition('acme_mailer.transport_chain');

        foreach ($container->findTaggedServiceIds('acme_mailer.transport') as $id =>
$attributes) {
            $definition->addMethodCall('addTransport', array(new Reference($id)));
        }
    }
}
```

The `process()` method checks for the existence of the `acme_mailer.transport_chain` service, then looks for all services tagged `acme_mailer.transport`. It adds to the definition of the `acme_mailer.transport_chain` service a call to `addTransport()` for each "acme_mailer.transport" service it has found. The first argument of each of these calls will be the mailer transport service itself.

By convention, tag names consist of the name of the bundle (lowercase, underscores as separators), followed by a dot, and finally the "real" name, so the tag "transport" in the AcmeMailerBundle should be: `acme_mailer.transport`.

The Compiled Service Definition

Adding the compiler pass will result in the automatic generation of the following lines of code in the compiled service container. In case you are working in the "dev" environment, open the file `/cache/dev/appDevDebugProjectContainer.php` and look for the method `getTransportChainService()`. It should look like this:

Listing 32-6

```
protected function getAcmeMailer_TransportChainService()
{
    $this->services['acme_mailer.transport_chain'] = $instance = new
\Acme\MailerBundle\TransportChain();

    $instance->addTransport($this->get('acme_mailer.transport.smtp'));
    $instance->addTransport($this->get('acme_mailer.transport.sendmail'));

    return $instance;
}
```

Chapter 33
Bundle Structure and Best Practices

A bundle is a directory that has a well-defined structure and can host anything from classes to controllers and web resources. Even if bundles are very flexible, you should follow some best practices if you want to distribute them.

Bundle Name

A bundle is also a PHP namespace. The namespace must follow the technical interoperability *standards*[1] for PHP 5.3 namespaces and class names: it starts with a vendor segment, followed by zero or more category segments, and it ends with the namespace short name, which must end with a `Bundle` suffix.

A namespace becomes a bundle as soon as you add a bundle class to it. The bundle class name must follow these simple rules:

- Use only alphanumeric characters and underscores;
- Use a CamelCased name;
- Use a descriptive and short name (no more than 2 words);
- Prefix the name with the concatenation of the vendor (and optionally the category namespaces);
- Suffix the name with `Bundle`.

Here are some valid bundle namespaces and class names:

Namespace	**Bundle Class Name**
`Acme\Bundle\BlogBundle`	`AcmeBlogBundle`
`Acme\Bundle\Social\BlogBundle`	`AcmeSocialBlogBundle`
`Acme\BlogBundle`	`AcmeBlogBundle`

By convention, the `getName()` method of the bundle class should return the class name.

1. `http://symfony.com/PSR0`

If you share your bundle publicly, you must use the bundle class name as the name of the repository (`AcmeBlogBundle` and not `BlogBundle` for instance).

Symfony2 core Bundles do not prefix the Bundle class with `Symfony` and always add a `Bundle` subnamespace; for example: *FrameworkBundle*[2].

Each bundle has an alias, which is the lower-cased short version of the bundle name using underscores (`acme_hello` for `AcmeHelloBundle`, or `acme_social_blog` for `Acme\Social\BlogBundle` for instance). This alias is used to enforce uniqueness within a bundle (see below for some usage examples).

Directory Structure

The basic directory structure of a `HelloBundle` bundle must read as follows:

Listing 33-1

```
XXX/...
    HelloBundle/
        HelloBundle.php
        Controller/
        Resources/
            meta/
                LICENSE
            config/
            doc/
                index.rst
            translations/
            views/
            public/
        Tests/
```

The `XXX` directory(ies) reflects the namespace structure of the bundle.

The following files are mandatory:

- `HelloBundle.php`;
- `Resources/meta/LICENSE`: The full license for the code;
- `Resources/doc/index.rst`: The root file for the Bundle documentation.

These conventions ensure that automated tools can rely on this default structure to work.

The depth of sub-directories should be kept to the minimal for most used classes and files (2 levels at a maximum). More levels can be defined for non-strategic, less-used files.

The bundle directory is read-only. If you need to write temporary files, store them under the `cache/` or `log/` directory of the host application. Tools can generate files in the bundle directory structure, but only if the generated files are going to be part of the repository.

The following classes and files have specific emplacements:

2. `http://api.symfony.com/2.0/Symfony/Bundle/FrameworkBundle/FrameworkBundle.html`

Type	**Directory**
Commands	`Command/`
Controllers	`Controller/`
Service Container Extensions	`DependencyInjection/`
Event Listeners	`EventListener/`
Configuration	`Resources/config/`
Web Resources	`Resources/public/`
Translation files	`Resources/translations/`
Templates	`Resources/views/`
Unit and Functional Tests	`Tests/`

Classes

The bundle directory structure is used as the namespace hierarchy. For instance, a `HelloController` controller is stored in `Bundle/HelloBundle/Controller/HelloController.php` and the fully qualified class name is `Bundle\HelloBundle\Controller\HelloController`.

All classes and files must follow the Symfony2 coding *standards*.

Some classes should be seen as facades and should be as short as possible, like Commands, Helpers, Listeners, and Controllers.

Classes that connect to the Event Dispatcher should be suffixed with `Listener`.

Exceptions classes should be stored in an `Exception` sub-namespace.

Vendors

A bundle must not embed third-party PHP libraries. It should rely on the standard Symfony2 autoloading instead.

A bundle should not embed third-party libraries written in JavaScript, CSS, or any other language.

Tests

A bundle should come with a test suite written with PHPUnit and stored under the `Tests/` directory. Tests should follow the following principles:

- The test suite must be executable with a simple `phpunit` command run from a sample application;
- The functional tests should only be used to test the response output and some profiling information if you have some;
- The code coverage should at least covers 95% of the code base.

A test suite must not contain `AllTests.php` scripts, but must rely on the existence of a `phpunit.xml.dist` file.

Documentation

All classes and functions must come with full PHPDoc.

Extensive documentation should also be provided in the *reStructuredText* format, under the `Resources/doc/` directory; the `Resources/doc/index.rst` file is the only mandatory file and must be the entry point for the documentation.

Controllers

As a best practice, controllers in a bundle that's meant to be distributed to others must not extend the *`Controller`*[3] base class. They can implement *`ContainerAwareInterface`*[4] or extend *`ContainerAware`*[5] instead.

If you have a look at *`Controller`*[6] methods, you will see that they are only nice shortcuts to ease the learning curve.

Routing

If the bundle provides routes, they must be prefixed with the bundle alias. For an AcmeBlogBundle for instance, all routes must be prefixed with `acme_blog_`.

Templates

If a bundle provides templates, they must use Twig. A bundle must not provide a main layout, except if it provides a full working application.

Translation Files

If a bundle provides message translations, they must be defined in the XLIFF format; the domain should be named after the bundle name (`bundle.hello`).

A bundle must not override existing messages from another bundle.

Configuration

To provide more flexibility, a bundle can provide configurable settings by using the Symfony2 built-in mechanisms.

For simple configuration settings, rely on the default `parameters` entry of the Symfony2 configuration. Symfony2 parameters are simple key/value pairs; a value being any valid PHP value. Each parameter name should start with the bundle alias, though this is just a best-practice suggestion. The rest of the parameter name will use a period (`.`) to separate different parts (e.g. `acme_hello.email.from`).

3. http://api.symfony.com/2.0/Symfony/Bundle/FrameworkBundle/Controller/Controller.html
4. http://api.symfony.com/2.0/Symfony/Component/DependencyInjection/ContainerAwareInterface.html
5. http://api.symfony.com/2.0/Symfony/Component/DependencyInjection/ContainerAware.html
6. http://api.symfony.com/2.0/Symfony/Bundle/FrameworkBundle/Controller/Controller.html

The end user can provide values in any configuration file:

```yaml
# app/config/config.yml
parameters:
    acme_hello.email.from: fabien@example.com
```
Listing 33-2

Retrieve the configuration parameters in your code from the container:

```php
$container->getParameter('acme_hello.email.from');
```
Listing 33-3

Even if this mechanism is simple enough, you are highly encouraged to use the semantic configuration described in the cookbook.

If you are defining services, they should also be prefixed with the bundle alias.

Learn more from the Cookbook

- *How to expose a Semantic Configuration for a Bundle*

Chapter 34
How to use Bundle Inheritance to Override parts of a Bundle

When working with third-party bundles, you'll probably come across a situation where you want to override a file in that third-party bundle with a file in one of your own bundles. Symfony gives you a very convenient way to override things like controllers, templates, and other files in a bundle's `Resources/` directory.

For example, suppose that you're installing the *FOSUserBundle*[1], but you want to override its base `layout.html.twig` template, as well as one of its controllers. Suppose also that you have your own `AcmeUserBundle` where you want the overridden files to live. Start by registering the `FOSUserBundle` as the "parent" of your bundle:

Listing 34-1

```php
// src/Acme/UserBundle/AcmeUserBundle.php
namespace Acme\UserBundle;

use Symfony\Component\HttpKernel\Bundle\Bundle;

class AcmeUserBundle extends Bundle
{
    public function getParent()
    {
        return 'FOSUserBundle';
    }
}
```

By making this simple change, you can now override several parts of the `FOSUserBundle` simply by creating a file with the same name.

1. `https://github.com/friendsofsymfony/fosuserbundle`

Overriding Controllers

Suppose you want to add some functionality to the `registerAction` of a `RegistrationController` that lives inside `FOSUserBundle`. To do so, just create your own `RegistrationController.php` file, override the bundle's original method, and change its functionality:

Listing 34-2

```php
// src/Acme/UserBundle/Controller/RegistrationController.php
namespace Acme\UserBundle\Controller;

use FOS\UserBundle\Controller\RegistrationController as BaseController;

class RegistrationController extends BaseController
{
    public function registerAction()
    {
        $response = parent::registerAction();

        // do custom stuff

        return $response;
    }
}
```

Depending on how severely you need to change the behavior, you might call `parent::registerAction()` or completely replace its logic with your own.

Overriding controllers in this way only works if the bundle refers to the controller using the standard `FOSUserBundle:Registration:register` syntax in routes and templates. This is the best practice.

Overriding Resources: Templates, Routing, Validation, etc

Most resources can also be overridden, simply by creating a file in the same location as your parent bundle.

For example, it's very common to need to override the `FOSUserBundle`'s `layout.html.twig` template so that it uses your application's base layout. Since the file lives at `Resources/views/layout.html.twig` in the `FOSUserBundle`, you can create your own file in the same location of `AcmeUserBundle`. Symfony will ignore the file that lives inside the `FOSUserBundle` entirely, and use your file instead.

The same goes for routing files, validation configuration and other resources.

The overriding of resources only works when you refer to resources with the `@FosUserBundle/Resources/config/routing/security.xml` method. If you refer to resources without using the @BundleName shortcut, they can't be overridden in this way.

Translation files do not work in the same way as described above. All translation files are accumulated into a set of "pools" (one for each) domain. Symfony loads translation files from bundles first (in the order that the bundles are initialized) and then from your `app/Resources` directory. If the same translation is specified in two resources, the translation from the resource that's loaded last will win.

Chapter 35

How to Override any Part of a Bundle

This document is a quick reference for how to override different parts of third-party bundles.

Templates

For information on overriding templates, see * *Overriding Bundle Templates*. * *How to use Bundle Inheritance to Override parts of a Bundle*

Routing

Routing is never automatically imported in Symfony2. If you want to include the routes from any bundle, then they must be manually imported from somewhere in your application (e.g. `app/config/routing.yml`).

The easiest way to "override" a bundle's routing is to never import it at all. Instead of importing a third-party bundle's routing, simply copying that routing file into your application, modify it, and import it instead.

Controllers

Assuming the third-party bundle involved uses non-service controllers (which is almost always the case), you can easily override controllers via bundle inheritance. For more information, see *How to use Bundle Inheritance to Override parts of a Bundle*.

Services & Configuration

In progress...

Entities & Entity mapping

In progress...

Forms

In progress...

Validation metadata

In progress...

Translations

In progress...

Chapter 36

How to expose a Semantic Configuration for a Bundle

If you open your application configuration file (usually `app/config/config.yml`), you'll see a number of different configuration "namespaces", such as `framework`, `twig`, and `doctrine`. Each of these configures a specific bundle, allowing you to configure things at a high level and then let the bundle make all the low-level, complex changes that result.

For example, the following tells the `FrameworkBundle` to enable the form integration, which involves the defining of quite a few services as well as integration of other related components:

Listing 36-1

```
framework:
    # ...
    form:            true
```

When you create a bundle, you have two choices on how to handle configuration:

1. **Normal Service Configuration** (*easy*):

    You can specify your services in a configuration file (e.g. `services.yml`) that lives in your bundle and then import it from your main application configuration. This is really easy, quick and totally effective. If you make use of *parameters*, then you still have the flexibility to customize your bundle from your application configuration. See "*Importing Configuration with imports*" for more details.

2. **Exposing Semantic Configuration** (*advanced*):

    This is the way configuration is done with the core bundles (as described above). The basic idea is that, instead of having the user override individual parameters, you let the user configure just a few, specifically created options. As the bundle developer, you then parse through that configuration and load services inside an "Extension" class. With this method, you won't need to import any configuration resources from your main application configuration: the Extension class can handle all of this.

The second option - which you'll learn about in this article - is much more flexible, but also requires more time to setup. If you're wondering which method you should use, it's probably a good idea to start with method #1, and then change to #2 later if you need to.

The second method has several specific advantages:

- Much more powerful than simply defining parameters: a specific option value might trigger the creation of many service definitions;
- Ability to have configuration hierarchy
- Smart merging when several configuration files (e.g. `config_dev.yml` and `config.yml`) override each other's configuration;
- Configuration validation (if you use a *Configuration Class*);
- IDE auto-completion when you create an XSD and developers use XML.

Overriding bundle parameters

If a Bundle provides an Extension class, then you should generally *not* override any service container parameters from that bundle. The idea is that if an Extension class is present, every setting that should be configurable should be present in the configuration made available by that class. In other words the extension class defines all the publicly supported configuration settings for which backward compatibility will be maintained.

Creating an Extension Class

If you do choose to expose a semantic configuration for your bundle, you'll first need to create a new "Extension" class, which will handle the process. This class should live in the `DependencyInjection` directory of your bundle and its name should be constructed by replacing the `Bundle` suffix of the Bundle class name with `Extension`. For example, the Extension class of `AcmeHelloBundle` would be called `AcmeHelloExtension`:

Listing 36-2

```php
// Acme/HelloBundle/DependencyInjection/AcmeHelloExtension.php
use Symfony\Component\HttpKernel\DependencyInjection\Extension;
use Symfony\Component\DependencyInjection\ContainerBuilder;

class AcmeHelloExtension extends Extension
{
    public function load(array $configs, ContainerBuilder $container)
    {
        // where all of the heavy logic is done
    }

    public function getXsdValidationBasePath()
    {
        return __DIR__.'/../Resources/config/';
    }

    public function getNamespace()
    {
        return 'http://www.example.com/symfony/schema/';
    }
}
```

The `getXsdValidationBasePath` and `getNamespace` methods are only required if the bundle provides optional XSD's for the configuration.

The presence of the previous class means that you can now define an `acme_hello` configuration namespace in any configuration file. The namespace `acme_hello` is constructed from the extension's class name by removing the word `Extension` and then lowercasing and underscoring the rest of the name. In other words, `AcmeHelloExtension` becomes `acme_hello`.

You can begin specifying configuration under this namespace immediately:

Listing 36-3

```
# app/config/config.yml
acme_hello: ~
```

If you follow the naming conventions laid out above, then the `load()` method of your extension code is always called as long as your bundle is registered in the Kernel. In other words, even if the user does not provide any configuration (i.e. the `acme_hello` entry doesn't even appear), the `load()` method will be called and passed an empty `$configs` array. You can still provide some sensible defaults for your bundle if you want.

Parsing the `$configs` Array

Whenever a user includes the `acme_hello` namespace in a configuration file, the configuration under it is added to an array of configurations and passed to the `load()` method of your extension (Symfony2 automatically converts XML and YAML to an array).

Take the following configuration:

Listing 36-4

```
# app/config/config.yml
acme_hello:
    foo: fooValue
    bar: barValue
```

The array passed to your `load()` method will look like this:

Listing 36-5

```
array(
    array(
        'foo' => 'fooValue',
        'bar' => 'barValue',
    )
)
```

Notice that this is an *array of arrays*, not just a single flat array of the configuration values. This is intentional. For example, if `acme_hello` appears in another configuration file - say `config_dev.yml` - with different values beneath it, then the incoming array might look like this:

Listing 36-6

```
array(
    array(
        'foo' => 'fooValue',
        'bar' => 'barValue',
    ),
    array(
        'foo' => 'fooDevValue',
        'baz' => 'newConfigEntry',
    ),
)
```

The order of the two arrays depends on which one is set first.

It's your job, then, to decide how these configurations should be merged together. You might, for example, have later values override previous values or somehow merge them together.

Later, in the *Configuration Class* section, you'll learn of a truly robust way to handle this. But for now, you might just merge them manually:

Listing 36-7

```php
public function load(array $configs, ContainerBuilder $container)
{
    $config = array();
    foreach ($configs as $subConfig) {
        $config = array_merge($config, $subConfig);
    }

    // now use the flat $config array
}
```

⚠ Make sure the above merging technique makes sense for your bundle. This is just an example, and you should be careful to not use it blindly.

Using the `load()` Method

Within `load()`, the `$container` variable refers to a container that only knows about this namespace configuration (i.e. it doesn't contain service information loaded from other bundles). The goal of the `load()` method is to manipulate the container, adding and configuring any methods or services needed by your bundle.

Loading External Configuration Resources

One common thing to do is to load an external configuration file that may contain the bulk of the services needed by your bundle. For example, suppose you have a `services.xml` file that holds much of your bundle's service configuration:

Listing 36-8

```php
use Symfony\Component\DependencyInjection\Loader\XmlFileLoader;
use Symfony\Component\Config\FileLocator;

public function load(array $configs, ContainerBuilder $container)
{
    // prepare your $config variable

    $loader = new XmlFileLoader($container, new FileLocator(__DIR__.'/../Resources/config'));
    $loader->load('services.xml');
}
```

You might even do this conditionally, based on one of the configuration values. For example, suppose you only want to load a set of services if an `enabled` option is passed and set to true:

Listing 36-9

```php
public function load(array $configs, ContainerBuilder $container)
{
    // prepare your $config variable

    $loader = new XmlFileLoader($container, new FileLocator(__DIR__.'/../Resources/config'));
    if (isset($config['enabled']) && $config['enabled']) {
        $loader->load('services.xml');
    }
}
```

Configuring Services and Setting Parameters

Once you've loaded some service configuration, you may need to modify the configuration based on some of the input values. For example, suppose you have a service whose first argument is some string "type" that it will use internally. You'd like this to be easily configured by the bundle user, so in your service configuration file (e.g. `services.xml`), you define this service and use a blank parameter - `acme_hello.my_service_type` - as its first argument:

Listing 36-10

```
<!-- src/Acme/HelloBundle/Resources/config/services.xml -->
<container xmlns="http://symfony.com/schema/dic/services"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/dic/services http://symfony.com/schema/dic/
services/services-1.0.xsd">

    <parameters>
        <parameter key="acme_hello.my_service_type" />
    </parameters>

    <services>
        <service id="acme_hello.my_service" class="Acme\HelloBundle\MyService">
            <argument>%acme_hello.my_service_type%</argument>
        </service>
    </services>
</container>
```

But why would you define an empty parameter and then pass it to your service? The answer is that you'll set this parameter in your extension class, based on the incoming configuration values. Suppose, for example, that you want to allow the user to define this *type* option under a key called `my_type`. Add the following to the `load()` method to do this:

Listing 36-11

```
public function load(array $configs, ContainerBuilder $container)
{
    // prepare your $config variable

    $loader = new XmlFileLoader($container, new FileLocator(__DIR__.'/../Resources/config'));
    $loader->load('services.xml');

    if (!isset($config['my_type'])) {
        throw new \InvalidArgumentException('The "my_type" option must be set');
    }

    $container->setParameter('acme_hello.my_service_type', $config['my_type']);
}
```

Now, the user can effectively configure the service by specifying the `my_type` configuration value:

Listing 36-12

```
# app/config/config.yml
acme_hello:
    my_type: foo
    # ...
```

Global Parameters

When you're configuring the container, be aware that you have the following global parameters available to use:

- `kernel.name`
- `kernel.environment`
- `kernel.debug`
- `kernel.root_dir`
- `kernel.cache_dir`
- `kernel.logs_dir`
- `kernel.bundle_dirs`
- `kernel.bundles`
- `kernel.charset`

All parameter and service names starting with a _ are reserved for the framework, and new ones must not be defined by bundles.

Validation and Merging with a Configuration Class

So far, you've done the merging of your configuration arrays by hand and are checking for the presence of config values manually using the `isset()` PHP function. An optional *Configuration* system is also available which can help with merging, validation, default values, and format normalization.

Format normalization refers to the fact that certain formats - largely XML - result in slightly different configuration arrays and that these arrays need to be "normalized" to match everything else.

To take advantage of this system, you'll create a `Configuration` class and build a tree that defines your configuration in that class:

Listing 36-13

```php
// src/Acme/HelloBundle/DependencyInjection/Configuration.php
namespace Acme\HelloBundle\DependencyInjection;

use Symfony\Component\Config\Definition\Builder\TreeBuilder;
use Symfony\Component\Config\Definition\ConfigurationInterface;

class Configuration implements ConfigurationInterface
{
    public function getConfigTreeBuilder()
    {
        $treeBuilder = new TreeBuilder();
        $rootNode = $treeBuilder->root('acme_hello');

        $rootNode
            ->children()
                ->scalarNode('my_type')->defaultValue('bar')->end()
            ->end()
        ;

        return $treeBuilder;
    }
```

This is a *very* simple example, but you can now use this class in your `load()` method to merge your configuration and force validation. If any options other than `my_type` are passed, the user will be notified with an exception that an unsupported option was passed:

Listing 36-14

```
public function load(array $configs, ContainerBuilder $container)
{
    $configuration = new Configuration();
    $config = $this->processConfiguration($configuration, $configs);

    // ...
}
```

The `processConfiguration()` method uses the configuration tree you've defined in the `Configuration` class to validate, normalize and merge all of the configuration arrays together.

The `Configuration` class can be much more complicated than shown here, supporting array nodes, "prototype" nodes, advanced validation, XML-specific normalization and advanced merging. The best way to see this in action is to checkout out some of the core Configuration classes, such as the one from the *FrameworkBundle Configuration*[1] or the *TwigBundle Configuration*[2].

Extension Conventions

When creating an extension, follow these simple conventions:

- The extension must be stored in the `DependencyInjection` sub-namespace;
- The extension must be named after the bundle name and suffixed with `Extension` (`AcmeHelloExtension` for `AcmeHelloBundle`);
- The extension should provide an XSD schema.

If you follow these simple conventions, your extensions will be registered automatically by Symfony2. If not, override the Bundle *`build()`*[3] method in your bundle:

Listing 36-15

```
use Acme\HelloBundle\DependencyInjection\UnconventionalExtensionClass;

class AcmeHelloBundle extends Bundle
{
    public function build(ContainerBuilder $container)
    {
        parent::build($container);

        // register extensions that do not follow the conventions manually
        $container->registerExtension(new UnconventionalExtensionClass());
    }
}
```

In this case, the extension class must also implement a `getAlias()` method and return a unique alias named after the bundle (e.g. `acme_hello`). This is required because the class name doesn't follow the standards by ending in `Extension`.

Additionally, the `load()` method of your extension will *only* be called if the user specifies the `acme_hello` alias in at least one configuration file. Once again, this is because the Extension class doesn't follow the standards set out above, so nothing happens automatically.

1. https://github.com/symfony/symfony/blob/master/src/Symfony/Bundle/FrameworkBundle/DependencyInjection/Configuration.php
2. https://github.com/symfony/symfony/blob/master/src/Symfony/Bundle/TwigBundle/DependencyInjection/Configuration.php
3. http://api.symfony.com/2.0/Symfony/Component/HttpKernel/Bundle/Bundle.html#build()

Chapter 37

How to send an Email

Sending emails is a classic task for any web application and one that has special complications and potential pitfalls. Instead of recreating the wheel, one solution to send emails is to use the `SwiftmailerBundle`, which leverages the power of the *Swiftmailer*[1] library.

Don't forget to enable the bundle in your kernel before using it:

Listing 37-1

```php
public function registerBundles()
{
    $bundles = array(
        // ...
        new Symfony\Bundle\SwiftmailerBundle\SwiftmailerBundle(),
    );

    // ...
}
```

Configuration

Before using Swiftmailer, be sure to include its configuration. The only mandatory configuration parameter is `transport`:

Listing 37-2

```yaml
# app/config/config.yml
swiftmailer:
    transport:  smtp
    encryption: ssl
    auth_mode:  login
    host:       smtp.gmail.com
    username:   your_username
    password:   your_password
```

1. `http://www.swiftmailer.org/`

The majority of the Swiftmailer configuration deals with how the messages themselves should be delivered.

The following configuration attributes are available:

- `transport` (`smtp`, `mail`, `sendmail`, or `gmail`)
- `username`
- `password`
- `host`
- `port`
- `encryption` (`tls`, or `ssl`)
- `auth_mode` (`plain`, `login`, or `cram-md5`)
- `spool`
    - `type` (how to queue the messages, only `file` is supported currently)
    - `path` (where to store the messages)
- `delivery_address` (an email address where to send ALL emails)
- `disable_delivery` (set to true to disable delivery completely)

Sending Emails

The Swiftmailer library works by creating, configuring and then sending `Swift_Message` objects. The "mailer" is responsible for the actual delivery of the message and is accessible via the `mailer` service. Overall, sending an email is pretty straightforward:

Listing 37-3

```
public function indexAction($name)
{
    $message = \Swift_Message::newInstance()
        ->setSubject('Hello Email')
        ->setFrom('send@example.com')
        ->setTo('recipient@example.com')
        ->setBody($this->renderView('HelloBundle:Hello:email.txt.twig', array('name' =>
$name)))
    ;
    $this->get('mailer')->send($message);

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

To keep things decoupled, the email body has been stored in a template and rendered with the `renderView()` method.

The `$message` object supports many more options, such as including attachments, adding HTML content, and much more. Fortunately, Swiftmailer covers the topic of *Creating Messages*[2] in great detail in its documentation.

Several other cookbook articles are available related to sending emails in Symfony2:

- *How to use Gmail to send Emails*
- *How to Work with Emails During Development*
- *How to Spool Email*

2. `http://swiftmailer.org/docs/messages.html`

Chapter 38

How to use Gmail to send Emails

During development, instead of using a regular SMTP server to send emails, you might find using Gmail easier and more practical. The Swiftmailer bundle makes it really easy.

Instead of using your regular Gmail account, it's of course recommended that you create a special account.

In the development configuration file, change the `transport` setting to `gmail` and set the `username` and `password` to the Google credentials:

Listing 38-1

```
# app/config/config_dev.yml
swiftmailer:
    transport: gmail
    username:  your_gmail_username
    password:  your_gmail_password
```

You're done!

The `gmail` transport is simply a shortcut that uses the `smtp` transport and sets `encryption`, `auth_mode` and `host` to work with Gmail.

Chapter 39

How to Work with Emails During Development

When developing an application which sends email, you will often not want to actually send the email to the specified recipient during development. If you are using the `SwiftmailerBundle` with Symfony2, you can easily achieve this through configuration settings without having to make any changes to your application's code at all. There are two main choices when it comes to handling email during development: (a) disabling the sending of email altogether or (b) sending all email to a specific address.

Disabling Sending

You can disable sending email by setting the `disable_delivery` option to `true`. This is the default in the `test` environment in the Standard distribution. If you do this in the `test` specific config then email will not be sent when you run tests, but will continue to be sent in the `prod` and `dev` environments:

Listing 39-1

```
# app/config/config_test.yml
swiftmailer:
    disable_delivery:  true
```

If you'd also like to disable deliver in the `dev` environment, simply add this same configuration to the `config_dev.yml` file.

Sending to a Specified Address

You can also choose to have all email sent to a specific address, instead of the address actually specified when sending the message. This can be done via the `delivery_address` option:

Listing 39-2

```
# app/config/config_dev.yml
swiftmailer:
    delivery_address:  dev@example.com
```

Now, suppose you're sending an email to `recipient@example.com`.

Listing 39-3

```
public function indexAction($name)
{
    $message = \Swift_Message::newInstance()
        ->setSubject('Hello Email')
        ->setFrom('send@example.com')
        ->setTo('recipient@example.com')
        ->setBody($this->renderView('HelloBundle:Hello:email.txt.twig', array('name' =>
$name)))
    ;
    $this->get('mailer')->send($message);

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

In the `dev` environment, the email will instead be sent to `dev@example.com`. Swiftmailer will add an extra header to the email, `X-Swift-To`, containing the replaced address, so you can still see who it would have been sent to.

In addition to the `to` addresses, this will also stop the email being sent to any `CC` and `BCC` addresses set for it. Swiftmailer will add additional headers to the email with the overridden addresses in them. These are `X-Swift-Cc` and `X-Swift-Bcc` for the `CC` and `BCC` addresses respectively.

Viewing from the Web Debug Toolbar

You can view any email sent during a single response when you are in the `dev` environment using the Web Debug Toolbar. The email icon in the toolbar will show how many emails were sent. If you click it, a report will open showing the details of the sent emails.

If you're sending an email and then immediately redirecting to another page, the web debug toolbar will not display an email icon or a report on the next page.

Instead, you can set the `intercept_redirects` option to `true` in the `config_dev.yml` file, which will cause the redirect to stop and allow you to open the report with details of the sent emails.

Alternatively, you can open the profiler after the redirect and search by the submit URL used on previous request (e.g. `/contact/handle`). The profiler's search feature allows you to load the profiler information for any past requests.

Listing 39-4

```
# app/config/config_dev.yml
web_profiler:
    intercept_redirects: true
```

Chapter 40

How to Spool Email

When you are using the `SwiftmailerBundle` to send an email from a Symfony2 application, it will default to sending the email immediately. You may, however, want to avoid the performance hit of the communication between `Swiftmailer` and the email transport, which could cause the user to wait for the next page to load while the email is sending. This can be avoided by choosing to "spool" the emails instead of sending them directly. This means that `Swiftmailer` does not attempt to send the email but instead saves the message to somewhere such as a file. Another process can then read from the spool and take care of sending the emails in the spool. Currently only spooling to file is supported by `Swiftmailer`.

In order to use the spool, use the following configuration:

Listing 40-1

```
# app/config/config.yml
swiftmailer:
    # ...
    spool:
        type: file
        path: /path/to/spool
```

If you want to store the spool somewhere with your project directory, remember that you can use the *%kernel.root_dir%* parameter to reference the project's root:

Listing 40-2

```
path: "%kernel.root_dir%/spool"
```

Now, when your app sends an email, it will not actually be sent but instead added to the spool. Sending the messages from the spool is done separately. There is a console command to send the messages in the spool:

Listing 40-3

```
php app/console swiftmailer:spool:send
```

It has an option to limit the number of messages to be sent:

Listing 40-4

```
php app/console swiftmailer:spool:send --message-limit=10
```

You can also set the time limit in seconds:

Listing 40-5

```
php app/console swiftmailer:spool:send --time-limit=10
```

Of course you will not want to run this manually in reality. Instead, the console command should be triggered by a cron job or scheduled task and run at a regular interval.

Chapter 41

How to simulate HTTP Authentication in a Functional Test

If your application needs HTTP authentication, pass the username and password as server variables to `createClient()`:

Listing 41-1

```php
$client = static::createClient(array(), array(
    'PHP_AUTH_USER' => 'username',
    'PHP_AUTH_PW'   => 'pa$$word',
));
```

You can also override it on a per request basis:

Listing 41-2

```php
$client->request('DELETE', '/post/12', array(), array(
    'PHP_AUTH_USER' => 'username',
    'PHP_AUTH_PW'   => 'pa$$word',
));
```

When your application is using a `form_login`, you can simplify your tests by allowing your test configuration to make use of HTTP authentication. This way you can use the above to authenticate in tests, but still have your users login via the normal `form_login`. The trick is to include the `http_basic` key in your firewall, along with the `form_login` key:

Listing 41-3

```yaml
# app/config/config_test.yml
security:
    firewalls:
        your_firewall_name:
            http_basic:
```

Chapter 42

How to test the Interaction of several Clients

If you need to simulate an interaction between different Clients (think of a chat for instance), create several Clients:

Listing 42-1

```
$harry = static::createClient();
$sally = static::createClient();

$harry->request('POST', '/say/sally/Hello');
$sally->request('GET', '/messages');

$this->assertEquals(201, $harry->getResponse()->getStatusCode());
$this->assertRegExp('/Hello/', $sally->getResponse()->getContent());
```

This works except when your code maintains a global state or if it depends on third-party libraries that has some kind of global state. In such a case, you can insulate your clients:

Listing 42-2

```
$harry = static::createClient();
$sally = static::createClient();

$harry->insulate();
$sally->insulate();

$harry->request('POST', '/say/sally/Hello');
$sally->request('GET', '/messages');

$this->assertEquals(201, $harry->getResponse()->getStatusCode());
$this->assertRegExp('/Hello/', $sally->getResponse()->getContent());
```

Insulated clients transparently execute their requests in a dedicated and clean PHP process, thus avoiding any side-effects.

As an insulated client is slower, you can keep one client in the main process, and insulate the other ones.

Chapter 43

How to use the Profiler in a Functional Test

It's highly recommended that a functional test only tests the Response. But if you write functional tests that monitor your production servers, you might want to write tests on the profiling data as it gives you a great way to check various things and enforce some metrics.

The Symfony2 *Profiler* gathers a lot of data for each request. Use this data to check the number of database calls, the time spent in the framework, ... But before writing assertions, always check that the profiler is indeed available (it is enabled by default in the `test` environment):

Listing 43-1

```
class HelloControllerTest extends WebTestCase
{
    public function testIndex()
    {
        $client = static::createClient();
        $crawler = $client->request('GET', '/hello/Fabien');

        // Write some assertions about the Response
        // ...

        // Check that the profiler is enabled
        if ($profile = $client->getProfile()) {
            // check the number of requests
            $this->assertTrue($profile->getCollector('db')->getQueryCount() < 10);

            // check the time spent in the framework
            $this->assertTrue( $profile->getCollector('timer')->getTime() < 0.5);
        }
    }
}
```

If a test fails because of profiling data (too many DB queries for instance), you might want to use the Web Profiler to analyze the request after the tests finish. It's easy to achieve if you embed the token in the error message:

Listing 43-2

```
$this->assertTrue(
    $profile->get('db')->getQueryCount() < 30,
    sprintf('Checks that query count is less than 30 (token %s)', $profile->getToken())
);
```

The profiler store can be different depending on the environment (especially if you use the SQLite store, which is the default configured one).

The profiler information is available even if you insulate the client or if you use an HTTP layer for your tests.

Read the API for built-in *data collectors* to learn more about their interfaces.

Chapter 44

How to test Doctrine Repositories

Unit testing Doctrine repositories in a Symfony project is not recommended. When you're dealing with a repository, you're really dealing with something that's meant to be tested against a real database connection.

Fortunately, you can easily test your queries against a real database, as described below.

Functional Testing

If you need to actually execute a query, you will need to boot the kernel to get a valid connection. In this case, you'll extend the `WebTestCase`, which makes all of this quite easy:

Listing 44-1

```
// src/Acme/StoreBundle/Tests/Entity/ProductRepositoryFunctionalTest.php

namespace Acme\StoreBundle\Tests\Entity;

use Symfony\Bundle\FrameworkBundle\Test\WebTestCase;

class ProductRepositoryFunctionalTest extends WebTestCase
{
    /**
     * @var \Doctrine\ORM\EntityManager
     */
    private $em;

    public function setUp()
    {
        $kernel = static::createKernel();
        $kernel->boot();
        $this->em = $kernel->getContainer()->get('doctrine.orm.entity_manager');
    }

    public function testProductByCategoryName()
    {
        $results = $this->em
            ->getRepository('AcmeStoreBundle:Product')
            ->searchProductsByNameQuery('foo')
            ->getResult()
        ;

        $this->assertCount(1, $results);
    }
}
```

Chapter 45

How to load Security Users from the Database (the Entity Provider)

The security layer is one of the smartest tools of Symfony. It handles two things: the authentication and the authorization processes. Although it may seem difficult to understand how it works internally, the security system is very flexible and allows you to integrate your application with any authentication backend, like Active Directory, an OAuth server or a database.

Introduction

This article focuses on how to authenticate users against a database table managed by a Doctrine entity class. The content of this cookbook entry is split in three parts. The first part is about designing a Doctrine `User` entity class and making it usable in the security layer of Symfony. The second part describes how to easily authenticate a user with the Doctrine *`EntityUserProvider`*[1] object bundled with the framework and some configuration. Finally, the tutorial will demonstrate how to create a custom *`EntityUserProvider`*[2] object to retrieve users from a database with custom conditions.

This tutorial assumes there is a bootstrapped and loaded `Acme\UserBundle` bundle in the application kernel.

The Data Model

For the purpose of this cookbook, the `AcmeUserBundle` bundle contains a `User` entity class with the following fields: `id`, `username`, `salt`, `password`, `email` and `isActive`. The `isActive` field tells whether or not the user account is active.

To make it shorter, the getter and setter methods for each have been removed to focus on the most important methods that come from the *`UserInterface`*[3].

1. http://api.symfony.com/2.0/Symfony/Bridge/Doctrine/Security/User/EntityUserProvider.html
2. http://api.symfony.com/2.0/Symfony/Bridge/Doctrine/Security/User/EntityUserProvider.html
3. http://api.symfony.com/2.0/Symfony/Component/Security/Core/User/UserInterface.html

Listing 45-1

```php
// src/Acme/UserBundle/Entity/User.php

namespace Acme\UserBundle\Entity;

use Doctrine\ORM\Mapping as ORM;
use Symfony\Component\Security\Core\User\UserInterface;

/**
 * Acme\UserBundle\Entity\User
 *
 * @ORM\Table(name="acme_users")
 * @ORM\Entity(repositoryClass="Acme\UserBundle\Entity\UserRepository")
 */
class User implements UserInterface
{
    /**
     * @ORM\Column(type="integer")
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="AUTO")
     */
    private $id;

    /**
     * @ORM\Column(type="string", length=25, unique=true)
     */
    private $username;

    /**
     * @ORM\Column(type="string", length=32)
     */
    private $salt;

    /**
     * @ORM\Column(type="string", length=40)
     */
    private $password;

    /**
     * @ORM\Column(type="string", length=60, unique=true)
     */
    private $email;

    /**
     * @ORM\Column(name="is_active", type="boolean")
     */
    private $isActive;

    public function __construct()
    {
        $this->isActive = true;
        $this->salt = md5(uniqid(null, true));
    }

    /**
     * @inheritDoc
     */
    public function getUsername()
    {
        return $this->username;
    }
    /**
     * @inheritDoc
     */
    public function getSalt()
    {
        return $this->salt;
    }

    /**
     * @inheritDoc
     */
    public function getPassword()
    {
        return $this->password;
    }

    /**
     * @inheritDoc
     */
    public function getRoles()
    {
        return array('ROLE_USER');
    }

    /**
     * @inheritDoc
     */
    public function eraseCredentials()
    {
    }

    /**
     * @inheritDoc
     */
    public function equals(UserInterface $user)
    {
        return $this->username === $user->getUsername();
    }
}
```

In order to use an instance of the `AcmeUserBundle:User` class in the Symfony security layer, the entity class must implement the *`UserInterface`*[4]. This interface forces the class to implement the six following methods:

- `getUsername()`
- `getSalt()`
- `getPassword()`
- `getRoles()`
- `eraseCredentials()`
- `equals()`

For more details on each of these, see *`UserInterface`*[5].

To keep it simple, the `equals()` method just compares the `username` field but it's also possible to do more checks depending on the complexity of your data model. On the other hand, the `eraseCredentials()` method remains empty as we don't care about it in this tutorial.

4. http://api.symfony.com/2.0/Symfony/Component/Security/Core/User/UserInterface.html
5. http://api.symfony.com/2.0/Symfony/Component/Security/Core/User/UserInterface.html

Below is an export of my `User` table from MySQL. For details on how to create user records and encode their password, see *Encoding the User's Password*.

Listing 45-2

```
mysql> select * from user;
+----+----------+----------------------------------+------------------------------------------+----------------
| id | username | salt                             | password
| email              | is_active |
+----+----------+----------------------------------+------------------------------------------+----------------
|  1 | hhamon   | 7308e59b97f6957fb42d66f894793079 | 09610f61637408828a35d7debee5b38a8350eebe
| hhamon@example.com |         1 |
|  2 | jsmith   | ce617a6cca9126bf4036ca0c02e82dee | 8390105917f3a3d533815250ed7c64b4594d7ebf
| jsmith@example.com |         1 |
|  3 | maxime   | cd01749bb995dc658fa56ed45458d807 | 9764731e5f7fb944de5fd8efad4949b995b72a3c
| maxime@example.com |         0 |
|  4 | donald   | 6683c2bfd90c0426088402930cadd0f8 | 5c3bcec385f59edcc04490d1db95fdb8673bf612
| donald@example.com |         1 |
+----+----------+----------------------------------+------------------------------------------+----------------
4 rows in set (0.00 sec)
```

The database now contains four users with different usernames, emails and statuses. The next part will focus on how to authenticate one of these users thanks to the Doctrine entity user provider and a couple of lines of configuration.

Authenticating Someone against a Database

Authenticating a Doctrine user against the database with the Symfony security layer is a piece of cake. Everything resides in the configuration of the *SecurityBundle* stored in the `app/config/security.yml` file.

Below is an example of configuration where the user will enter his/her username and password via HTTP basic authentication. That information will then be checked against our User entity records in the database:

Listing 45-3

```yaml
# app/config/security.yml

security:
    encoders:
        Acme\UserBundle\Entity\User:
            algorithm:        sha1
            encode_as_base64: false
            iterations:       1

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

    providers:
        administrators:
            entity: { class: AcmeUserBundle:User, property: username }

    firewalls:
        admin_area:
            pattern:    ^/admin
            http_basic: ~

    access_control:
        - { path: ^/admin, roles: ROLE_ADMIN }
```

The `encoders` section associates the `sha1` password encoder to the entity class. This means that Symfony will expect the password that's encoded in the database to be encoded using this algorithm. For details on how to create a new User object with a properly encoded password, see the *Encoding the User's Password* section of the security chapter.

The `providers` section defines an `administrators` user provider. A user provider is a "source" of where users are loaded during authentication. In this case, the `entity` keyword means that Symfony will use the Doctrine entity user provider to load User entity objects from the database by using the `username` unique field. In other words, this tells Symfony how to fetch the user from the database before checking the password validity.

This code and configuration works but it's not enough to secure the application for **active** users. As of now, we still can authenticate with `maxime`. The next section explains how to forbid non active users.

Forbid non Active Users

The easiest way to exclude non active users is to implement the *`AdvancedUserInterface`*[6] interface that takes care of checking the user's account status. The *`AdvancedUserInterface`*[7] extends the *`UserInterface`*[8] interface, so you just need to switch to the new interface in the `AcmeUserBundle:User` entity class to benefit from simple and advanced authentication behaviors.

The *`AdvancedUserInterface`*[9] interface adds four extra methods to validate the account status:

- `isAccountNonExpired()` checks whether the user's account has expired,
- `isAccountNonLocked()` checks whether the user is locked,
- `isCredentialsNonExpired()` checks whether the user's credentials (password) has expired,
- `isEnabled()` checks whether the user is enabled.

For this example, the first three methods will return `true` whereas the `isEnabled()` method will return the boolean value in the `isActive` field.

Listing 45-4

```php
// src/Acme/UserBundle/Entity/User.php

namespace Acme\Bundle\UserBundle\Entity;

// ...
use Symfony\Component\Security\Core\User\AdvancedUserInterface;

// ...
class User implements AdvancedUserInterface
{
    // ...
    public function isAccountNonExpired()
    {
        return true;
    }

    public function isAccountNonLocked()
    {
        return true;
    }

    public function isCredentialsNonExpired()
    {
```

6. `http://api.symfony.com/2.0/Symfony/Component/Security/Core/User/AdvancedUserInterface.html`
7. `http://api.symfony.com/2.0/Symfony/Component/Security/Core/User/AdvancedUserInterface.html`
8. `http://api.symfony.com/2.0/Symfony/Component/Security/Core/User/UserInterface.html`
9. `http://api.symfony.com/2.0/Symfony/Component/Security/Core/User/AdvancedUserInterface.html`

```
        return true;
    }

    public function isEnabled()
    {
        return $this->isActive;
    }
}
```

If we try to authenticate a `maxime`, the access is now forbidden as this user does not have an enabled account. The next session will focus on how to write a custom entity provider to authenticate a user with his username or his email address.

Authenticating Someone with a Custom Entity Provider

The next step is to allow a user to authenticate with his username or his email address as they are both unique in the database. Unfortunately, the native entity provider is only able to handle a single property to fetch the user from the database.

To accomplish this, create a custom entity provider that looks for a user whose username *or* email field matches the submitted login username. The good news is that a Doctrine repository object can act as an entity user provider if it implements the *UserProviderInterface*[10]. This interface comes with three methods to implement: `loadUserByUsername($username)`, `refreshUser(UserInterface $user)`, and `supportsClass($class)`. For more details, see *UserProviderInterface*[11].

The code below shows the implementation of the *UserProviderInterface*[12] in the `UserRepository` class:

Listing 45-5

```
// src/Acme/UserBundle/Entity/UserRepository.php

namespace Acme\UserBundle\Entity;

use Symfony\Component\Security\Core\User\UserInterface;
use Symfony\Component\Security\Core\User\UserProviderInterface;
use Symfony\Component\Security\Core\Exception\UsernameNotFoundException;
use Symfony\Component\Security\Core\Exception\UnsupportedUserException;
use Doctrine\ORM\EntityRepository;
use Doctrine\ORM\NoResultException;

class UserRepository extends EntityRepository implements UserProviderInterface
{
    public function loadUserByUsername($username)
    {
        $q = $this
            ->createQueryBuilder('u')
            ->where('u.username = :username OR u.email = :email')
            ->setParameter('username', $username)
            ->setParameter('email', $username)
            ->getQuery()
        ;

        try {
            // The Query::getSingleResult() method throws an exception
            // if there is no record matching the criteria.
            $user = $q->getSingleResult();
```

10. http://api.symfony.com/2.0/Symfony/Component/Security/Core/User/UserProviderInterface.html
11. http://api.symfony.com/2.0/Symfony/Component/Security/Core/User/UserProviderInterface.html
12. http://api.symfony.com/2.0/Symfony/Component/Security/Core/User/UserProviderInterface.html

```
        } catch (NoResultException $e) {
            throw new UsernameNotFoundException(sprintf('Unable to find an active admin
AcmeUserBundle:User object identified by "%s".', $username), null, 0, $e);
        }

        return $user;
    }

    public function refreshUser(UserInterface $user)
    {
        $class = get_class($user);
        if (!$this->supportsClass($class)) {
            throw new UnsupportedUserException(sprintf('Instances of "%s" are not supported.',
$class));
        }

        return $this->loadUserByUsername($user->getUsername());
    }

    public function supportsClass($class)
    {
        return $this->getEntityName() === $class || is_subclass_of($class,
$this->getEntityName());
    }
}
```

To finish the implementation, the configuration of the security layer must be changed to tell Symfony to use the new custom entity provider instead of the generic Doctrine entity provider. It's trival to achieve by removing the `property` field in the `security.providers.administrators.entity` section of the `security.yml` file.

Listing 45-6

```
# app/config/security.yml
security:
    # ...
    providers:
        administrators:
            entity: { class: AcmeUserBundle:User }
    # ...
```

By doing this, the security layer will use an instance of `UserRepository` and call its `loadUserByUsername()` method to fetch a user from the database whether he filled in his username or email address.

Managing Roles in the Database

The end of this tutorial focuses on how to store and retrieve a list of roles from the database. As mentioned previously, when your user is loaded, its `getRoles()` method returns the array of security roles that should be assigned to the user. You can load this data from anywhere - a hardcoded list used for all users (e.g. `array('ROLE_USER')`), a Doctrine array property called `roles`, or via a Doctrine relationship, as we'll learn about in this section.

⚠ In a typical setup, you should always return at least 1 role from the `getRoles()` method. By convention, a role called `ROLE_USER` is usually returned. If you fail to return any roles, it may appear as if your user isn't authenticated at all.

In this example, the `AcmeUserBundle:User` entity class defines a many-to-many relationship with a `AcmeUserBundle:Group` entity class. A user can be related several groups and a group can be composed of one or more users. As a group is also a role, the previous `getRoles()` method now returns the list of related groups:

Listing 45-7
```php
// src/Acme/UserBundle/Entity/User.php

namespace Acme\Bundle\UserBundle\Entity;

use Doctrine\Common\Collections\ArrayCollection;

// ...
class User implements AdvancedUserInterface
{
    /**
     * @ORM\ManyToMany(targetEntity="Group", inversedBy="users")
     *
     */
    private $groups;

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

    // ...

    public function getRoles()
    {
        return $this->groups->toArray();
    }
}
```

The `AcmeUserBundle:Group` entity class defines three table fields (`id`, `name` and `role`). The unique `role` field contains the role name used by the Symfony security layer to secure parts of the application. The most important thing to notice is that the `AcmeUserBundle:Group` entity class implements the *`RoleInterface`*[13] that forces it to have a `getRole()` method:

Listing 45-8
```php
namespace Acme\Bundle\UserBundle\Entity;

use Symfony\Component\Security\Core\Role\RoleInterface;
use Doctrine\Common\Collections\ArrayCollection;
use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Table(name="acme_groups")
 * @ORM\Entity()
 */
class Group implements RoleInterface
{
    /**
     * @ORM\Column(name="id", type="integer")
     * @ORM\Id()
     * @ORM\GeneratedValue(strategy="AUTO")
     */
    private $id;

    /**
```

13. `http://api.symfony.com/2.0/Symfony/Component/Security/Core/Role/RoleInterface.html`

```
     * @ORM\Column(name="name", type="string", length=30)
     */
    private $name;

    /**
     * @ORM\Column(name="role", type="string", length=20, unique=true)
     */
    private $role;

    /**
     * @ORM\ManyToMany(targetEntity="User", mappedBy="groups")
     */
    private $users;

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

    // ... getters and setters for each property

    /**
     * @see RoleInterface
     */
    public function getRole()
    {
        return $this->role;
    }
}
```

To improve performances and avoid lazy loading of groups when retrieving a user from the custom entity provider, the best solution is to join the groups relationship in the `UserRepository::loadUserByUsername()` method. This will fetch the user and his associated roles / groups with a single query:

Listing 45-9

```
// src/Acme/UserBundle/Entity/UserRepository.php

namespace Acme\Bundle\UserBundle\Entity;

// ...

class UserRepository extends EntityRepository implements UserProviderInterface
{
    public function loadUserByUsername($username)
    {
        $q = $this
            ->createQueryBuilder('u')
            ->select('u, g')
            ->leftJoin('u.groups', 'g')
            ->where('u.username = :username OR u.email = :email')
            ->setParameter('username', $username)
            ->setParameter('email', $username)
            ->getQuery()
        ;

        // ...
    }

    // ...
}
```

The `QueryBuilder::leftJoin()` method joins and fetches related groups from the `AcmeUserBundle:User` model class when a user is retrieved with his email address or username.

Chapter 46

How to add "Remember Me" Login Functionality

Once a user is authenticated, their credentials are typically stored in the session. This means that when the session ends they will be logged out and have to provide their login details again next time they wish to access the application. You can allow users to choose to stay logged in for longer than the session lasts using a cookie with the `remember_me` firewall option. The firewall needs to have a secret key configured, which is used to encrypt the cookie's content. It also has several options with default values which are shown here:

Listing 46-1

```
# app/config/security.yml

firewalls:
    main:
        remember_me:
            key:      "%secret%"
            lifetime: 3600
            path:     /
            domain:   ~ # Defaults to the current domain from $_SERVER
```

It's a good idea to provide the user with the option to use or not use the remember me functionality, as it will not always be appropriate. The usual way of doing this is to add a checkbox to the login form. By giving the checkbox the name `_remember_me`, the cookie will automatically be set when the checkbox is checked and the user successfully logs in. So, your specific login form might ultimately look like this:

Listing 46-2

```
{# src/Acme/SecurityBundle/Resources/views/Security/login.html.twig #}
{% if error %}
    <div>{{ error.message }}</div>
{% endif %}

<form action="{{ path('login_check') }}" method="post">
    <label for="username">Username:</label>
    <input type="text" id="username" name="_username" value="{{ last_username }}" />

    <label for="password">Password:</label>
    <input type="password" id="password" name="_password" />
    <input type="checkbox" id="remember_me" name="_remember_me" checked />
    <label for="remember_me">Keep me logged in</label>

    <input type="submit" name="login" />
</form>
```

The user will then automatically be logged in on subsequent visits while the cookie remains valid.

Forcing the User to Re-authenticate before accessing certain Resources

When the user returns to your site, he/she is authenticated automatically based on the information stored in the remember me cookie. This allows the user to access protected resources as if the user had actually authenticated upon visiting the site.

In some cases, however, you may want to force the user to actually re-authenticate before accessing certain resources. For example, you might allow a "remember me" user to see basic account information, but then require them to actually re-authenticate before modifying that information.

The security component provides an easy way to do this. In addition to roles explicitly assigned to them, users are automatically given one of the following roles depending on how they are authenticated:

- `IS_AUTHENTICATED_ANONYMOUSLY` - automatically assigned to a user who is in a firewall protected part of the site but who has not actually logged in. This is only possible if anonymous access has been allowed.
- `IS_AUTHENTICATED_REMEMBERED` - automatically assigned to a user who was authenticated via a remember me cookie.
- `IS_AUTHENTICATED_FULLY` - automatically assigned to a user that has provided their login details during the current session.

You can use these to control access beyond the explicitly assigned roles.

If you have the `IS_AUTHENTICATED_REMEMBERED` role, then you also have the `IS_AUTHENTICATED_ANONYMOUSLY` role. If you have the `IS_AUTHENTICATED_FULLY` role, then you also have the other two roles. In other words, these roles represent three levels of increasing "strength" of authentication.

You can use these additional roles for finer grained control over access to parts of a site. For example, you may want your user to be able to view their account at `/account` when authenticated by cookie but to have to provide their login details to be able to edit the account details. You can do this by securing specific controller actions using these roles. The edit action in the controller could be secured using the service context.

In the following example, the action is only allowed if the user has the `IS_AUTHENTICATED_FULLY` role.

Listing 46-3

```
use Symfony\Component\Security\Core\Exception\AccessDeniedException
// ...

public function editAction()
{
    if (false === $this->get('security.context')->isGranted(
        'IS_AUTHENTICATED_FULLY'
    )) {
        throw new AccessDeniedException();
    }
    // ...
}
```

You can also choose to install and use the optional *JMSSecurityExtraBundle*[1], which can secure your controller using annotations:

Listing 46-4

```
use JMS\SecurityExtraBundle\Annotation\Secure;

/**
 * @Secure(roles="IS_AUTHENTICATED_FULLY")
 */
public function editAction($name)
{
    // ...
}
```

If you also had an access control in your security configuration that required the user to have a `ROLE_USER` role in order to access any of the account area, then you'd have the following situation:

- If a non-authenticated (or anonymously authenticated user) tries to access the account area, the user will be asked to authenticate.
- Once the user has entered his username and password, assuming the user receives the `ROLE_USER` role per your configuration, the user will have the `IS_AUTHENTICATED_FULLY` role and be able to access any page in the account section, including the `editAction` controller.
- If the user's session ends, when the user returns to the site, he will be able to access every account page - except for the edit page - without being forced to re-authenticate. However, when he tries to access the `editAction` controller, he will be forced to re-authenticate, since he is not, yet, fully authenticated.

For more information on securing services or methods in this way, see *How to secure any Service or Method in your Application*.

1. `https://github.com/schmittjoh/JMSSecurityExtraBundle`

Chapter 47

How to implement your own Voter to blacklist IP Addresses

The Symfony2 security component provides several layers to authenticate users. One of the layers is called a *voter*. A voter is a dedicated class that checks if the user has the rights to be connected to the application. For instance, Symfony2 provides a layer that checks if the user is fully authenticated or if it has some expected roles.

It is sometimes useful to create a custom voter to handle a specific case not handled by the framework. In this section, you'll learn how to create a voter that will allow you to blacklist users by their IP.

The Voter Interface

A custom voter must implement *`VoterInterface`*[1], which requires the following three methods:

Listing 47-1

```
interface VoterInterface
{
    function supportsAttribute($attribute);
    function supportsClass($class);
    function vote(TokenInterface $token, $object, array $attributes);
}
```

The `supportsAttribute()` method is used to check if the voter supports the given user attribute (i.e: a role, an acl, etc.).

The `supportsClass()` method is used to check if the voter supports the current user token class.

The `vote()` method must implement the business logic that verifies whether or not the user is granted access. This method must return one of the following values:

- `VoterInterface::ACCESS_GRANTED`: The user is allowed to access the application
- `VoterInterface::ACCESS_ABSTAIN`: The voter cannot decide if the user is granted or not
- `VoterInterface::ACCESS_DENIED`: The user is not allowed to access the application

1. `http://api.symfony.com/2.0/Symfony/Component/Security/Core/Authorization/Voter/VoterInterface.html`

In this example, we will check if the user's IP address matches against a list of blacklisted addresses. If the user's IP is blacklisted, we will return `VoterInterface::ACCESS_DENIED`, otherwise we will return `VoterInterface::ACCESS_ABSTAIN` as this voter's purpose is only to deny access, not to grant access.

Creating a Custom Voter

To blacklist a user based on its IP, we can use the `request` service and compare the IP address against a set of blacklisted IP addresses:

Listing 47-2

```
namespace Acme\DemoBundle\Security\Authorization\Voter;

use Symfony\Component\DependencyInjection\ContainerInterface;
use Symfony\Component\Security\Core\Authorization\Voter\VoterInterface;
use Symfony\Component\Security\Core\Authentication\Token\TokenInterface;

class ClientIpVoter implements VoterInterface
{
    public function __construct(ContainerInterface $container, array $blacklistedIp = array())
    {
        $this->container     = $container;
        $this->blacklistedIp = $blacklistedIp;
    }

    public function supportsAttribute($attribute)
    {
        // we won't check against a user attribute, so we return true
        return true;
    }

    public function supportsClass($class)
    {
        // our voter supports all type of token classes, so we return true
        return true;
    }

    function vote(TokenInterface $token, $object, array $attributes)
    {
        $request = $this->container->get('request');
        if (in_array($request->getClientIp(), $this->blacklistedIp)) {
            return VoterInterface::ACCESS_DENIED;
        }

        return VoterInterface::ACCESS_ABSTAIN;
    }
}
```

That's it! The voter is done. The next step is to inject the voter into the security layer. This can be done easily through the service container.

Declaring the Voter as a Service

To inject the voter into the security layer, we must declare it as a service, and tag it as a "security.voter":

Listing 47-3

```
# src/Acme/AcmeBundle/Resources/config/services.yml

services:
    security.access.blacklist_voter:
        class:      Acme\DemoBundle\Security\Authorization\Voter\ClientIpVoter
        arguments:  [@service_container, [123.123.123.123, 171.171.171.171]]
        public:     false
        tags:
            -       { name: security.voter }
```

Be sure to import this configuration file from your main application configuration file (e.g. `app/config/config.yml`). For more information see *Importing Configuration with imports*. To read more about defining services in general, see the *Service Container* chapter.

Changing the Access Decision Strategy

In order for the new voter to take effect, we need to change the default access decision strategy, which, by default, grants access if *any* voter grants access.

In our case, we will choose the `unanimous` strategy. Unlike the `affirmative` strategy (the default), with the `unanimous` strategy, if only one voter denies access (e.g. the `ClientIpVoter`), access is not granted to the end user.

To do that, override the default `access_decision_manager` section of your application configuration file with the following code.

Listing 47-4

```
# app/config/security.yml
security:
    access_decision_manager:
        # Strategy can be: affirmative, unanimous or consensus
        strategy: unanimous
```

That's it! Now, when deciding whether or not a user should have access, the new voter will deny access to any user in the list of blacklisted IPs.

Chapter 48

Access Control Lists (ACLs)

In complex applications, you will often face the problem that access decisions cannot only be based on the person (`Token`) who is requesting access, but also involve a domain object that access is being requested for. This is where the ACL system comes in.

Imagine you are designing a blog system where your users can comment on your posts. Now, you want a user to be able to edit his own comments, but not those of other users; besides, you yourself want to be able to edit all comments. In this scenario, `Comment` would be our domain object that you want to restrict access to. You could take several approaches to accomplish this using Symfony2, two basic approaches are (non-exhaustive):

- *Enforce security in your business methods*: Basically, that means keeping a reference inside each `Comment` to all users who have access, and then compare these users to the provided `Token`.
- *Enforce security with roles*: In this approach, you would add a role for each `Comment` object, i.e. `ROLE_COMMENT_1`, `ROLE_COMMENT_2`, etc.

Both approaches are perfectly valid. However, they couple your authorization logic to your business code which makes it less reusable elsewhere, and also increases the difficulty of unit testing. Besides, you could run into performance issues if many users would have access to a single domain object.

Fortunately, there is a better way, which we will talk about now.

Bootstrapping

Now, before we finally can get into action, we need to do some bootstrapping. First, we need to configure the connection the ACL system is supposed to use:

Listing 48-1

```
# app/config/security.yml
security:
    acl:
        connection: default
```

The ACL system requires at least one Doctrine DBAL connection to be configured. However, that does not mean that you have to use Doctrine for mapping your domain objects. You can use whatever mapper you like for your objects, be it Doctrine ORM, Mongo ODM, Propel, or raw SQL, the choice is yours.

After the connection is configured, we have to import the database structure. Fortunately, we have a task for this. Simply run the following command:

Listing 48-2
```
php app/console init:acl
```

Getting Started

Coming back to our small example from the beginning, let's implement ACL for it.

Creating an ACL, and adding an ACE

Listing 48-3
```
use Symfony\Component\Security\Core\Exception\AccessDeniedException;
use Symfony\Component\Security\Acl\Domain\ObjectIdentity;
use Symfony\Component\Security\Acl\Domain\UserSecurityIdentity;
use Symfony\Component\Security\Acl\Permission\MaskBuilder;
// ...

// BlogController.php
public function addCommentAction(Post $post)
{
    $comment = new Comment();

    // setup $form, and bind data
    // ...

    if ($form->isValid()) {
        $entityManager = $this->get('doctrine.orm.default_entity_manager');
        $entityManager->persist($comment);
        $entityManager->flush();

        // creating the ACL
        $aclProvider = $this->get('security.acl.provider');
        $objectIdentity = ObjectIdentity::fromDomainObject($comment);
        $acl = $aclProvider->createAcl($objectIdentity);

        // retrieving the security identity of the currently logged-in user
        $securityContext = $this->get('security.context');
        $user = $securityContext->getToken()->getUser();
        $securityIdentity = UserSecurityIdentity::fromAccount($user);

        // grant owner access
        $acl->insertObjectAce($securityIdentity, MaskBuilder::MASK_OWNER);
        $aclProvider->updateAcl($acl);
    }
}
```

There are a couple of important implementation decisions in this code snippet. For now, I only want to highlight two:

First, you may have noticed that `->createAcl()` does not accept domain objects directly, but only implementations of the `ObjectIdentityInterface`. This additional step of indirection allows you to work with ACLs even when you have no actual domain object instance at hand. This will be extremely helpful if you want to check permissions for a large number of objects without actually hydrating these objects.

The other interesting part is the `->insertObjectAce()` call. In our example, we are granting the user who is currently logged in owner access to the Comment. The `MaskBuilder::MASK_OWNER` is a pre-defined integer bitmask; don't worry the mask builder will abstract away most of the technical details, but using this technique we can store many different permissions in one database row which gives us a considerable boost in performance.

The order in which ACEs are checked is significant. As a general rule, you should place more specific entries at the beginning.

Checking Access

Listing 48-4

```
// BlogController.php
public function editCommentAction(Comment $comment)
{
    $securityContext = $this->get('security.context');

    // check for edit access
    if (false === $securityContext->isGranted('EDIT', $comment))
    {
        throw new AccessDeniedException();
    }

    // retrieve actual comment object, and do your editing here
    // ...
}
```

In this example, we check whether the user has the `EDIT` permission. Internally, Symfony2 maps the permission to several integer bitmasks, and checks whether the user has any of them.

You can define up to 32 base permissions (depending on your OS PHP might vary between 30 to 32). In addition, you can also define cumulative permissions.

Cumulative Permissions

In our first example above, we only granted the user the `OWNER` base permission. While this effectively also allows the user to perform any operation such as view, edit, etc. on the domain object, there are cases where we want to grant these permissions explicitly.

The `MaskBuilder` can be used for creating bit masks easily by combining several base permissions:

Listing 48-5

```
$builder = new MaskBuilder();
$builder
    ->add('view')
    ->add('edit')
    ->add('delete')
    ->add('undelete')
;
$mask = $builder->get(); // int(15)
```

This integer bitmask can then be used to grant a user the base permissions you added above:

Listing 48-6
```
$acl->insertObjectAce(new UserSecurityIdentity('johannes'), $mask);
```

The user is now allowed to view, edit, delete, and un-delete objects.

Chapter 49

Advanced ACL Concepts

The aim of this chapter is to give a more in-depth view of the ACL system, and also explain some of the design decisions behind it.

Design Concepts

Symfony2's object instance security capabilities are based on the concept of an Access Control List. Every domain object **instance** has its own ACL. The ACL instance holds a detailed list of Access Control Entries (ACEs) which are used to make access decisions. Symfony2's ACL system focuses on two main objectives:

- providing a way to efficiently retrieve a large amount of ACLs/ACEs for your domain objects, and to modify them;
- providing a way to easily make decisions of whether a person is allowed to perform an action on a domain object or not.

As indicated by the first point, one of the main capabilities of Symfony2's ACL system is a high-performance way of retrieving ACLs/ACEs. This is extremely important since each ACL might have several ACEs, and inherit from another ACL in a tree-like fashion. Therefore, we specifically do not leverage any ORM, but the default implementation interacts with your connection directly using Doctrine's DBAL.

Object Identities

The ACL system is completely decoupled from your domain objects. They don't even have to be stored in the same database, or on the same server. In order to achieve this decoupling, in the ACL system your objects are represented through object identity objects. Everytime, you want to retrieve the ACL for a domain object, the ACL system will first create an object identity from your domain object, and then pass this object identity to the ACL provider for further processing.

Security Identities

This is analog to the object identity, but represents a user, or a role in your application. Each role, or user has its own security identity.

Database Table Structure

The default implementation uses five database tables as listed below. The tables are ordered from least rows to most rows in a typical application:

- *acl_security_identities*: This table records all security identities (SID) which hold ACEs. The default implementation ships with two security identities: `RoleSecurityIdentity`, and `UserSecurityIdentity`
- *acl_classes*: This table maps class names to a unique id which can be referenced from other tables.
- *acl_object_identities*: Each row in this table represents a single domain object instance.
- *acl_object_identity_ancestors*: This table allows us to determine all the ancestors of an ACL in a very efficient way.
- *acl_entries*: This table contains all ACEs. This is typically the table with the most rows. It can contain tens of millions without significantly impacting performance.

Scope of Access Control Entries

Access control entries can have different scopes in which they apply. In Symfony2, we have basically two different scopes:

- Class-Scope: These entries apply to all objects with the same class.
- Object-Scope: This was the scope we solely used in the previous chapter, and it only applies to one specific object.

Sometimes, you will find the need to apply an ACE only to a specific field of the object. Let's say you want the ID only to be viewable by an administrator, but not by your customer service. To solve this common problem, we have added two more sub-scopes:

- Class-Field-Scope: These entries apply to all objects with the same class, but only to a specific field of the objects.
- Object-Field-Scope: These entries apply to a specific object, and only to a specific field of that object.

Pre-Authorization Decisions

For pre-authorization decisions, that is decisions before any method, or secure action is invoked, we rely on the proven AccessDecisionManager service that is also used for reaching authorization decisions based on roles. Just like roles, the ACL system adds several new attributes which may be used to check for different permissions.

Built-in Permission Map

Attribute	**Intended Meaning**	**Integer Bitmasks**
VIEW	Whether someone is allowed to view the domain object.	VIEW, EDIT, OPERATOR, MASTER, or OWNER
EDIT	Whether someone is allowed to make changes to the domain object.	EDIT, OPERATOR, MASTER, or OWNER
CREATE	Whether someone is allowed to create the domain object.	CREATE, OPERATOR, MASTER, or OWNER

Attribute	Intended Meaning	Integer Bitmasks
DELETE	Whether someone is allowed to delete the domain object.	DELETE, OPERATOR, MASTER, or OWNER
UNDELETE	Whether someone is allowed to restore a previously deleted domain object.	UNDELETE, OPERATOR, MASTER, or OWNER
OPERATOR	Whether someone is allowed to perform all of the above actions.	OPERATOR, MASTER, or OWNER
MASTER	Whether someone is allowed to perform all of the above actions, and in addition is allowed to grant any of the above permissions to others.	MASTER, or OWNER
OWNER	Whether someone owns the domain object. An owner can perform any of the above actions *and* grant master and owner permissions.	OWNER

Permission Attributes vs. Permission Bitmasks

Attributes are used by the AccessDecisionManager, just like roles are attributes used by the AccessDecisionManager. Often, these attributes represent in fact an aggregate of integer bitmasks. Integer bitmasks on the other hand, are used by the ACL system internally to efficiently store your users' permissions in the database, and perform access checks using extremely fast bitmask operations.

Extensibility

The above permission map is by no means static, and theoretically could be completely replaced at will. However, it should cover most problems you encounter, and for interoperability with other bundles, we encourage you to stick to the meaning we have envisaged for them.

Post Authorization Decisions

Post authorization decisions are made after a secure method has been invoked, and typically involve the domain object which is returned by such a method. After invocation providers also allow to modify, or filter the domain object before it is returned.

Due to current limitations of the PHP language, there are no post-authorization capabilities build into the core Security component. However, there is an experimental *JMSSecurityExtraBundle*[1] which adds these capabilities. See its documentation for further information on how this is accomplished.

Process for Reaching Authorization Decisions

The ACL class provides two methods for determining whether a security identity has the required bitmasks, `isGranted` and `isFieldGranted`. When the ACL receives an authorization request through one of these methods, it delegates this request to an implementation of PermissionGrantingStrategy. This

1. `https://github.com/schmittjoh/JMSSecurityExtraBundle`

allows you to replace the way access decisions are reached without actually modifying the ACL class itself.

The PermissionGrantingStrategy first checks all your object-scope ACEs if none is applicable, the class-scope ACEs will be checked, if none is applicable, then the process will be repeated with the ACEs of the parent ACL. If no parent ACL exists, an exception will be thrown.

Chapter 50

How to force HTTPS or HTTP for Different URLs

You can force areas of your site to use the `HTTPS` protocol in the security config. This is done through the `access_control` rules using the `requires_channel` option. For example, if you want to force all URLs starting with `/secure` to use `HTTPS` then you could use the following config:

Listing 50-1

```
access_control:
    - path: ^/secure
      roles: ROLE_ADMIN
      requires_channel: https
```

The login form itself needs to allow anonymous access otherwise users will be unable to authenticate. To force it to use `HTTPS` you can still use `access_control` rules by using the `IS_AUTHENTICATED_ANONYMOUSLY` role:

Listing 50-2

```
access_control:
    - path: ^/login
      roles: IS_AUTHENTICATED_ANONYMOUSLY
      requires_channel: https
```

It is also possible to specify using `HTTPS` in the routing configuration see *How to force routes to always use HTTPS or HTTP* for more details.

Chapter 51

How to customize your Form Login

Using a *form login* for authentication is a common, and flexible, method for handling authentication in Symfony2. Pretty much every aspect of the form login can be customized. The full, default configuration is shown in the next section.

Form Login Configuration Reference

Listing 51-1

```
# app/config/security.yml
security:
    firewalls:
        main:
            form_login:
                # the user is redirected here when he/she needs to login
                login_path:                     /login

                # if true, forward the user to the login form instead of redirecting
                use_forward:                    false

                # submit the login form here
                check_path:                     /login_check

                # by default, the login form *must* be a POST, not a GET
                post_only:                      true

                # login success redirecting options (read further below)
                always_use_default_target_path: false
                default_target_path:            /
                target_path_parameter:          _target_path
                use_referer:                    false

                # login failure redirecting options (read further below)
                failure_path:                   null
                failure_forward:                false

                # field names for the username and password fields
                username_parameter:             _username
                password_parameter:             _password

                # csrf token options
                csrf_parameter:                 _csrf_token
                intention:                      authenticate
```

Redirecting after Success

You can change where the login form redirects after a successful login using the various config options. By default the form will redirect to the URL the user requested (i.e. the URL which triggered the login form being shown). For example, if the user requested `http://www.example.com/admin/post/18/edit` then after he/she will eventually be sent back to `http://www.example.com/admin/post/18/edit` after successfully logging in. This is done by storing the requested URL in the session. If no URL is present in the session (perhaps the user went directly to the login page), then the user is redirected to the default page, which is `/` (i.e. the homepage) by default. You can change this behavior in several ways.

Changing the Default Page

First, the default page can be set (i.e. the page the user is redirected to if no previous page was stored in the session). To set it to `/admin` use the following config:

Listing 51-2

```yaml
# app/config/security.yml
security:
    firewalls:
        main:
            form_login:
                # ...
                default_target_path: /admin
```

Now, when no URL is set in the session users will be sent to `/admin`.

Always Redirect to the Default Page

You can make it so that users are always redirected to the default page regardless of what URL they had requested previously by setting the `always_use_default_target_path` option to true:

Listing 51-3

```yaml
# app/config/security.yml
security:
    firewalls:
        main:
            form_login:
                # ...
                always_use_default_target_path: true
```

Using the Referring URL

In case no previous URL was stored in the session, you may wish to try using the `HTTP_REFERER` instead, as this will often be the same. You can do this by setting `use_referer` to true (it defaults to false):

Listing 51-4

```yaml
# app/config/security.yml
security:
    firewalls:
        main:
            form_login:
                # ...
                use_referer:        true
```

Control the Redirect URL from inside the Form

You can also override where the user is redirected to via the form itself by including a hidden field with the name `_target_path`. For example, to redirect to the URL defined by some `acount` route, use the following:

Listing 51-5

```
{# src/Acme/SecurityBundle/Resources/views/Security/login.html.twig #}
{% if error %}
    <div>{{ error.message }}</div>
{% endif %}

<form action="{{ path('login_check') }}" method="post">
    <label for="username">Username:</label>
    <input type="text" id="username" name="_username" value="{{ last_username }}" />

    <label for="password">Password:</label>
    <input type="password" id="password" name="_password" />

    <input type="hidden" name="_target_path" value="account" />

    <input type="submit" name="login" />
</form>
```

Now, the user will be redirected to the value of the hidden form field. The value attribute can be a relative path, absolute URL, or a route name. You can even change the name of the hidden form field by changing the `target_path_parameter` option to another value.

Listing 51-6

```
# app/config/security.yml
security:
    firewalls:
        main:
            form_login:
                target_path_parameter: redirect_url
```

Redirecting on Login Failure

In addition to redirect the user after a successful login, you can also set the URL that the user should be redirected to after a failed login (e.g. an invalid username or password was submitted). By default, the user is redirected back to the login form itself. You can set this to a different URL with the following config:

Listing 51-7

```
# app/config/security.yml
security:
    firewalls:
        main:
            form_login:
                # ...
                failure_path: /login_failure
```

Chapter 52

How to secure any Service or Method in your Application

In the security chapter, you can see how to *secure a controller* by requesting the `security.context` service from the Service Container and checking the current user's role:

Listing 52-1

```
use Symfony\Component\Security\Core\Exception\AccessDeniedException;
// ...

public function helloAction($name)
{
    if (false === $this->get('security.context')->isGranted('ROLE_ADMIN')) {
        throw new AccessDeniedException();
    }

    // ...
}
```

You can also secure *any* service in a similar way by injecting the `security.context` service into it. For a general introduction to injecting dependencies into services see the *Service Container* chapter of the book. For example, suppose you have a `NewsletterManager` class that sends out emails and you want to restrict its use to only users who have some `ROLE_NEWSLETTER_ADMIN` role. Before you add security, the class looks something like this:

Listing 52-2

```
namespace Acme\HelloBundle\Newsletter;

class NewsletterManager
{

    public function sendNewsletter()
    {
        // where you actually do the work
    }

    // ...
}
```

Your goal is to check the user's role when the `sendNewsletter()` method is called. The first step towards this is to inject the `security.context` service into the object. Since it won't make sense *not* to perform the security check, this is an ideal candidate for constructor injection, which guarantees that the security context object will be available inside the `NewsletterManager` class:

Listing 52-3

```
namespace Acme\HelloBundle\Newsletter;

use Symfony\Component\Security\Core\SecurityContextInterface;

class NewsletterManager
{
    protected $securityContext;

    public function __construct(SecurityContextInterface $securityContext)
    {
        $this->securityContext = $securityContext;
    }

    // ...
}
```

Then in your service configuration, you can inject the service:

Listing 52-4

```
# src/Acme/HelloBundle/Resources/config/services.yml
parameters:
    newsletter_manager.class: Acme\HelloBundle\Newsletter\NewsletterManager

services:
    newsletter_manager:
        class:     %newsletter_manager.class%
        arguments: [@security.context]
```

The injected service can then be used to perform the security check when the `sendNewsletter()` method is called:

Listing 52-5

```
namespace Acme\HelloBundle\Newsletter;

use Symfony\Component\Security\Core\Exception\AccessDeniedException;
use Symfony\Component\Security\Core\SecurityContextInterface;
// ...

class NewsletterManager
{
    protected $securityContext;

    public function __construct(SecurityContextInterface $securityContext)
    {
        $this->securityContext = $securityContext;
    }

    public function sendNewsletter()
    {
        if (false === $this->securityContext->isGranted('ROLE_NEWSLETTER_ADMIN')) {
            throw new AccessDeniedException();
        }

        //--
    }
    // ...
}
```

If the current user does not have the `ROLE_NEWSLETTER_ADMIN`, they will be prompted to log in.

Securing Methods Using Annotations

You can also secure method calls in any service with annotations by using the optional *JMSSecurityExtraBundle*[1] bundle. This bundle is included in the Symfony2 Standard Distribution.

To enable the annotations functionality, *tag* the service you want to secure with the `security.secure_service` tag (you can also automatically enable this functionality for all services, see the *sidebar* below):

Listing 52-6

```
# src/Acme/HelloBundle/Resources/config/services.yml
# ...

services:
    newsletter_manager:
        # ...
        tags:
            -  { name: security.secure_service }
```

You can then achieve the same results as above using an annotation:

Listing 52-7

```
namespace Acme\HelloBundle\Newsletter;

use JMS\SecurityExtraBundle\Annotation\Secure;
// ...

class NewsletterManager
{

    /**
     * @Secure(roles="ROLE_NEWSLETTER_ADMIN")
     */
    public function sendNewsletter()
    {
        //--
    }

    // ...
}
```

The annotations work because a proxy class is created for your class which performs the security checks. This means that, whilst you can use annotations on public and protected methods, you cannot use them with private methods or methods marked final.

The `JMSSecurityExtraBundle` also allows you to secure the parameters and return values of methods. For more information, see the *JMSSecurityExtraBundle*[2] documentation.

1. https://github.com/schmittjoh/JMSSecurityExtraBundle
2. https://github.com/schmittjoh/JMSSecurityExtraBundle

Activating the Annotations Functionality for all Services

When securing the method of a service (as shown above), you can either tag each service individually, or activate the functionality for *all* services at once. To do so, set the `secure_all_services` configuration option to true:

Listing 52-8

```
# app/config/config.yml
jms_security_extra:
    # ...
    secure_all_services: true
```

The disadvantage of this method is that, if activated, the initial page load may be very slow depending on how many services you have defined.

Chapter 53

How to create a custom User Provider

Part of Symfony's standard authentication process depends on "user providers". When a user submits a username and password, the authentication layer asks the configured user provider to return a user object for a given username. Symfony then checks whether the password of this user is correct and generates a security token so the user stays authenticated during the current session. Out of the box, Symfony has an "in_memory" and an "entity" user provider. In this entry we'll see how you can create your own user provider, which could be useful if your users are accessed via a custom database, a file, or - as we show in this example - a web service.

Create a User Class

First, regardless of *where* your user data is coming from, you'll need to create a `User` class that represents that data. The `User` can look however you want and contain any data. The only requirement is that the class implements *`UserInterface`*[1]. The methods in this interface should therefore be defined in the custom user class: `getRoles()`, `getPassword()`, `getSalt()`, `getUsername()`, `eraseCredentials()`, `equals()`.

Let's see this in action:

Listing 53-1

```
// src/Acme/WebserviceUserBundle/Security/User.php
namespace Acme\WebserviceUserBundle\Security\User;

use Symfony\Component\Security\Core\User\UserInterface;

class WebserviceUser implements UserInterface
{
    private $username;
    private $password;
    private $salt;
    private $roles;

    public function __construct($username, $password, $salt, array $roles)
    {
        $this->username = $username;
```

1. http://api.symfony.com/2.0/Symfony/Component/Security/Core/User/UserInterface.html

```
        $this->password = $password;
        $this->salt = $salt;
        $this->roles = $roles;
    }

    public function getRoles()
    {
        return $this->roles;
    }

    public function getPassword()
    {
        return $this->password;
    }

    public function getSalt()
    {
        return $this->salt;
    }

    public function getUsername()
    {
        return $this->username;
    }

    public function eraseCredentials()
    {
    }

    public function equals(UserInterface $user)
    {
        if (!$user instanceof WebserviceUser) {
            return false;
        }

        if ($this->password !== $user->getPassword()) {
            return false;
        }

        if ($this->getSalt() !== $user->getSalt()) {
            return false;
        }

        if ($this->username !== $user->getUsername()) {
            return false;
        }

        return true;
    }
}
```

If you have more information about your users - like a "first name" - then you can add a `firstName` field to hold that data.

For more details on each of the methods, see *`UserInterface`*[2].

2. http://api.symfony.com/2.0/Symfony/Component/Security/Core/User/UserInterface.html

Create a User Provider

Now that we have a `User` class, we'll create a user provider, which will grab user information from some web service, create a `WebserviceUser` object, and populate it with data.

The user provider is just a plain PHP class that has to implement the *`UserProviderInterface`*[3], which requires three methods to be defined: `loadUserByUsername($username)`, `refreshUser(UserInterface $user)`, and `supportsClass($class)`. For more details, see *`UserProviderInterface`*[4].

Here's an example of how this might look:

Listing 53-2

```
// src/Acme/WebserviceUserBundle/Security/User/WebserviceUserProvider.php
namespace Acme\WebserviceUserBundle\Security\User;

use Symfony\Component\Security\Core\User\UserProviderInterface;
use Symfony\Component\Security\Core\User\UserInterface;
use Symfony\Component\Security\Core\Exception\UsernameNotFoundException;
use Symfony\Component\Security\Core\Exception\UnsupportedUserException;

class WebserviceUserProvider implements UserProviderInterface
{
    public function loadUserByUsername($username)
    {
        // make a call to your webservice here
        // $userData = ...
        // pretend it returns an array on success, false if there is no user

        if ($userData) {
            // $password = '...';
            // ...

            return new WebserviceUser($username, $password, $salt, $roles)
        } else {
            throw new UsernameNotFoundException(sprintf('Username "%s" does not exist.',
$username));
        }
    }

    public function refreshUser(UserInterface $user)
    {
        if (!$user instanceof WebserviceUser) {
            throw new UnsupportedUserException(sprintf('Instances of "%s" are not supported.',
get_class($user)));
        }

        return $this->loadUserByUsername($user->getUsername());
    }

    public function supportsClass($class)
    {
        return $class === 'Acme\WebserviceUserBundle\Security\User\WebserviceUser';
    }
}
```

3. http://api.symfony.com/2.0/Symfony/Component/Security/Core/User/UserProviderInterface.html
4. http://api.symfony.com/2.0/Symfony/Component/Security/Core/User/UserProviderInterface.html

Create a Service for the User Provider

Now we make the user provider available as service.

Listing 53-3
```
# src/Acme/WebserviceUserBundle/Resources/config/services.yml
parameters:
    webservice_user_provider.class:
Acme\WebserviceUserBundle\Security\User\WebserviceUserProvider

services:
    webservice_user_provider:
        class: %webservice_user_provider.class%
```

The real implementation of the user provider will probably have some dependencies or configuration options or other services. Add these as arguments in the service definition.

Make sure the services file is being imported. See *Importing Configuration with imports* for details.

Modify `security.yml`

In `/app/config/security.yml` everything comes together. Add the user provider to the list of providers in the "security" section. Choose a name for the user provider (e.g. "webservice") and mention the id of the service you just defined.

Listing 53-4
```
security:
    providers:
        webservice:
            id: webservice_user_provider
```

Symfony also needs to know how to encode passwords that are supplied by website users, e.g. by filling in a login form. You can do this by adding a line to the "encoders" section in `/app/config/security.yml`.

Listing 53-5
```
security:
    encoders:
        Acme\WebserviceUserBundle\Security\User\WebserviceUser: sha512
```

The value here should correspond with however the passwords were originally encoded when creating your users (however those users were created). When a user submits her password, the password is appended to the salt value and then encoded using this algorithm before being compared to the hashed password returned by your `getPassword()` method. Additionally, depending on your options, the password may be encoded multiple times and encoded to base64.

Specifics on how passwords are encoded

Symfony uses a specific method to combine the salt and encode the password before comparing it to your encoded password. If `getSalt()` returns nothing, then the submitted password is simply encoded using the algorithm you specify in `security.yml`. If a salt *is* specified, then the following value is created and *then* hashed via the algorithm:

```
$password.'{'.$salt.'}';
```

If your external users have their passwords salted via a different method, then you'll need to do a bit more work so that Symfony properly encodes the password. That is beyond the scope of this entry, but would include sub-classing `MessageDigestPasswordEncoder` and overriding the `mergePasswordAndSalt` method.

Additionally, the hash, by default, is encoded multiple times and encoded to base64. For specific details, see *MessageDigestPasswordEncoder*[5]. To prevent this, configure it in `security.yml`:

Listing 53-6

```
security:
    encoders:
        Acme\WebserviceUserBundle\Security\User\WebserviceUser:
            algorithm: sha512
            encode_as_base64: false
            iterations: 1
```

5. `https://github.com/symfony/symfony/blob/master/src/Symfony/Component/Security/Core/Encoder/MessageDigestPasswordEncoder.php`

Chapter 54

How to create a custom Authentication Provider

If you have read the chapter on *Security*, you understand the distinction Symfony2 makes between authentication and authorization in the implementation of security. This chapter discusses the core classes involved in the authentication process, and how to implement a custom authentication provider. Because authentication and authorization are separate concepts, this extension will be user-provider agnostic, and will function with your application's user providers, may they be based in memory, a database, or wherever else you choose to store them.

Meet WSSE

The following chapter demonstrates how to create a custom authentication provider for WSSE authentication. The security protocol for WSSE provides several security benefits:

1. Username / Password encryption
2. Safe guarding against replay attacks
3. No web server configuration required

WSSE is very useful for the securing of web services, may they be SOAP or REST.

There is plenty of great documentation on *WSSE*[1], but this article will focus not on the security protocol, but rather the manner in which a custom protocol can be added to your Symfony2 application. The basis of WSSE is that a request header is checked for encrypted credentials, verified using a timestamp and *nonce*[2], and authenticated for the requested user using a password digest.

WSSE also supports application key validation, which is useful for web services, but is outside the scope of this chapter.

1. `http://www.xml.com/pub/a/2003/12/17/dive.html`
2. `http://en.wikipedia.org/wiki/Cryptographic_nonce`

The Token

The role of the token in the Symfony2 security context is an important one. A token represents the user authentication data present in the request. Once a request is authenticated, the token retains the user's data, and delivers this data across the security context. First, we will create our token class. This will allow the passing of all relevant information to our authentication provider.

Listing 54-1

```php
// src/Acme/DemoBundle/Security/Authentication/Token/WsseUserToken.php
namespace Acme\DemoBundle\Security\Authentication\Token;

use Symfony\Component\Security\Core\Authentication\Token\AbstractToken;

class WsseUserToken extends AbstractToken
{
    public $created;
    public $digest;
    public $nonce;

    public function getCredentials()
    {
        return '';
    }
}
```

The `WsseUserToken` class extends the security component's *`AbstractToken`*[3] class, which provides basic token functionality. Implement the *`TokenInterface`*[4] on any class to use as a token.

The Listener

Next, you need a listener to listen on the security context. The listener is responsible for fielding requests to the firewall and calling the authentication provider. A listener must be an instance of *`ListenerInterface`*[5]. A security listener should handle the *`GetResponseEvent`*[6] event, and set an authenticated token in the security context if successful.

Listing 54-2

```php
// src/Acme/DemoBundle/Security/Firewall/WsseListener.php
namespace Acme\DemoBundle\Security\Firewall;

use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\HttpKernel\Event\GetResponseEvent;
use Symfony\Component\Security\Http\Firewall\ListenerInterface;
use Symfony\Component\Security\Core\Exception\AuthenticationException;
use Symfony\Component\Security\Core\SecurityContextInterface;
use Symfony\Component\Security\Core\Authentication\AuthenticationManagerInterface;
use Symfony\Component\Security\Core\Authentication\Token\TokenInterface;
use Acme\DemoBundle\Security\Authentication\Token\WsseUserToken;

class WsseListener implements ListenerInterface
{
    protected $securityContext;
    protected $authenticationManager;
```

3. http://api.symfony.com/2.0/Symfony/Component/Security/Core/Authentication/Token/AbstractToken.html
4. http://api.symfony.com/2.0/Symfony/Component/Security/Core/Authentication/Token/TokenInterface.html
5. http://api.symfony.com/2.0/Symfony/Component/Security/Http/Firewall/ListenerInterface.html
6. http://api.symfony.com/2.0/Symfony/Component/HttpKernel/Event/GetResponseEvent.html

```
    public function __construct(SecurityContextInterface $securityContext,
AuthenticationManagerInterface $authenticationManager)
    {
        $this->securityContext = $securityContext;
        $this->authenticationManager = $authenticationManager;
    }

    public function handle(GetResponseEvent $event)
    {
        $request = $event->getRequest();

        if ($request->headers->has('x-wsse')) {

            $wsseRegex = '/UsernameToken Username="([^"]+)", PasswordDigest="([^"]+)",
Nonce="([^"]+)", Created="([^"]+)"/';

            if (preg_match($wsseRegex, $request->headers->get('x-wsse'), $matches)) {
                $token = new WsseUserToken();
                $token->setUser($matches[1]);

                $token->digest   = $matches[2];
                $token->nonce    = $matches[3];
                $token->created  = $matches[4];

                try {
                    $returnValue = $this->authenticationManager->authenticate($token);

                    if ($returnValue instanceof TokenInterface) {
                        return $this->securityContext->setToken($returnValue);
                    } else if ($returnValue instanceof Response) {
                        return $event->setResponse($returnValue);
                    }
                } catch (AuthenticationException $e) {
                    // you might log something here
                }
            }
        }

        $response = new Response();
        $response->setStatusCode(403);
        $event->setResponse($response);
    }
}
```

This listener checks the request for the expected *X-WSSE* header, matches the value returned for the expected WSSE information, creates a token using that information, and passes the token on to the authentication manager. If the proper information is not provided, or the authentication manager throws an *AuthenticationException*[7], a 403 Response is returned.

A class not used above, the *AbstractAuthenticationListener*[8] class, is a very useful base class which provides commonly needed functionality for security extensions. This includes maintaining the token in the session, providing success / failure handlers, login form urls, and more. As WSSE does not require maintaining authentication sessions or login forms, it won't be used for this example.

7. http://api.symfony.com/2.0/Symfony/Component/Security/Core/Exception/AuthenticationException.html
8. http://api.symfony.com/2.0/Symfony/Component/Security/Http/Firewall/AbstractAuthenticationListener.html

The Authentication Provider

The authentication provider will do the verification of the `WsseUserToken`. Namely, the provider will verify the `Created` header value is valid within five minutes, the `Nonce` header value is unique within five minutes, and the `PasswordDigest` header value matches with the user's password.

Listing 54-3

```
// src/Acme/DemoBundle/Security/Authentication/Provider/WsseProvider.php
namespace Acme\DemoBundle\Security\Authentication\Provider;

use Symfony\Component\Security\Core\Authentication\Provider\AuthenticationProviderInterface;
use Symfony\Component\Security\Core\User\UserProviderInterface;
use Symfony\Component\Security\Core\Exception\AuthenticationException;
use Symfony\Component\Security\Core\Exception\NonceExpiredException;
use Symfony\Component\Security\Core\Authentication\Token\TokenInterface;
use Acme\DemoBundle\Security\Authentication\Token\WsseUserToken;

class WsseProvider implements AuthenticationProviderInterface
{
    private $userProvider;
    private $cacheDir;

    public function __construct(UserProviderInterface $userProvider, $cacheDir)
    {
        $this->userProvider = $userProvider;
        $this->cacheDir     = $cacheDir;
    }

    public function authenticate(TokenInterface $token)
    {
        $user = $this->userProvider->loadUserByUsername($token->getUsername());

        if ($user && $this->validateDigest($token->digest, $token->nonce, $token->created,
$user->getPassword())) {
            $authenticatedToken = new WsseUserToken($user->getRoles());
            $authenticatedToken->setUser($user);

            return $authenticatedToken;
        }

        throw new AuthenticationException('The WSSE authentication failed.');
    }

    protected function validateDigest($digest, $nonce, $created, $secret)
    {
        // Expire timestamp after 5 minutes
        if (time() - strtotime($created) > 300) {
            return false;
        }

        // Validate nonce is unique within 5 minutes
        if (file_exists($this->cacheDir.'/'.$nonce) &&
file_get_contents($this->cacheDir.'/'.$nonce) + 300 < time()) {
            throw new NonceExpiredException('Previously used nonce detected');
        }
        file_put_contents($this->cacheDir.'/'.$nonce, time());

        // Validate Secret
        $expected = base64_encode(sha1(base64_decode($nonce).$created.$secret, true));

        return $digest === $expected;
    }

    public function supports(TokenInterface $token)
    {
        return $token instanceof WsseUserToken;
    }
}
```

The *AuthenticationProviderInterface*[9] requires an `authenticate` method on the user token, and a `supports` method, which tells the authentication manager whether or not to use this provider for the given token. In the case of multiple providers, the authentication manager will then move to the next provider in the list.

The Factory

You have created a custom token, custom listener, and custom provider. Now you need to tie them all together. How do you make your provider available to your security configuration? The answer is by using a `factory`. A factory is where you hook into the security component, telling it the name of your provider and any configuration options available for it. First, you must create a class which implements *SecurityFactoryInterface*[10].

Listing 54-4

```
// src/Acme/DemoBundle/DependencyInjection/Security/Factory/WsseFactory.php
namespace Acme\DemoBundle\DependencyInjection\Security\Factory;

use Symfony\Component\DependencyInjection\ContainerBuilder;
use Symfony\Component\DependencyInjection\Reference;
use Symfony\Component\DependencyInjection\DefinitionDecorator;
use Symfony\Component\Config\Definition\Builder\NodeDefinition;
use
Symfony\Bundle\SecurityBundle\DependencyInjection\Security\Factory\SecurityFactoryInterface;

class WsseFactory implements SecurityFactoryInterface
{
    public function create(ContainerBuilder $container, $id, $config, $userProvider,
$defaultEntryPoint)
    {
        $providerId = 'security.authentication.provider.wsse.'.$id;
        $container
            ->setDefinition($providerId, new
DefinitionDecorator('wsse.security.authentication.provider'))
            ->replaceArgument(0, new Reference($userProvider))
        ;

        $listenerId = 'security.authentication.listener.wsse.'.$id;
        $listener = $container->setDefinition($listenerId, new
DefinitionDecorator('wsse.security.authentication.listener'));

        return array($providerId, $listenerId, $defaultEntryPoint);
    }

    public function getPosition()
    {
        return 'pre_auth';
    }
```

9. http://api.symfony.com/2.0/Symfony/Component/Security/Core/Authentication/Provider/AuthenticationProviderInterface.html
10. http://api.symfony.com/2.0/Symfony/Bundle/SecurityBundle/DependencyInjection/Security/Factory/SecurityFactoryInterface.html

```
    public function getKey()
    {
        return 'wsse';
    }

    public function addConfiguration(NodeDefinition $node)
    {}
}
```

The *SecurityFactoryInterface*[11] requires the following methods:

- `create` method, which adds the listener and authentication provider to the DI container for the appropriate security context;
- `getPosition` method, which must be of type `pre_auth`, `form`, `http`, and `remember_me` and defines the position at which the provider is called;
- `getKey` method which defines the configuration key used to reference the provider;
- `addConfiguration` method, which is used to define the configuration options underneath the configuration key in your security configuration. Setting configuration options are explained later in this chapter.

A class not used in this example, *AbstractFactory*[12], is a very useful base class which provides commonly needed functionality for security factories. It may be useful when defining an authentication provider of a different type.

Now that you have created a factory class, the `wsse` key can be used as a firewall in your security configuration.

You may be wondering "why do we need a special factory class to add listeners and providers to the dependency injection container?". This is a very good question. The reason is you can use your firewall multiple times, to secure multiple parts of your application. Because of this, each time your firewall is used, a new service is created in the DI container. The factory is what creates these new services.

Configuration

It's time to see your authentication provider in action. You will need to do a few things in order to make this work. The first thing is to add the services above to the DI container. Your factory class above makes reference to service ids that do not exist yet: `wsse.security.authentication.provider` and `wsse.security.authentication.listener`. It's time to define those services.

Listing 54-5

```
# src/Acme/DemoBundle/Resources/config/services.yml
services:
  wsse.security.authentication.provider:
    class:  Acme\DemoBundle\Security\Authentication\Provider\WsseProvider
    arguments: ['', %kernel.cache_dir%/security/nonces]

  wsse.security.authentication.listener:
    class:  Acme\DemoBundle\Security\Firewall\WsseListener
    arguments: [@security.context, @security.authentication.manager]
```

11. http://api.symfony.com/2.0/Symfony/Bundle/SecurityBundle/DependencyInjection/Security/Factory/SecurityFactoryInterface.html
12. http://api.symfony.com/2.0/Symfony/Bundle/SecurityBundle/DependencyInjection/Security/Factory/AbstractFactory.html

Now that your services are defined, tell your security context about your factory. Factories must be included in an individual configuration file, at the time of this writing. So, start first by creating the file with the factory service, tagged as `security.listener.factory`:

Listing 54-6

```
# src/Acme/DemoBundle/Resources/config/security_factories.yml
services:
    security.authentication.factory.wsse:
        class:  Acme\DemoBundle\DependencyInjection\Security\Factory\WsseFactory
        tags:
            - { name: security.listener.factory }
```

Now, import the factory configuration via the the `factories` key in your security configuration:

Listing 54-7

```
# app/config/security.yml
security:
  factories:
    - "%kernel.root_dir%/../src/Acme/DemoBundle/Resources/config/security_factories.yml"
```

You are finished! You can now define parts of your app as under WSSE protection.

Listing 54-8

```
security:
    firewalls:
        wsse_secured:
            pattern:   /api/.*
            wsse:      true
```

Congratulations! You have written your very own custom security authentication provider!

A Little Extra

How about making your WSSE authentication provider a bit more exciting? The possibilities are endless. Why don't you start by adding some sparkle to that shine?

Configuration

You can add custom options under the `wsse` key in your security configuration. For instance, the time allowed before expiring the Created header item, by default, is 5 minutes. Make this configurable, so different firewalls can have different timeout lengths.

You will first need to edit `WsseFactory` and define the new option in the `addConfiguration` method.

Listing 54-9

```
class WsseFactory implements SecurityFactoryInterface
{
    # ...

    public function addConfiguration(NodeDefinition $node)
    {
      $node
        ->children()
          ->scalarNode('lifetime')->defaultValue(300)
        ->end()
      ;
    }
}
```

Now, in the `create` method of the factory, the `$config` argument will contain a 'lifetime' key, set to 5 minutes (300 seconds) unless otherwise set in the configuration. Pass this argument to your authentication provider in order to put it to use.

```php
class WsseFactory implements SecurityFactoryInterface
{
    public function create(ContainerBuilder $container, $id, $config, $userProvider,
$defaultEntryPoint)
    {
        $providerId = 'security.authentication.provider.wsse.'.$id;
        $container
            ->setDefinition($providerId,
              new DefinitionDecorator('wsse.security.authentication.provider'))
            ->replaceArgument(0, new Reference($userProvider))
            ->replaceArgument(2, $config['lifetime'])
        ;
        // ...
    }
    // ...
}
```

Listing 54-10

You'll also need to add a third argument to the `wsse.security.authentication.provider` service configuration, which can be blank, but will be filled in with the lifetime in the factory. The `WsseProvider` class will also now need to accept a third constructor argument - the lifetime - which it should use instead of the hard-coded 300 seconds. These two steps are not shown here.

The lifetime of each wsse request is now configurable, and can be set to any desirable value per firewall.

```yaml
security:
    firewalls:
        wsse_secured:
            pattern:   /api/.*
            wsse:      { lifetime: 30 }
```

Listing 54-11

The rest is up to you! Any relevant configuration items can be defined in the factory and consumed or passed to the other classes in the container.

Chapter 55

How to use Varnish to speed up my Website

Because Symfony2's cache uses the standard HTTP cache headers, the *Symfony2 Reverse Proxy* can easily be replaced with any other reverse proxy. Varnish is a powerful, open-source, HTTP accelerator capable of serving cached content quickly and including support for *Edge Side Includes*.

Configuration

As seen previously, Symfony2 is smart enough to detect whether it talks to a reverse proxy that understands ESI or not. It works out of the box when you use the Symfony2 reverse proxy, but you need a special configuration to make it work with Varnish. Thankfully, Symfony2 relies on yet another standard written by Akamaï (*Edge Architecture*[1]), so the configuration tips in this chapter can be useful even if you don't use Symfony2.

Varnish only supports the `src` attribute for ESI tags (`onerror` and `alt` attributes are ignored).

First, configure Varnish so that it advertises its ESI support by adding a `Surrogate-Capability` header to requests forwarded to the backend application:

Listing 55-1

```
sub vcl_recv {
    set req.http.Surrogate-Capability = "abc=ESI/1.0";
}
```

Then, optimize Varnish so that it only parses the Response contents when there is at least one ESI tag by checking the `Surrogate-Control` header that Symfony2 adds automatically:

Listing 55-2

```
sub vcl_fetch {
    if (beresp.http.Surrogate-Control ~ "ESI/1.0") {
        unset beresp.http.Surrogate-Control;

        // for Varnish >= 3.0
```

1. `http://www.w3.org/TR/edge-arch`

```
        set beresp.do_esi = true;
        // for Varnish < 3.0
        // esi;
    }
}
```

⚠ Compression with ESI was not supported in Varnish until version 3.0 (read *GZIP and Varnish*[2]). If you're not using Varnish 3.0, put a web server in front of Varnish to perform the compression.

Cache Invalidation

You should never need to invalidate cached data because invalidation is already taken into account natively in the HTTP cache models (see *Cache Invalidation*).

Still, Varnish can be configured to accept a special HTTP `PURGE` method that will invalidate the cache for a given resource:

Listing 55-3

```
sub vcl_hit {
    if (req.request == "PURGE") {
        set obj.ttl = 0s;
        error 200 "Purged";
    }
}

sub vcl_miss {
    if (req.request == "PURGE") {
        error 404 "Not purged";
    }
}
```

⚠ You must protect the `PURGE` HTTP method somehow to avoid random people purging your cached data.

2. `https://www.varnish-cache.org/docs/3.0/phk/gzip.html`

Chapter 56

Injecting variables into all templates (i.e. Global Variables)

Sometimes you want a variable to be accessible to all the templates you use. This is possible inside your `app/config/config.yml` file:

Listing 56-1
```
# app/config/config.yml
twig:
    # ...
    globals:
        ga_tracking: UA-xxxxx-x
```

Now, the variable `ga_tracking` is available in all Twig templates:

Listing 56-2
```
<p>Our google tracking code is: {{ ga_tracking }} </p>
```

It's that easy! You can also take advantage of the built-in *Service Parameters* system, which lets you isolate or reuse the value:

Listing 56-3
```
; app/config/parameters.ini
[parameters]
    ga_tracking: UA-xxxxx-x
```

Listing 56-4
```
# app/config/config.yml
twig:
    globals:
        ga_tracking: %ga_tracking%
```

The same variable is available exactly as before.

More Complex Global Variables

If the global variable you want to set is more complicated - say an object - then you won't be able to use the above method. Instead, you'll need to create a *Twig Extension* and return the global variable as one of the entries in the `getGlobals` method.

Chapter 57

How to use PHP instead of Twig for Templates

Even if Symfony2 defaults to Twig for its template engine, you can still use plain PHP code if you want. Both templating engines are supported equally in Symfony2. Symfony2 adds some nice features on top of PHP to make writing templates with PHP more powerful.

Rendering PHP Templates

If you want to use the PHP templating engine, first, make sure to enable it in your application configuration file:

Listing 57-1

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

You can now render a PHP template instead of a Twig one simply by using the `.php` extension in the template name instead of `.twig`. The controller below renders the `index.html.php` template:

Listing 57-2

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

public function indexAction($name)
{
    return $this->render('AcmeHelloBundle:Hello:index.html.php', array('name' => $name));
}
```

Decorating Templates

More often than not, templates in a project share common elements, like the well-known header and footer. In Symfony2, we like to think about this problem differently: a template can be decorated by another one.

The `index.html.php` template is decorated by `layout.html.php`, thanks to the `extend()` call:

Listing 57-3

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

Hello <?php echo $name ?>!
```

The `AcmeHelloBundle::layout.html.php` notation sounds familiar, doesn't it? It is the same notation used to reference a template. The `::` part simply means that the controller element is empty, so the corresponding file is directly stored under `views/`.

Now, let's have a look at the `layout.html.php` file:

Listing 57-4

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

<h1>Hello Application</h1>

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

The layout is itself decorated by another one (`::base.html.php`). Symfony2 supports multiple decoration levels: a layout can itself be decorated by another one. When the bundle part of the template name is empty, views are looked for in the `app/Resources/views/` directory. This directory store global views for your entire project:

Listing 57-5

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

For both layouts, the `$view['slots']->output('_content')` expression is replaced by the content of the child template, `index.html.php` and `layout.html.php` respectively (more on slots in the next section).

As you can see, Symfony2 provides methods on a mysterious `$view` object. In a template, the `$view` variable is always available and refers to a special object that provides a bunch of methods that makes the template engine tick.

Working with Slots

A slot is a snippet of code, defined in a template, and reusable in any layout decorating the template. In the `index.html.php` template, define a `title` slot:

Listing 57-6

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

<?php $view['slots']->set('title', 'Hello World Application') ?>

Hello <?php echo $name ?>!
```

The base layout already has the code to output the title in the header:

Listing 57-7

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

The `output()` method inserts the content of a slot and optionally takes a default value if the slot is not defined. And `_content` is just a special slot that contains the rendered child template.

For large slots, there is also an extended syntax:

Listing 57-8

```
<?php $view['slots']->start('title') ?>
    Some large amount of HTML
<?php $view['slots']->stop() ?>
```

Including other Templates

The best way to share a snippet of template code is to define a template that can then be included into other templates.

Create a `hello.html.php` template:

Listing 57-9

```
<!-- src/Acme/HelloBundle/Resources/views/Hello/hello.html.php -->
Hello <?php echo $name ?>!
```

And change the `index.html.php` template to include it:

Listing 57-10

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

<?php echo $view->render('AcmeHelloBundle:Hello:hello.html.php', array('name' => $name)) ?>
```

The `render()` method evaluates and returns the content of another template (this is the exact same method as the one used in the controller).

Embedding other Controllers

And what if you want to embed the result of another controller in a template? That's very useful when working with Ajax, or when the embedded template needs some variable not available in the main template.

If you create a `fancy` action, and want to include it into the `index.html.php` template, simply use the following code:

Listing 57-11

```
<!-- src/Acme/HelloBundle/Resources/views/Hello/index.html.php -->
<?php echo $view['actions']->render('AcmeHelloBundle:Hello:fancy', array('name' => $name,
'color' => 'green')) ?>
```

Here, the `AcmeHelloBundle:Hello:fancy` string refers to the `fancy` action of the `Hello` controller:

Listing 57-12

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

class HelloController extends Controller
{
    public function fancyAction($name, $color)
    {
        // create some object, based on the $color variable
        $object = ...;
        return $this->render('AcmeHelloBundle:Hello:fancy.html.php', array('name' => $name,
'object' => $object));
    }

    // ...
}
```

But where is the `$view['actions']` array element defined? Like `$view['slots']`, it's called a template helper, and the next section tells you more about those.

Using Template Helpers

The Symfony2 templating system can be easily extended via helpers. Helpers are PHP objects that provide features useful in a template context. `actions` and `slots` are two of the built-in Symfony2 helpers.

Creating Links between Pages

Speaking of web applications, creating links between pages is a must. Instead of hardcoding URLs in templates, the `router` helper knows how to generate URLs based on the routing configuration. That way, all your URLs can be easily updated by changing the configuration:

Listing 57-13

```
<a href="<?php echo $view['router']->generate('hello', array('name' => 'Thomas')) ?>">
    Greet Thomas!
</a>
```

The `generate()` method takes the route name and an array of parameters as arguments. The route name is the main key under which routes are referenced and the parameters are the values of the placeholders defined in the route pattern:

Listing 57-14

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

Using Assets: images, JavaScripts, and stylesheets

What would the Internet be without images, JavaScripts, and stylesheets? Symfony2 provides the `assets` tag to deal with them easily:

Listing 57-15

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

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

The `assets` helper's main purpose is to make your application more portable. Thanks to this helper, you can move the application root directory anywhere under your web root directory without changing anything in your template's code.

Output Escaping

When using PHP templates, escape variables whenever they are displayed to the user:

Listing 57-16

```
<?php echo $view->escape($var) ?>
```

By default, the `escape()` method assumes that the variable is outputted within an HTML context. The second argument lets you change the context. For instance, to output something in a JavaScript script, use the `js` context:

Listing 57-17

```
<?php echo $view->escape($var, 'js') ?>
```

Chapter 58

How to write a custom Twig Extension

The main motivation for writing an extension is to move often used code into a reusable class like adding support for internationalization. An extension can define tags, filters, tests, operators, global variables, functions, and node visitors.

Creating an extension also makes for a better separation of code that is executed at compilation time and code needed at runtime. As such, it makes your code faster.

Before writing your own extensions, have a look at the *Twig official extension repository*[1].

Create the Extension Class

To get your custom functionality you must first create a Twig Extension class. As an example we will create a price filter to format a given number into price:

Listing 58-1

```
// src/Acme/DemoBundle/Twig/AcmeExtension.php

namespace Acme\DemoBundle\Twig;

use Twig_Extension;
use Twig_Filter_Method;
use Twig_Function_Method;

class AcmeExtension extends Twig_Extension
{
    public function getFilters()
    {
        return array(
            'price' => new Twig_Filter_Method($this, 'priceFilter'),
        );
    }
```

1. http://github.com/fabpot/Twig-extensions

```
    public function priceFilter($number, $decimals = 0, $decPoint = '.', $thousandsSep = ',')
    {
        $price = number_format($number, $decimals, $decPoint, $thousandsSep);
        $price = '$' . $price;

        return $price;
    }

    public function getName()
    {
        return 'acme_extension';
    }
}
```

Along with custom filters, you can also add custom *functions* and register *global variables*.

Register an Extension as a Service

Now you must let Service Container know about your newly created Twig Extension:

Listing 58-2

```
<!-- src/Acme/DemoBundle/Resources/config/services.xml -->
<services>
    <service id="acme.twig.acme_extension" class="Acme\DemoBundle\Twig\AcmeExtension">
        <tag name="twig.extension" />
    </service>
</services>
```

Keep in mind that Twig Extensions are not lazily loaded. This means that there's a higher chance that you'll get a **CircularReferenceException** or a **ScopeWideningInjectionException** if any services (or your Twig Extension in this case) are dependent on the request service. For more information take a look at *How to work with Scopes*.

Using the custom Extension

Using your newly created Twig Extension is no different than any other:

Listing 58-3

```
{# outputs $5,500.00 #}
{{ '5500' | price }}
```

Passing other arguments to your filter:

Listing 58-4

```
{# outputs $5500,2516 #}
{{ '5500.25155' | price(4, ',', '') }}
```

Learning further

For a more in-depth look into Twig Extensions, please take a look at the *Twig extensions documentation*[2].

2. http://twig.sensiolabs.org/doc/advanced.html#creating-an-extension

Chapter 59

How to use Monolog to write Logs

Monolog[1] is a logging library for PHP 5.3 used by Symfony2. It is inspired by the Python LogBook library.

Usage

In Monolog each logger defines a logging channel. Each channel has a stack of handlers to write the logs (the handlers can be shared).

When injecting the logger in a service you can *use a custom channel* to see easily which part of the application logged the message.

The basic handler is the `StreamHandler` which writes logs in a stream (by default in the `app/logs/prod.log` in the prod environment and `app/logs/dev.log` in the dev environment).

Monolog comes also with a powerful built-in handler for the logging in prod environment: `FingersCrossedHandler`. It allows you to store the messages in a buffer and to log them only if a message reaches the action level (ERROR in the configuration provided in the standard edition) by forwarding the messages to another handler.

To log a message simply get the logger service from the container in your controller:

Listing 59-1
```
$logger = $this->get('logger');
$logger->info('We just got the logger');
$logger->err('An error occurred');
```

Using only the methods of the *`LoggerInterface`*[2] interface allows to change the logger implementation without changing your code.

1. `https://github.com/Seldaek/monolog`
2. `http://api.symfony.com/2.0/Symfony/Component/HttpKernel/Log/LoggerInterface.html`

Using several handlers

The logger uses a stack of handlers which are called successively. This allows you to log the messages in several ways easily.

Listing 59-2

```
monolog:
    handlers:
        syslog:
            type: stream
            path: /var/log/symfony.log
            level: error
        main:
            type: fingers_crossed
            action_level: warning
            handler: file
        file:
            type: stream
            level: debug
```

The above configuration defines a stack of handlers which will be called in the order where they are defined.

The handler named "file" will not be included in the stack itself as it is used as a nested handler of the `fingers_crossed` handler.

If you want to change the config of MonologBundle in another config file you need to redefine the whole stack. It cannot be merged because the order matters and a merge does not allow to control the order.

Changing the formatter

The handler uses a `Formatter` to format the record before logging it. All Monolog handlers use an instance of `Monolog\Formatter\LineFormatter` by default but you can replace it easily. Your formatter must implement `Monolog\Formatter\FormatterInterface`.

Listing 59-3

```
services:
    my_formatter:
        class: Monolog\Formatter\JsonFormatter
monolog:
    handlers:
        file:
            type: stream
            level: debug
            formatter: my_formatter
```

Adding some extra data in the log messages

Monolog allows to process the record before logging it to add some extra data. A processor can be applied for the whole handler stack or only for a specific handler.

A processor is simply a callable receiving the record as it's first argument.

Processors are configured using the `monolog.processor` DIC tag. See the *reference about it*.

Adding a Session/Request Token

Sometimes it is hard to tell which entries in the log belong to which session and/or request. The following example will add a unique token for each request using a processor.

Listing 59-4

```
namespace Acme\MyBundle;

use Symfony\Component\HttpFoundation\Session;

class SessionRequestProcessor
{
    private $session;
    private $token;

    public function __construct(Session $session)
    {
        $this->session = $session;
    }

    public function processRecord(array $record)
    {
        if (null === $this->token) {
            try {
                $this->token = substr($this->session->getId(), 0, 8);
            } catch (\RuntimeException $e) {
                $this->token = '????????';
            }
            $this->token .= '-' . substr(uniqid(), -8);
        }
        $record['extra']['token'] = $this->token;

        return $record;
    }
}
```

Listing 59-5

```
services:
    monolog.formatter.session_request:
        class: Monolog\Formatter\LineFormatter
        arguments:
            - "[%%datetime%%] [%%extra.token%%] %%channel%%.%%level_name%%: %%message%%\n"

    monolog.processor.session_request:
        class: Acme\MyBundle\SessionRequestProcessor
        arguments:  [ @session ]
        tags:
            - { name: monolog.processor, method: processRecord }

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

If you use several handlers, you can also register the processor at the handler level instead of globally.

Chapter 60

How to Configure Monolog to Email Errors

Monolog[1] can be configured to send an email when an error occurs with an application. The configuration for this requires a few nested handlers in order to avoid receiving too many emails. This configuration looks complicated at first but each handler is fairly straight forward when it is broken down.

Listing 60-1

```
# app/config/config.yml
monolog:
    handlers:
        mail:
            type:         fingers_crossed
            action_level: critical
            handler:      buffered
        buffered:
            type:    buffer
            handler: swift
        swift:
            type:       swift_mailer
            from_email: error@example.com
            to_email:   error@example.com
            subject:    An Error Occurred!
            level:      debug
```

The `mail` handler is a `fingers_crossed` handler which means that it is only triggered when the action level, in this case `critical` is reached. It then logs everything including messages below the action level. The `critical` level is only triggered for 5xx HTTP code errors. The `handler` setting means that the output is then passed onto the `buffered` handler.

If you want both 400 level and 500 level errors to trigger an email, set the `action_level` to `error` instead of `critical`.

The `buffered` handler simply keeps all the messages for a request and then passes them onto the nested handler in one go. If you do not use this handler then each message will be emailed separately. This is

1. https://github.com/Seldaek/monolog

then passed to the `swift` handler. This is the handler that actually deals with emailing you the error. The settings for this are straightforward, the to and from addresses and the subject.

You can combine these handlers with other handlers so that the errors still get logged on the server as well as the emails being sent:

Listing 60-2

```
# app/config/config.yml
monolog:
    handlers:
        main:
            type:         fingers_crossed
            action_level: critical
            handler:      grouped
        grouped:
            type:    group
            members: [streamed, buffered]
        streamed:
            type:  stream
            path:  "%kernel.logs_dir%/%kernel.environment%.log"
            level: debug
        buffered:
            type:    buffer
            handler: swift
        swift:
            type:       swift_mailer
            from_email: error@example.com
            to_email:   error@example.com
            subject:    An Error Occurred!
            level:      debug
```

This uses the `group` handler to send the messages to the two group members, the `buffered` and the `stream` handlers. The messages will now be both written to the log file and emailed.

Chapter 61

How to create a Console Command

The Console page of the Components section (*The Console Component*) covers how to create a Console command. This cookbook articles covers the differences when creating Console commands within the Symfony2 framework.

Automatically Registering Commands

To make the console commands available automatically with Symfony2, create a `Command` directory inside your bundle and create a php file suffixed with `Command.php` for each command that you want to provide. For example, if you want to extend the `AcmeDemoBundle` (available in the Symfony Standard Edition) to greet us from the command line, create `GreetCommand.php` and add the following to it:

Listing 61-1

```
// src/Acme/DemoBundle/Command/GreetCommand.php
namespace Acme\DemoBundle\Command;

use Symfony\Bundle\FrameworkBundle\Command\ContainerAwareCommand;
use Symfony\Component\Console\Input\InputArgument;
use Symfony\Component\Console\Input\InputInterface;
use Symfony\Component\Console\Input\InputOption;
use Symfony\Component\Console\Output\OutputInterface;

class GreetCommand extends ContainerAwareCommand
{
    protected function configure()
    {
        $this
            ->setName('demo:greet')
            ->setDescription('Greet someone')
            ->addArgument('name', InputArgument::OPTIONAL, 'Who do you want to greet?')
            ->addOption('yell', null, InputOption::VALUE_NONE, 'If set, the task will yell in
uppercase letters')
        ;
    }

    protected function execute(InputInterface $input, OutputInterface $output)
    {
        $name = $input->getArgument('name');
        if ($name) {
            $text = 'Hello '.$name;
        } else {
            $text = 'Hello';
        }

        if ($input->getOption('yell')) {
            $text = strtoupper($text);
        }

        $output->writeln($text);
    }
}
```

This command will now automatically be available to run:

Listing 61-2

```
app/console demo:greet Fabien
```

Testing Commands

When testing commands used as part of the full framework *Application*[1] should be used instead of *Application*[2]:

Listing 61-3

```
use Symfony\Component\Console\Tester\CommandTester;
use Symfony\Bundle\FrameworkBundle\Console\Application;
use Acme\DemoBundle\Command\GreetCommand;

class ListCommandTest extends \PHPUnit_Framework_TestCase
{
    public function testExecute()
    {
        // mock the Kernel or create one depending on your needs
        $application = new Application($kernel);
        $application->add(new GreetCommand());

        $command = $application->find('demo:greet');
        $commandTester = new CommandTester($command);
        $commandTester->execute(array('command' => $command->getName()));

        $this->assertRegExp('/.../', $commandTester->getDisplay());

        // ...
    }
}
```

Getting Services from the Service Container

By using *ContainerAwareCommand*[3] as the base class for the command (instead of the more basic *Command*[4]), you have access to the service container. In other words, you have access to any configured service. For example, you could easily extend the task to be translatable:

1. http://api.symfony.com/2.0/Symfony/Bundle/FrameworkBundle/Console/Application.html
2. http://api.symfony.com/2.0/Symfony/Component/Console/Application.html
3. http://api.symfony.com/2.0/Symfony/Bundle/FrameworkBundle/Command/ContainerAwareCommand.html
4. http://api.symfony.com/2.0/Symfony/Component/Console/Command/Command.html

Listing 61-4

```php
protected function execute(InputInterface $input, OutputInterface $output)
{
    $name = $input->getArgument('name');
    $translator = $this->getContainer()->get('translator');
    if ($name) {
        $output->writeln($translator->trans('Hello %name%!', array('%name%' => $name)));
    } else {
        $output->writeln($translator->trans('Hello!'));
    }
}
```

Chapter 62

How to optimize your development Environment for debugging

When you work on a Symfony project on your local machine, you should use the `dev` environment (`app_dev.php` front controller). This environment configuration is optimized for two main purposes:

- Give the developer accurate feedback whenever something goes wrong (web debug toolbar, nice exception pages, profiler, ...);
- Be as similar as possible as the production environment to avoid problems when deploying the project.

Disabling the Bootstrap File and Class Caching

And to make the production environment as fast as possible, Symfony creates big PHP files in your cache containing the aggregation of PHP classes your project needs for every request. However, this behavior can confuse your IDE or your debugger. This recipe shows you how you can tweak this caching mechanism to make it friendlier when you need to debug code that involves Symfony classes.

The `app_dev.php` front controller reads as follows by default:

Listing 62-1

```
// ...

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

use Symfony\Component\HttpFoundation\Request;

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

To make you debugger happier, disable all PHP class caches by removing the call to `loadClassCache()` and by replacing the require statements like below:

```php
// ...

// require_once __DIR__.'/../app/bootstrap.php.cache';
require_once __DIR__.'/../vendor/symfony/src/Symfony/Component/ClassLoader/
UniversalClassLoader.php';
require_once __DIR__.'/../app/autoload.php';
require_once __DIR__.'/../app/AppKernel.php';

use Symfony\Component\HttpFoundation\Request;

$kernel = new AppKernel('dev', true);
// $kernel->loadClassCache();
$kernel->handle(Request::createFromGlobals())->send();
```

Listing 62-2

If you disable the PHP caches, don't forget to revert after your debugging session.

Some IDEs do not like the fact that some classes are stored in different locations. To avoid problems, you can either tell your IDE to ignore the PHP cache files, or you can change the extension used by Symfony for these files:

```php
$kernel->loadClassCache('classes', '.php.cache');
```

Listing 62-3

Chapter 63

How to extend a Class without using Inheritance

To allow multiple classes to add methods to another one, you can define the magic `__call()` method in the class you want to be extended like this:

Listing 63-1

```
class Foo
{
    // ...

    public function __call($method, $arguments)
    {
        // create an event named 'foo.method_is_not_found'
        $event = new HandleUndefinedMethodEvent($this, $method, $arguments);
        $this->dispatcher->dispatch($this, 'foo.method_is_not_found', $event);

        // no listener was able to process the event? The method does not exist
        if (!$event->isProcessed()) {
            throw new \Exception(sprintf('Call to undefined method %s::%s.', get_class($this),
$method));
        }

        // return the listener returned value
        return $event->getReturnValue();
    }
}
```

This uses a special `HandleUndefinedMethodEvent` that should also be created. This is a generic class that could be reused each time you need to use this pattern of class extension:

Listing 63-2

```
use Symfony\Component\EventDispatcher\Event;

class HandleUndefinedMethodEvent extends Event
{
    protected $subject;
    protected $method;
    protected $arguments;
    protected $returnValue;
    protected $isProcessed = false;

    public function __construct($subject, $method, $arguments)
    {
        $this->subject = $subject;
        $this->method = $method;
        $this->arguments = $arguments;
    }

    public function getSubject()
    {
        return $this->subject;
    }

    public function getMethod()
    {
        return $this->method;
    }

    public function getArguments()
    {
        return $this->arguments;
    }

    /**
     * Sets the value to return and stops other listeners from being notified
     */
    public function setReturnValue($val)
    {
        $this->returnValue = $val;
        $this->isProcessed = true;
        $this->stopPropagation();
    }

    public function getReturnValue($val)
    {
        return $this->returnValue;
    }

    public function isProcessed()
    {
        return $this->isProcessed;
    }
}
```

Next, create a class that will listen to the `foo.method_is_not_found` event and *add* the method `bar()`:

Listing 63-3

```
class Bar
{
    public function onFooMethodIsNotFound(HandleUndefinedMethodEvent $event)
    {
        // we only want to respond to the calls to the 'bar' method
        if ('bar' != $event->getMethod()) {
            // allow another listener to take care of this unknown method
            return;
        }

        // the subject object (the foo instance)
        $foo = $event->getSubject();
        // the bar method arguments
        $arguments = $event->getArguments();

        // do something
        // ...

        // set the return value
        $event->setReturnValue($someValue);
    }
}
```

Finally, add the new `bar` method to the `Foo` class by register an instance of `Bar` with the `foo.method_is_not_found` event:

Listing 63-4

```
$bar = new Bar();
$dispatcher->addListener('foo.method_is_not_found', $bar);
```

Chapter 64

How to customize a Method Behavior without using Inheritance

Doing something before or after a Method Call

If you want to do something just before, or just after a method is called, you can dispatch an event respectively at the beginning or at the end of the method:

Listing 64-1

```
class Foo
{
    // ...

    public function send($foo, $bar)
    {
        // do something before the method
        $event = new FilterBeforeSendEvent($foo, $bar);
        $this->dispatcher->dispatch('foo.pre_send', $event);

        // get $foo and $bar from the event, they may have been modified
        $foo = $event->getFoo();
        $bar = $event->getBar();
        // the real method implementation is here
        // $ret = ...;

        // do something after the method
        $event = new FilterSendReturnValue($ret);
        $this->dispatcher->dispatch('foo.post_send', $event);

        return $event->getReturnValue();
    }
}
```

In this example, two events are thrown: `foo.pre_send`, before the method is executed, and `foo.post_send` after the method is executed. Each uses a custom Event class to communicate information to the listeners of the two events. These event classes would need to be created by you and should allow, in this example, the variables `$foo`, `$bar` and `$ret` to be retrieved and set by the listeners.

For example, assuming the `FilterSendReturnValue` has a `setReturnValue` method, one listener might look like this:

Listing 64-2

```php
public function onFooPostSend(FilterSendReturnValue $event)
{
    $ret = $event->getReturnValue();
    // modify the original ``$ret`` value

    $event->setReturnValue($ret);
}
```

Chapter 65

How to register a new Request Format and Mime Type

Every `Request` has a "format" (e.g. `html`, `json`), which is used to determine what type of content to return in the `Response`. In fact, the request format, accessible via *`getRequestFormat()`*[1], is used to set the MIME type of the `Content-Type` header on the `Response` object. Internally, Symfony contains a map of the most common formats (e.g. `html`, `json`) and their associated MIME types (e.g. `text/html`, `application/json`). Of course, additional format-MIME type entries can easily be added. This document will show how you can add the `jsonp` format and corresponding MIME type.

Create an `kernel.request` Listener

The key to defining a new MIME type is to create a class that will "listen" to the `kernel.request` event dispatched by the Symfony kernel. The `kernel.request` event is dispatched early in Symfony's request handling process and allows you to modify the request object.

Create the following class, replacing the path with a path to a bundle in your project:

Listing 65-1

```php
// src/Acme/DemoBundle/RequestListener.php
namespace Acme\DemoBundle;

use Symfony\Component\HttpKernel\HttpKernelInterface;
use Symfony\Component\HttpKernel\Event\GetResponseEvent;

class RequestListener
{
    public function onKernelRequest(GetResponseEvent $event)
    {
        $event->getRequest()->setFormat('jsonp', 'application/javascript');
    }
}
```

1. `http://api.symfony.com/2.0/Symfony/Component/HttpFoundation/Request.html#getRequestFormat()`

Registering your Listener

As for any other listener, you need to add it in one of your configuration file and register it as a listener by adding the `kernel.event_listener` tag:

Listing 65-2

```
<!-- app/config/config.xml -->
<?xml version="1.0" ?>

<container xmlns="http://symfony.com/schema/dic/services"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/dic/services http://symfony.com/schema/dic/
services/services-1.0.xsd">
    <services>
    <service id="acme.demobundle.listener.request" class="Acme\DemoBundle\RequestListener">
        <tag name="kernel.event_listener" event="kernel.request" method="onKernelRequest" />
    </service>
    </services>
</container>
```

At this point, the `acme.demobundle.listener.request` service has been configured and will be notified when the Symfony kernel dispatches the `kernel.request` event.

You can also register the listener in a configuration extension class (see *Importing Configuration via Container Extensions* for more information).

Chapter 66

How to create a custom Data Collector

The Symfony2 *Profiler* delegates data collecting to data collectors. Symfony2 comes bundled with a few of them, but you can easily create your own.

Creating a Custom Data Collector

Creating a custom data collector is as simple as implementing the *DataCollectorInterface*[1]:

Listing 66-1

```
interface DataCollectorInterface
{
    /**
     * Collects data for the given Request and Response.
     *
     * @param Request    $request   A Request instance
     * @param Response   $response  A Response instance
     * @param \Exception $exception An Exception instance
     */
    function collect(Request $request, Response $response, \Exception $exception = null);

    /**
     * Returns the name of the collector.
     *
     * @return string The collector name
     */
    function getName();
}
```

The `getName()` method must return a unique name. This is used to access the information later on (see *How to use the Profiler in a Functional Test* for instance).

The `collect()` method is responsible for storing the data it wants to give access to in local properties.

1. http://api.symfony.com/2.0/Symfony/Component/HttpKernel/DataCollector/DataCollectorInterface.html

As the profiler serializes data collector instances, you should not store objects that cannot be serialized (like PDO objects), or you need to provide your own `serialize()` method.

Most of the time, it is convenient to extend *`DataCollector`*[2] and populate the `$this->data` property (it takes care of serializing the `$this->data` property):

Listing 66-2

```
class MemoryDataCollector extends DataCollector
{
    public function collect(Request $request, Response $response, \Exception $exception = null)
    {
        $this->data = array(
            'memory' => memory_get_peak_usage(true),
        );
    }

    public function getMemory()
    {
        return $this->data['memory'];
    }

    public function getName()
    {
        return 'memory';
    }
}
```

Enabling Custom Data Collectors

To enable a data collector, add it as a regular service in one of your configuration, and tag it with `data_collector`:

Listing 66-3

```
services:
    data_collector.your_collector_name:
        class: Fully\Qualified\Collector\Class\Name
        tags:
            - { name: data_collector }
```

Adding Web Profiler Templates

When you want to display the data collected by your Data Collector in the web debug toolbar or the web profiler, create a Twig template following this skeleton:

Listing 66-4

```
{% extends 'WebProfilerBundle:Profiler:layout.html.twig' %}

{% block toolbar %}
    {# the web debug toolbar content #}
{% endblock %}

{% block head %}
    {# if the web profiler panel needs some specific JS or CSS files #}
{% endblock %}
```

2. http://api.symfony.com/2.0/Symfony/Component/HttpKernel/DataCollector/DataCollector.html

```
{% block menu %}
    {# the menu content #}
{% endblock %}

{% block panel %}
    {# the panel content #}
{% endblock %}
```

Each block is optional. The `toolbar` block is used for the web debug toolbar and `menu` and `panel` are used to add a panel to the web profiler.

All blocks have access to the `collector` object.

Built-in templates use a base64 encoded image for the toolbar (`<img src="src="data:image/png;base64,..."`). You can easily calculate the base64 value for an image with this little script: `echo base64_encode(file_get_contents($_SERVER['argv'][1]));`.

To enable the template, add a `template` attribute to the `data_collector` tag in your configuration. For example, assuming your template is in some `AcmeDebugBundle`:

Listing 66-5

```
services:
    data_collector.your_collector_name:
        class: Acme\DebugBundle\Collector\Class\Name
        tags:
            - { name: data_collector, template: "AcmeDebug:Collector:templatename", id:
"your_collector_name" }
```

Chapter 67

How to Create a SOAP Web Service in a Symfony2 Controller

Setting up a controller to act as a SOAP server is simple with a couple tools. You must, of course, have the *PHP SOAP*[1] extension installed. As the PHP SOAP extension can not currently generate a WSDL, you must either create one from scratch or use a 3rd party generator.

There are several SOAP server implementations available for use with PHP. *Zend SOAP*[2] and *NuSOAP*[3] are two examples. Although we use the PHP SOAP extension in our examples, the general idea should still be applicable to other implementations.

SOAP works by exposing the methods of a PHP object to an external entity (i.e. the person using the SOAP service). To start, create a class - `HelloService` - which represents the functionality that you'll expose in your SOAP service. In this case, the SOAP service will allow the client to call a method called `hello`, which happens to send an email:

Listing 67-1

```php
namespace Acme\SoapBundle;

class HelloService
{
    private $mailer;

    public function __construct(\Swift_Mailer $mailer)
    {
        $this->mailer = $mailer;
    }

    public function hello($name)
    {

        $message = \Swift_Message::newInstance()
```

1. `http://php.net/manual/en/book.soap.php`
2. `http://framework.zend.com/manual/en/zend.soap.server.html`
3. `http://sourceforge.net/projects/nusoap`

```
                                    ->setTo('me@example.com')
                                    ->setSubject('Hello Service')
                                    ->setBody($name . ' says hi!');

        $this->mailer->send($message);


        return 'Hello, ' . $name;
    }
}
```

Next, you can train Symfony to be able to create an instance of this class. Since the class sends an e-mail, it's been designed to accept a `Swift_Mailer` instance. Using the Service Container, we can configure Symfony to construct a `HelloService` object properly:

Listing 67-2

```
# app/config/config.yml
services:
    hello_service:
        class: Acme\DemoBundle\Services\HelloService
        arguments: [@mailer]
```

Below is an example of a controller that is capable of handling a SOAP request. If `indexAction()` is accessible via the route `/soap`, then the WSDL document can be retrieved via `/soap?wsdl`.

Listing 67-3

```
namespace Acme\SoapBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class HelloServiceController extends Controller
{
    public function indexAction()
    {
        $server = new \SoapServer('/path/to/hello.wsdl');
        $server->setObject($this->get('hello_service'));

        $response = new Response();
        $response->headers->set('Content-Type', 'text/xml; charset=ISO-8859-1');

        ob_start();
        $server->handle();
        $response->setContent(ob_get_clean());

        return $response;
    }
}
```

Take note of the calls to `ob_start()` and `ob_get_clean()`. These methods control *output buffering*[4] which allows you to "trap" the echoed output of `$server->handle()`. This is necessary because Symfony expects your controller to return a `Response` object with the output as its "content". You must also remember to set the "Content-Type" header to "text/xml", as this is what the client will expect. So, you use `ob_start()` to start buffering the STDOUT and use `ob_get_clean()` to dump the echoed output into the content of the Response and clear the output buffer. Finally, you're ready to return the `Response`.

Below is an example calling the service using *NuSOAP*[5] client. This example assumes that the `indexAction` in the controller above is accessible via the route `/soap`:

Listing 67-4

4. http://php.net/manual/en/book.outcontrol.php
5. http://sourceforge.net/projects/nusoap

```
$client = new \soapclient('http://example.com/app.php/soap?wsdl', true);

$result = $client->call('hello', array('name' => 'Scott'));
```

An example WSDL is below.

Listing 67-5

```
<?xml version="1.0" encoding="ISO-8859-1"?>
 <definitions xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
     xmlns:tns="urn:arnleadservicewsdl"
     xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
     xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
     xmlns="http://schemas.xmlsoap.org/wsdl/"
     targetNamespace="urn:helloservicewsdl">
  <types>
   <xsd:schema targetNamespace="urn:hellowsdl">
    <xsd:import namespace="http://schemas.xmlsoap.org/soap/encoding/" />
    <xsd:import namespace="http://schemas.xmlsoap.org/wsdl/" />
   </xsd:schema>
  </types>
  <message name="helloRequest">
   <part name="name" type="xsd:string" />
  </message>
  <message name="helloResponse">
   <part name="return" type="xsd:string" />
  </message>
  <portType name="hellowsdlPortType">
   <operation name="hello">
    <documentation>Hello World</documentation>
    <input message="tns:helloRequest"/>
    <output message="tns:helloResponse"/>
   </operation>
  </portType>
  <binding name="hellowsdlBinding" type="tns:hellowsdlPortType">
  <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="hello">
   <soap:operation soapAction="urn:arnleadservicewsdl#hello" style="rpc"/>
   <input>
    <soap:body use="encoded" namespace="urn:hellowsdl"
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
   </input>
   <output>
    <soap:body use="encoded" namespace="urn:hellowsdl"
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
   </output>
  </operation>
 </binding>
 <service name="hellowsdl">
  <port name="hellowsdlPort" binding="tns:hellowsdlBinding">
   <soap:address location="http://example.com/app.php/soap" />
  </port>
 </service>
</definitions>
```

Chapter 68

How Symfony2 differs from symfony1

The Symfony2 framework embodies a significant evolution when compared with the first version of the framework. Fortunately, with the MVC architecture at its core, the skills used to master a symfony1 project continue to be very relevant when developing in Symfony2. Sure, `app.yml` is gone, but routing, controllers and templates all remain.

In this chapter, we'll walk through the differences between symfony1 and Symfony2. As you'll see, many tasks are tackled in a slightly different way. You'll come to appreciate these minor differences as they promote stable, predictable, testable and decoupled code in your Symfony2 applications.

So, sit back and relax as we take you from "then" to "now".

Directory Structure

When looking at a Symfony2 project - for example, the *Symfony2 Standard*[1] - you'll notice a very different directory structure than in symfony1. The differences, however, are somewhat superficial.

The `app/` Directory

In symfony1, your project has one or more applications, and each lives inside the `apps/` directory (e.g. `apps/frontend`). By default in Symfony2, you have just one application represented by the `app/` directory. Like in symfony1, the `app/` directory contains configuration specific to that application. It also contains application-specific cache, log and template directories as well as a `Kernel` class (`AppKernel`), which is the base object that represents the application.

Unlike symfony1, almost no PHP code lives in the `app/` directory. This directory is not meant to house modules or library files as it did in symfony1. Instead, it's simply the home of configuration and other resources (templates, translation files).

The `src/` Directory

Put simply, your actual code goes here. In Symfony2, all actual application-code lives inside a bundle (roughly equivalent to a symfony1 plugin) and, by default, each bundle lives inside the `src` directory.

1. `https://github.com/symfony/symfony-standard`

In that way, the `src` directory is a bit like the `plugins` directory in symfony1, but much more flexible. Additionally, while *your* bundles will live in the `src/` directory, third-party bundles may live in the `vendor/bundles/` directory.

To get a better picture of the `src/` directory, let's first think of a symfony1 application. First, part of your code likely lives inside one or more applications. Most commonly these include modules, but could also include any other PHP classes you put in your application. You may have also created a `schema.yml` file in the `config` directory of your project and built several model files. Finally, to help with some common functionality, you're using several third-party plugins that live in the `plugins/` directory. In other words, the code that drives your application lives in many different places.

In Symfony2, life is much simpler because *all* Symfony2 code must live in a bundle. In our pretend symfony1 project, all the code *could* be moved into one or more plugins (which is a very good practice, in fact). Assuming that all modules, PHP classes, schema, routing configuration, etc were moved into a plugin, the symfony1 `plugins/` directory would be very similar to the Symfony2 `src/` directory.

Put simply again, the `src/` directory is where your code, assets, templates and most anything else specific to your project will live.

The `vendor/` Directory

The `vendor/` directory is basically equivalent to the `lib/vendor/` directory in symfony1, which was the conventional directory for all vendor libraries and bundles. By default, you'll find the Symfony2 library files in this directory, along with several other dependent libraries such as Doctrine2, Twig and Swiftmailer. 3rd party Symfony2 bundles usually live in the `vendor/bundles/`.

The `web/` Directory

Not much has changed in the `web/` directory. The most noticeable difference is the absence of the `css/`, `js/` and `images/` directories. This is intentional. Like with your PHP code, all assets should also live inside a bundle. With the help of a console command, the `Resources/public/` directory of each bundle is copied or symbolically-linked to the `web/bundles/` directory. This allows you to keep assets organized inside your bundle, but still make them available to the public. To make sure that all bundles are available, run the following command:

Listing 68-1
```
php app/console assets:install web
```

This command is the Symfony2 equivalent to the symfony1 `plugin:publish-assets` command.

Autoloading

One of the advantages of modern frameworks is never needing to worry about requiring files. By making use of an autoloader, you can refer to any class in your project and trust that it's available. Autoloading has changed in Symfony2 to be more universal, faster, and independent of needing to clear your cache.

In symfony1, autoloading was done by searching the entire project for the presence of PHP class files and caching this information in a giant array. That array told symfony1 exactly which file contained each class. In the production environment, this caused you to need to clear the cache when classes were added or moved.

In Symfony2, a new class - `UniversalClassLoader` - handles this process. The idea behind the autoloader is simple: the name of your class (including the namespace) must match up with the path to the file containing that class. Take the FrameworkExtraBundle from the Symfony2 Standard Edition as an example:

Listing 68-2

```
namespace Sensio\Bundle\FrameworkExtraBundle;

use Symfony\Component\HttpKernel\Bundle\Bundle;
// ...

class SensioFrameworkExtraBundle extends Bundle
{
    // ...
```

The file itself lives at vendor/bundle/Sensio/Bundle/FrameworkExtraBundle/SensioFrameworkExtraBundle.php. As you can see, the location of the file follows the namespace of the class. Specifically, the namespace, Sensio\Bundle\FrameworkExtraBundle, spells out the directory that the file should live in (vendor/bundle/Sensio/Bundle/FrameworkExtraBundle). This is because, in the app/autoload.php file, you'll configure Symfony to look for the Sensio namespace in the vendor/bundle directory:

Listing 68-3

```
// app/autoload.php

// ...
$loader->registerNamespaces(array(
    // ...
    'Sensio'           => __DIR__.'/../vendor/bundles',
));
```

If the file did *not* live at this exact location, you'd receive a Class "Sensio\Bundle\FrameworkExtraBundle\SensioFrameworkExtraBundle" does not exist. error. In Symfony2, a "class does not exist" means that the suspect class namespace and physical location do not match. Basically, Symfony2 is looking in one exact location for that class, but that location doesn't exist (or contains a different class). In order for a class to be autoloaded, you **never need to clear your cache** in Symfony2.

As mentioned before, for the autoloader to work, it needs to know that the Sensio namespace lives in the vendor/bundles directory and that, for example, the Doctrine namespace lives in the vendor/doctrine/lib/ directory. This mapping is entirely controlled by you via the app/autoload.php file.

If you look at the HelloController from the Symfony2 Standard Edition you can see that it lives in the Acme\DemoBundle\Controller namespace. Yet, the Acme namespace is not defined in the app/autoload.php. By default you do not need to explicitly configure the location of bundles that live in the src/ directory. The UniversalClassLoader is configured to fallback to the src/ directory using its registerNamespaceFallbacks method:

Listing 68-4

```
// app/autoload.php

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

Using the Console

In symfony1, the console is in the root directory of your project and is called symfony:

Listing 68-5

```
php symfony
```

In Symfony2, the console is now in the app sub-directory and is called `console`:

Listing 68-6
```
php app/console
```

Applications

In a symfony1 project, it is common to have several applications: one for the frontend and one for the backend for instance.

In a Symfony2 project, you only need to create one application (a blog application, an intranet application, ...). Most of the time, if you want to create a second application, you might instead create another project and share some bundles between them.

And if you need to separate the frontend and the backend features of some bundles, you can create sub-namespaces for controllers, sub-directories for templates, different semantic configurations, separate routing configurations, and so on.

Of course, there's nothing wrong with having multiple applications in your project, that's entirely up to you. A second application would mean a new directory, e.g. `my_app/`, with the same basic setup as the `app/` directory.

Read the definition of a *Project*, an *Application*, and a *Bundle* in the glossary.

Bundles and Plugins

In a symfony1 project, a plugin could contain configuration, modules, PHP libraries, assets and anything else related to your project. In Symfony2, the idea of a plugin is replaced by the "bundle". A bundle is even more powerful than a plugin because the core Symfony2 framework is brought in via a series of bundles. In Symfony2, bundles are first-class citizens that are so flexible that even core code itself is a bundle.

In symfony1, a plugin must be enabled inside the `ProjectConfiguration` class:

Listing 68-7
```
// config/ProjectConfiguration.class.php
public function setup()
{
    $this->enableAllPluginsExcept(array(/* some plugins here */));
}
```

In Symfony2, the bundles are activated inside the application kernel:

Listing 68-8
```
// app/AppKernel.php
public function registerBundles()
{
    $bundles = array(
        new Symfony\Bundle\FrameworkBundle\FrameworkBundle(),
        new Symfony\Bundle\TwigBundle\TwigBundle(),
        // ...
        new Acme\DemoBundle\AcmeDemoBundle(),
    );

    return $bundles;
}
```

Routing (`routing.yml`) and Configuration (`config.yml`)

In symfony1, the `routing.yml` and `app.yml` configuration files were automatically loaded inside any plugin. In Symfony2, routing and application configuration inside a bundle must be included manually. For example, to include a routing resource from a bundle called `AcmeDemoBundle`, you can do the following:

Listing 68-9

```
# app/config/routing.yml
_hello:
    resource: "@AcmeDemoBundle/Resources/config/routing.yml"
```

This will load the routes found in the `Resources/config/routing.yml` file of the `AcmeDemoBundle`. The special `@AcmeDemoBundle` is a shortcut syntax that, internally, resolves to the full path to that bundle.

You can use this same strategy to bring in configuration from a bundle:

Listing 68-10

```
# app/config/config.yml
imports:
    - { resource: "@AcmeDemoBundle/Resources/config/config.yml" }
```

In Symfony2, configuration is a bit like `app.yml` in symfony1, except much more systematic. With `app.yml`, you could simply create any keys you wanted. By default, these entries were meaningless and depended entirely on how you used them in your application:

Listing 68-11

```
# some app.yml file from symfony1
all:
  email:
    from_address:  foo.bar@example.com
```

In Symfony2, you can also create arbitrary entries under the `parameters` key of your configuration:

Listing 68-12

```
parameters:
    email.from_address: foo.bar@example.com
```

You can now access this from a controller, for example:

Listing 68-13

```
public function helloAction($name)
{
    $fromAddress = $this->container->getParameter('email.from_address');
}
```

In reality, the Symfony2 configuration is much more powerful and is used primarily to configure objects that you can use. For more information, see the chapter titled "*Service Container*".