Symfony

The Cookbook

for Symfony 2.3

generated on October 24, 2013

The Cookbook (2.3)

If you find typos or errors, feel free to report them by creating a ticket on the Symfony ticketing system (*http://github.com/symfony/symfony-docs/issues*). Based on tickets and users feedback, this book is continuously updated.

Contents at a Glance

How to Create and store a Symfony2 Project in git 5
How to Create and store a Symfony2 Project in Subversion 9
How to customize Error Pages 13
How to define Controllers as Services 15
How to force routes to always use HTTPS or HTTP 19
How to allow a "/" character in a route parameter 20
How to configure a redirect to another route without a custom controller 21
How to use HTTP Methods beyond GET and POST in Routes 22
How to use Service Container Parameters in your Routes 24
How to create a custom Route Loader 25
How to Use Assetic for Asset Management 29
How to Minify CSS/JS Files (using UglifyJs and UglifyCss) 34
How to Minify JavaScripts and Stylesheets with YUI Compressor 38
How to Use Assetic For Image Optimization with Twig Functions 40
How to Apply an Assetic Filter to a Specific File Extension 43
How to handle File Uploads with Doctrine 45
How to use Doctrine Extensions: Timestampable, Sluggable, Translatable, etc. 54
How to Register Event Listeners and Subscribers 55
How to use Doctrine's DBAL Layer 58
How to generate Entities from an Existing Database 60
How to work with Multiple Entity Managers and Connections 64
How to Register Custom DQL Functions 67
How to Define Relationships with Abstract Classes and Interfaces 68
How to provide model classes for several Doctrine implementations 71
How to implement a simple Registration Form 74
How to customize Form Rendering 80
How to use Data Transformers 92
How to Dynamically Modify Forms Using Form Events 98
How to Embed a Collection of Forms 107
How to Create a Custom Form Field Type 120
How to Create a Form Type Extension 125
How to Reduce Code Duplication with "inherit_data" 130
How to Unit Test your Forms 133
How to configure Empty Data for a Form Class 138
How to use the submit() Function to handle Form Submissions 140
How to use the Virtual Form Field Option 143

How to create a Custom Validation Constraint 144
How to Master and Create new Environments 147
How to override Symfony's Default Directory Structure 152
Understanding how the Front Controller, Kernel and Environments work together 155
How to Set External Parameters in the Service Container 158
How to use PdoSessionHandler to store Sessions in the Database 161
How to use the Apache Router 164
Configuring a web server 166
How to use the Serializer 168
How to create an Event Listener 170
How to work with Scopes 173
How to work with Compiler Passes in Bundles 178
Session Proxy Examples 179
Making the Locale "Sticky" during a User's Session 181
Configuring the Directory where Sessions Files are Saved 183
Bridge a legacy application with Symfony Sessions 184
How to create a custom Data Collector 186
How to use Matchers to enable the Profiler Conditionally 189
How to Create a SOAP Web Service in a Symfony2 Controller 191
How Symfony2 differs from symfony1 195
How to deploy a Symfony2 application 201

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 `composer.json` 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.yml
```

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

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

6. Add all of the initial files to git:

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

7. Create an initial commit with your started project:

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

8. Finally, download all of the third-party vendor libraries by executing composer. For details, see *Updating Vendors*.

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.

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 in the "*How to remove the AcmeDemoBundle*" article.

Managing Vendor Libraries with composer.json

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 composer.phar install` "downloader" binary. This `composer.phar` file is from a library called *Composer*[4] and you can read more about installing it in the *Installation* chapter.

The `composer.phar` file reads from the `composer.json` file at the root of your project. This is an JSON-formatted file, which holds a list of each of the external packages you need, the version to be downloaded and more. The `composer.phar` file also reads from a `composer.lock` file, which allows you to pin each library to an **exact** version. In fact, if a `composer.lock` file exists, the versions inside will override those in `composer.json`. To upgrade your libraries to new versions, run `php composer.phar update`.

3. https://help.github.com/articles/ignoring-files
4. http://getcomposer.org/

If you want to add a new package to your application, modify the `composer.json` file:

Listing 1-5
```
{
    "require": {
        ...
        "doctrine/doctrine-fixtures-bundle": "@dev"
    }
}
```

and then execute the `update` command for this specific package, i.e.:

Listing 1-6
```
1 $ php composer.phar update doctrine/doctrine-fixtures-bundle
```

You can also combine both steps into a single command:

Listing 1-7
```
1 $ php composer.phar require doctrine/doctrine-fixtures-bundle:@dev
```

To learn more about Composer, see *GetComposer.org*[5]:

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/`. But since all the information needed to download these files is saved in `composer.json` and `composer.lock` (which *are* stored in the repository), any other developer can use the project, run `php composer.phar 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 composer.phar 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 `composer.json` and `composer.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 `composer.json`, be sure to replace only the original parts (i.e. be sure not to also delete any of your custom entries).

Vendors and Submodules

Instead of using the `composer.json` system for managing your vendor libraries, you may instead choose to use native *git submodules*[6]. There is nothing wrong with this approach, though the `composer.json` system is the official way to solve this problem and probably much easier to deal with. Unlike git submodules, `Composer` is smart enough to calculate which libraries depend on which other libraries.

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.

5. `http://getcomposer.org/`
6. `http://git-scm.com/book/en/Git-Tools-Submodules`

The easiest way to store your project on a remote server is via *GitHub*[7]. 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*[8] and then pushing to it. One library that helps manage this is *Gitolite*[9].

7. https://github.com/
8. http://git-scm.com/book/en/Git-Basics-Getting-a-Git-Repository
9. 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 it's assumed that your repository layout follows the widespread standard structure:

Listing 2-1

```
1 myproject/
2     branches/
3     tags/
4     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 specific files can be ignored.

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.yml" 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.yml, 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.yml` to `app/config/parameters.yml.dist`. The `parameters.yml` file is ignored by svn (see above) so that machine-specific settings like database passwords aren't committed. By creating the `parameters.yml.dist` file, new developers can quickly clone the project, copy this file to `parameters.yml`, customize it, and start developing.

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

8. Finally, download all of the third-party vendor libraries by executing composer. For details, see *Updating Vendors*.

If you rely on any "dev" versions, then git may be used to install those libraries, since there is no archive available for download.

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 in the "*How to remove the AcmeDemoBundle*" article.

Managing Vendor Libraries with composer.json

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 composer.phar install` "downloader" binary. This `composer.phar` file is from a library called *Composer*[8] and you can read more about installing it in the *Installation* chapter.

The `composer.phar` file reads from the `composer.json` file at the root of your project. This is an JSON-formatted file, which holds a list of each of the external packages you need, the version to be downloaded and more. The `composer.phar` file also reads from a `composer.lock` file, which allows you to pin each library to an **exact** version. In fact, if a `composer.lock` file exists, the versions inside will override those in `composer.json`. To upgrade your libraries to new versions, run `php composer.phar update`.

If you want to add a new package to your application, modify the `composer.json` file:

Listing 2-6
```
{
    "require": {
        ...
        "doctrine/doctrine-fixtures-bundle": "@dev"
    }
}
```

and then execute the `update` command for this specific package, i.e.:

Listing 2-7
```
1 $ php composer.phar update doctrine/doctrine-fixtures-bundle
```

You can also combine both steps into a single command:

Listing 2-8
```
1 $ php composer.phar require doctrine/doctrine-fixtures-bundle:@dev
```

8. `http://getcomposer.org/`

To learn more about Composer, see *GetComposer.org*[9]:

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/`. But since all the information needed to download these files is saved in `composer.json` and `composer.lock` (which *are* stored in the repository), any other developer can use the project, run `php composer.phar 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 composer.phar 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 `composer.json` and `composer.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 `composer.json`, be sure to replace only the original parts (i.e. be sure not to also delete any of your custom entries).

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.

9. http://getcomposer.org/
10. http://git-scm.com/
11. http://subversion.apache.org/
12. https://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 `twig.controller.exception:showAction` with your own controller and handle it however you want (see *exception_controller in the Twig reference*). The default exception controller is registered as a service - the actual class is `Symfony\Bundle\TwigBundle\Controller\ExceptionController`.

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, 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>
```

You **must not** use `is_granted` in your error pages (or layout used by your error pages), because the router runs before the firewall. If the router throws an exception (for instance, when the route does not match), then using `is_granted` will throw a further exception. You can use `is_granted` safely by saying `{% if app.user and is_granted('...') %}`.

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/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.

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.

A second advantage is that your controllers are more "sandboxed". By looking at the constructor arguments, it's easy to see what types of things this controller may or may not do. And because each dependency needs to be injected manually, it's more obvious (i.e. if you have many constructor arguments) when your controller has become too big, and may need to be split into multiple controllers.

So, even if you don't specify your controllers as services, you'll likely see this done in some open-source Symfony2 bundles. It's also important to understand the pros and cons of both approaches.

Defining the Controller as a Service

A controller can be defined as a service in the same way as any other class. For example, if you have the following simple controller:

Listing 4-1

```
1  // src/Acme/HelloBundle/Controller/HelloController.php
2  namespace Acme\HelloBundle\Controller;
3
4  use Symfony\Component\HttpFoundation\Response;
5
6  class HelloController
7  {
8      public function indexAction($name)
9      {
```

1. http://api.symfony.com/2.3/Symfony/Bundle/FrameworkBundle/Controller/Controller.html

```
10          return new Response('<html><body>Hello '.$name.'!</body></html>');
11      }
12  }
```

Then you can define it as a service as follows:

Listing 4-2
```
1  # src/Acme/HelloBundle/Resources/config/services.yml
2  parameters:
3      # ...
4      acme.controller.hello.class: Acme\HelloBundle\Controller\HelloController
5
6  services:
7      acme.hello.controller:
8          class:     "%acme.controller.hello.class%"
```

Referring to the service

To refer to a controller that's defined as a service, use the single colon (:) notation. For example, to forward to the `indexAction()` method of the service defined above with the id `acme.hello.controller`:

Listing 4-3
```
1  $this->forward('acme.hello.controller:indexAction');
```

You cannot drop the `Action` part of the method name when using this syntax.

You can also route to the service by using the same notation when defining the route `_controller` value:

Listing 4-4
```
1  # app/config/routing.yml
2  hello:
3      pattern:      /hello
4      defaults:     { _controller: acme.hello.controller:indexAction }
```

You can also use annotations to configure routing using a controller defined as a service. See the *FrameworkExtraBundle documentation* for details.

Alternatives to Base Controller Methods

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 source code*[2] is a great source on how to perform many common tasks.

For example, if you want to render a template instead of creating the `Response` object directly, then your code would look like this if you were extending Symfony's base controller:

2. https://github.com/symfony/symfony/blob/master/src/Symfony/Bundle/FrameworkBundle/Controller/Controller.php

Listing 4-5
```php
// src/Acme/HelloBundle/Controller/HelloController.php
namespace Acme\HelloBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class HelloController extends Controller
{
    public function indexAction($name)
    {
        return $this->render(
            'AcmeHelloBundle:Hello:index.html.twig',
            array('name' => $name)
        );
    }
}
```

If you look at the source code for the `render` function in Symfony's *base Controller class*[3], you'll see that this method actually uses the `templating` service:

Listing 4-6
```php
public function render($view, array $parameters = array(), Response $response = null)
{
    return $this->container->get('templating')->renderResponse($view, $parameters,
$response);
}
```

In a controller that's defined as a service, you can instead inject the `templating` service and use it directly:

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

use Symfony\Bundle\FrameworkBundle\Templating\EngineInterface;
use Symfony\Component\HttpFoundation\Response;

class HelloController
{
    private $templating;

    public function __construct(EngineInterface $templating)
    {
        $this->templating = $templating;
    }

    public function indexAction($name)
    {
        return $this->templating->renderResponse(
            'AcmeHelloBundle:Hello:index.html.twig',
            array('name' => $name)
        );
    }
}
```

The service definition also needs modifying to specify the constructor argument:

Listing 4-8
```yaml
# src/Acme/HelloBundle/Resources/config/services.yml
parameters:
```

3. https://github.com/symfony/symfony/blob/master/src/Symfony/Bundle/FrameworkBundle/Controller/Controller.php

```
    # ...
    acme.controller.hello.class: Acme\HelloBundle\Controller\HelloController

services:
    acme.hello.controller:
        class:     "%acme.controller.hello.class%"
        arguments: ["@templating"]
```

Rather than fetching the `templating` service from the container, you can inject *only* the exact service(s) that you need directly into the controller.

This does not mean that you cannot extend these controllers from your own base controller. The move away from the standard base controller is because its helper methods rely on having the container available which is not the case for controllers that are defined as services. It may be a good idea to extract common code into a service that's injected rather than place that code into a base controller that you extend. Both approaches are valid, exactly how you want to organize your reusable code is up to you.

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 schemes:

Listing 5-1
```yaml
secure:
    path:     /secure
    defaults: { _controller: AcmeDemoBundle:Main:secure }
    schemes:  [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
```twig
{# 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 path: `[^/]+`. This means that all characters are allowed except `/`.

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

Listing 6-1

```
1 _hello:
2     path:     /hello/{name}
3     defaults: { _controller: AcmeDemoBundle:Demo:hello }
4     requirements:
5         name: ".+"
```

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

Chapter 7

How to configure a redirect to another route without a custom controller

This guide explains how to configure a redirect from one route to another without using a custom controller.

Assume that there is no useful default controller for the `/` path of your application and you want to redirect these requests to `/app`.

Your configuration will look like this:

Listing 7-1

```yaml
AppBundle:
    resource: "@App/Controller/"
    type:     annotation
    prefix:   /app

root:
    path:     /
    defaults:
        _controller: FrameworkBundle:Redirect:urlRedirect
        path: /app
        permanent: true
```

In this example, you configure a route for the `/` path and let *`RedirectController`*[1] handle it. This controller comes standard with Symfony and offers two actions for redirecting request:

- `urlRedirect` redirects to another *path*. You must provide the `path` parameter containing the path of the resource you want to redirect to.
- `redirect` (not shown here) redirects to another *route*. You must provide the `route` parameter with the *name* of the route you want to redirect to.

The `permanent` switch tells both methods to issue a 301 HTTP status code instead of the default `302` status code.

1. http://api.symfony.com/2.3/Symfony/Bundle/FrameworkBundle/Controller/RedirectController.html

Chapter 8

How to use HTTP Methods beyond GET and POST in Routes

The HTTP method of a request is one of the requirements that can be checked when seeing if it matches a route. This is introduced in the routing chapter of the book "*Routing*" with examples using GET and POST. You can also use other HTTP verbs in this way. For example, if you have a blog post entry then you could use the same URL path to show it, make changes to it and delete it by matching on GET, PUT and DELETE.

Listing 8-1

```
1  blog_show:
2      path:     /blog/{slug}
3      defaults: { _controller: AcmeDemoBundle:Blog:show }
4      methods:   [GET]
5
6  blog_update:
7      path:     /blog/{slug}
8      defaults: { _controller: AcmeDemoBundle:Blog:update }
9      methods:   [PUT]
10
11 blog_delete:
12     path:     /blog/{slug}
13     defaults: { _controller: AcmeDemoBundle:Blog:delete }
14     methods:   [DELETE]
```

Faking the Method with _method

The `_method` functionality shown here is disabled by default in Symfony 2.2 and enabled by default in Symfony 2.3. To control it in Symfony 2.2, you must call *`Request::enableHttpMethodParameterOverride`*[1] before you handle the request (e.g. in your front controller). In Symfony 2.3, use the *http_method_override* option.

Unfortunately, life isn't quite this simple, since most browsers do not support sending PUT and DELETE requests. Fortunately Symfony2 provides you with a simple way of working around this limitation. By including a `_method` parameter in the query string or parameters of an HTTP request, Symfony2 will use this as the method when matching routes. Forms automatically include a hidden field for this parameter if their submission method is not GET or POST. See *the related chapter in the forms documentation* for more information.

1. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Request.html#enableHttpMethodParameterOverride()

Chapter 9

How to use Service Container Parameters in your Routes

Sometimes you may find it useful to make some parts of your routes globally configurable. For instance, if you build an internationalized site, you'll probably start with one or two locales. Surely you'll add a requirement to your routes to prevent a user from matching a locale other than the locales your support.

You *could* hardcode your `_locale` requirement in all your routes. But a better solution is to use a configurable service container parameter right inside your routing configuration:

Listing 9-1
```
contact:
    path:     /{_locale}/contact
    defaults: { _controller: AcmeDemoBundle:Main:contact }
    requirements:
        _locale: %acme_demo.locales%
```

You can now control and set the `acme_demo.locales` parameter somewhere in your container:

Listing 9-2
```
# app/config/config.yml
parameters:
    acme_demo.locales: en|es
```

You can also use a parameter to define your route path (or part of your path):

Listing 9-3
```
some_route:
    path:     /%acme_demo.route_prefix%/contact
    defaults: { _controller: AcmeDemoBundle:Main:contact }
```

Just like in normal service container configuration files, if you actually need a `%` in your route, you can escape the percent sign by doubling it, e.g. `/score-50%%`, which would resolve to `/score-50%`.

However, as the `%` characters included in any URL are automatically encoded, the resulting URL of this example would be `/score-50%25` (`%25` is the result of encoding the `%` character).

Chapter 10

How to create a custom Route Loader

A custom route loader allows you to add routes to an application without including them, for example, in a Yaml file. This comes in handy when you have a bundle but don't want to manually add the routes for the bundle to `app/config/routing.yml`. This may be especially important when you want to make the bundle reusable, or when you have open-sourced it as this would slow down the installation process and make it error-prone.

Alternatively, you could also use a custom route loader when you want your routes to be automatically generated or located based on some convention or pattern. One example is the *FOSRestBundle*[1] where routing is generated based off the names of the action methods in a controller.

There are many bundles out there that use their own route loaders to accomplish cases like those described above, for instance *FOSRestBundle*[2], *KnpRadBundle*[3] and *SonataAdminBundle*[4].

Loading Routes

The routes in a Symfony application are loaded by the *`DelegatingLoader`*[5]. This loader uses several other loaders (delegates) to load resources of different types, for instance Yaml files or `@Route` and `@Method` annotations in controller files. The specialized loaders implement *`LoaderInterface`*[6] and therefore have two important methods: *`supports()`*[7] and *`load()`*[8].

Take these lines from the `routing.yml` in the AcmeDemoBundle of the Standard Edition:

Listing 10-1

1. https://github.com/FriendsOfSymfony/FOSRestBundle
2. https://github.com/FriendsOfSymfony/FOSRestBundle
3. https://github.com/KnpLabs/KnpRadBundle
4. https://github.com/sonata-project/SonataAdminBundle
5. http://api.symfony.com/2.3/Symfony/Bundle/FrameworkBundle/Routing/DelegatingLoader.html
6. http://api.symfony.com/2.3/Symfony/Component/Config/Loader/LoaderInterface.html
7. http://api.symfony.com/2.3/Symfony/Component/Config/Loader/LoaderInterface.html#supports()
8. http://api.symfony.com/2.3/Symfony/Component/Config/Loader/LoaderInterface.html#load()

```yaml
# src/Acme/DemoBundle/Resources/config/routing.yml
_demo:
    resource: "@AcmeDemoBundle/Controller/DemoController.php"
    type:     annotation
    prefix:   /demo
```

When the main loader parses this, it tries all the delegate loaders and calls their *supports()*[9] method with the given resource (`@AcmeDemoBundle/Controller/DemoController.php`) and type (`annotation`) as arguments. When one of the loader returns `true`, its *load()*[10] method will be called, which should return a *RouteCollection*[11] containing *Route*[12] objects.

Creating a Custom Loader

To load routes from some custom source (i.e. from something other than annotations, Yaml or XML files), you need to create a custom route loader. This loader should implement *LoaderInterface*[13].

The sample loader below supports loading routing resources with a type of `extra`. The type `extra` isn't important - you can just invent any resource type you want. The resource name itself is not actually used in the example:

Listing 10-2

```php
namespace Acme\DemoBundle\Routing;

use Symfony\Component\Config\Loader\LoaderInterface;
use Symfony\Component\Config\Loader\LoaderResolverInterface;
use Symfony\Component\Routing\Route;
use Symfony\Component\Routing\RouteCollection;

class ExtraLoader implements LoaderInterface
{
    private $loaded = false;

    public function load($resource, $type = null)
    {
        if (true === $this->loaded) {
            throw new \RuntimeException('Do not add the "extra" loader twice');
        }

        $routes = new RouteCollection();

        // prepare a new route
        $pattern = '/extra/{parameter}';
        $defaults = array(
            '_controller' => 'AcmeDemoBundle:Demo:extra',
        );
        $requirements = array(
            'parameter' => '\d+',
        );
        $route = new Route($pattern, $defaults, $requirements);

        // add the new route to the route collection:
```

9. http://api.symfony.com/2.3/Symfony/Component/Config/Loader/LoaderInterface.html#supports()
10. http://api.symfony.com/2.3/Symfony/Component/Config/Loader/LoaderInterface.html#load()
11. http://api.symfony.com/2.3/Symfony/Component/Routing/RouteCollection.html
12. http://api.symfony.com/2.3/Symfony/Component/Routing/Route.html
13. http://api.symfony.com/2.3/Symfony/Component/Config/Loader/LoaderInterface.html

```
        $routeName = 'extraRoute';
        $routes->add($routeName, $route);

        $this->loaded = true;

        return $routes;
    }

    public function supports($resource, $type = null)
    {
        return 'extra' === $type;
    }

    public function getResolver()
    {
        // needed, but can be blank, unless you want to load other resources
        // and if you do, using the Loader base class is easier (see below)
    }

    public function setResolver(LoaderResolverInterface $resolver)
    {
        // same as above
    }
}
```

Make sure the controller you specify really exists.

Now define a service for the `ExtraLoader`:

Listing 10-3
```
services:
    acme_demo.routing_loader:
        class: Acme\DemoBundle\Routing\ExtraLoader
        tags:
            - { name: routing.loader }
```

Notice the tag `routing.loader`. All services with this tag will be marked as potential route loaders and added as specialized routers to the *`DelegatingLoader`*[14].

Using the Custom Loader

If you did nothing else, your custom routing loader would *not* be called. Instead, you only need to add a few extra lines to the routing configuration:

Listing 10-4
```
# app/config/routing.yml
AcmeDemoBundle_Extra:
    resource: .
    type: extra
```

The important part here is the `type` key. Its value should be "extra". This is the type which our `ExtraLoader` supports and this will make sure its `load()` method gets called. The `resource` key is insignificant for the `ExtraLoader`, so we set it to ".".

14. http://api.symfony.com/2.3/Symfony/Bundle/FrameworkBundle/Routing/DelegatingLoader.html

The routes defined using custom route loaders will be automatically cached by the framework. So whenever you change something in the loader class itself, don't forget to clear the cache.

More Advanced Loaders

In most cases it's better not to implement *LoaderInterface*[15] yourself, but extend from *Loader*[16]. This class knows how to use a *LoaderResolver*[17] to load secondary routing resources.

Of course you still need to implement *supports()*[18] and *load()*[19]. Whenever you want to load another resource - for instance a Yaml routing configuration file - you can call the *import()*[20] method:

Listing 10-5

```php
namespace Acme\DemoBundle\Routing;

use Symfony\Component\Config\Loader\Loader;
use Symfony\Component\Routing\RouteCollection;

class AdvancedLoader extends Loader
{
    public function load($resource, $type = null)
    {
        $collection = new RouteCollection();

        $resource = '@AcmeDemoBundle/Resources/config/import_routing.yml';
        $type = 'yaml';

        $importedRoutes = $this->import($resource, $type);

        $collection->addCollection($importedRoutes);

        return $collection;
    }

    public function supports($resource, $type = null)
    {
        return $type === 'advanced_extra';
    }
}
```

The resource name and type of the imported routing configuration can be anything that would normally be supported by the routing configuration loader (Yaml, XML, PHP, annotation, etc.).

15. http://api.symfony.com/2.3/Symfony/Component/Config/Loader/LoaderInterface.html
16. http://api.symfony.com/2.3/Symfony/Component/Config/Loader/Loader.html
17. http://api.symfony.com/2.3/Symfony/Component/Config/Loader/LoaderResolver.html
18. http://api.symfony.com/2.3/Symfony/Component/Config/Loader/LoaderInterface.html#supports()
19. http://api.symfony.com/2.3/Symfony/Component/Config/Loader/LoaderInterface.html#load()
20. http://api.symfony.com/2.3/Symfony/Component/Config/Loader/Loader.html#import()

Chapter 11

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 11-1
```
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. This 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.

You can use Assetic to process both *CSS stylesheets* and *JavaScript files*. The philosophy behind adding either is basically the same, but with a slightly different syntax.

Including JavaScript Files

To include JavaScript files, use the `javascript` tag in any template. This will most commonly live in the `javascripts` block, if you're using the default block names from the Symfony Standard Distribution:

Listing 11-2
```
1 {% javascripts '@AcmeFooBundle/Resources/public/js/*' %}
2     <script type="text/javascript" src="{{ asset_url }}"></script>
3 {% endjavascripts %}
```

You can also include CSS Stylesheets: see *Including CSS Stylesheets*.

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 11-3
```
1 <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 *will* cause problems with CSS files that reference images by their relative path. See *Fixing CSS Paths with the cssrewrite Filter*.

Including CSS Stylesheets

To bring in CSS stylesheets, you can use the same methodologies seen above, except with the `stylesheets` tag. If you're using the default block names from the Symfony Standard Distribution, this will usually live inside a `stylesheets` block:

Listing 11-4
```
1 {% stylesheets 'bundles/acme_foo/css/*' filter='cssrewrite' %}
2     <link rel="stylesheet" href="{{ asset_url }}" />
3 {% endstylesheets %}
```

But because Assetic changes the paths to your assets, this *will* break any background images (or other paths) that uses relative paths, unless you use the *cssrewrite* filter.

Notice that in the original example that included JavaScript files, you referred to the files using a path like `@AcmeFooBundle/Resources/public/file.js`, but that in this example, you referred to the CSS files using their actual, publicly-accessible path: `bundles/acme_foo/css`. You can use either, except that there is a known issue that causes the `cssrewrite` filter to fail when using the `@AcmeFooBundle` syntax for CSS Stylesheets.

Fixing CSS Paths with the `cssrewrite` Filter

Since Assetic generates new URLs for your assets, any relative paths inside your CSS files will break. To fix this, make sure to use the `cssrewrite` filter with your `stylesheets` tag. This parses your CSS files and corrects the paths internally to reflect the new location.

You can see an example in the previous section.

When using the `cssrewrite` filter, don't refer to your CSS files using the `@AcmeFooBundle` syntax. See the note in the above section for details.

Combining Assets

One feature of Assetic is that it will combine many 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 11-5
```
{% 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 (or more specifically, when the `debug` flag is `false`), this will be rendered as a single `script` tag, which contains the contents of all of the JavaScript files.

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 11-6
```
{% 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 (you'll use the filter below).

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

Listing 11-7
```
# 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 11-8
```
{% 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 11-9
```
{% 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 11-10
```
<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 11-11
```
$ 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 11-12
```
# 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 11-13
```
$ 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 11-14
```
$ 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 11-15
```
{% javascripts '@AcmeFooBundle/Resources/public/js/*' output='js/compiled/main.js' %}
    <script src="{{ asset_url }}"></script>
{% endjavascripts %}
```

Chapter 12

How to Minify CSS/JS Files (using UglifyJs and UglifyCss)

UglifyJs[1] is a javascript parser/compressor/beautifier toolkit. It can be used to combine and minify javascript assets so that they require less HTTP requests and make your site load faster. *UglifyCss*[2] is a css compressor/beautifier that is very similar to UglifyJs.

In this cookbook, the installation, configuration and usage of UglifyJs is shown in detail. `UglifyCss` works pretty much the same way and is only talked about briefly.

Install UglifyJs

UglifyJs is available as an *Node.js*[3] npm module and can be installed using npm. First, you need to *install node.js*[4]. Afterwards you can install UglifyJs using npm:

Listing 12-1

```
1 $ npm install -g uglify-js
```

This command will install UglifyJs globally and you may need to run it as a root user.

1. https://github.com/mishoo/UglifyJS
2. https://github.com/fmarcia/UglifyCSS
3. http://nodejs.org/
4. http://nodejs.org/

It's also possible to install UglifyJs inside your project only. To do this, install it without the `-g` option and specify the path where to put the module:

Listing 12-2
```
$ cd /path/to/symfony
$ mkdir app/Resources/node_modules
$ npm install uglify-js --prefix app/Resources
```

It is recommended that you install UglifyJs in your `app/Resources` folder and add the `node_modules` folder to version control. Alternatively, you can create an npm *package.json*[5] file and specify your dependencies there.

Depending on your installation method, you should either be able to execute the `uglifyjs` executable globally, or execute the physical file that lives in the `node_modules` directory:

Listing 12-3
```
$ uglifyjs --help

$ ./app/Resources/node_modules/.bin/uglifyjs --help
```

Configure the uglifyjs2 Filter

Now we need to configure Symfony2 to use the `uglifyjs2` filter when processing your javascripts:

Listing 12-4
```
# app/config/config.yml
assetic:
    filters:
        uglifyjs2:
            # the path to the uglifyjs executable
            bin: /usr/local/bin/uglifyjs
```

The path where UglifyJs is installed may vary depending on your system. To find out where npm stores the `bin` folder, you can use the following command:

Listing 12-5
```
$ npm bin -g
```

It should output a folder on your system, inside which you should find the UglifyJs executable.

If you installed UglifyJs locally, you can find the bin folder inside the `node_modules` folder. It's called `.bin` in this case.

You now have access to the `uglifyjs2` filter in your application.

Minify your Assets

In order to use UglifyJs on your assets, you need to apply it to them. Since your assets are a part of the view layer, this work is done in your templates:

Listing 12-6

5. http://package.json.nodejitsu.com/

```
{% javascripts '@AcmeFooBundle/Resources/public/js/*' filter='uglifyjs2' %}
    <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 `uglifyjs2` filter to the asset tags above, you should now see minified JavaScripts coming over the wire much faster.

Disable Minification in Debug Mode

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

Listing 12-7
```
{% javascripts '@AcmeFooBundle/Resources/public/js/*' filter='?uglifyjs2' %}
    <script src="{{ asset_url }}"></script>
{% endjavascripts %}
```

To try this out, switch to your `prod` environment (`app.php`). But before you do, don't forget to *clear your cache* and *dump your assetic assets*.

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 `uglifyjs2` 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*.

Install, configure and use UglifyCss

The usage of UglifyCss works the same way as UglifyJs. First, make sure the node package is installed:

Listing 12-8
```
$ npm install -g uglifycss
```

Next, add the configuration for this filter:

Listing 12-9
```
# app/config/config.yml
assetic:
    filters:
        uglifycss:
            bin: /usr/local/bin/uglifycss
```

To use the filter for your css files, add the filter to the Assetic `stylesheets` helper:

Listing 12-10
```
{% stylesheets '@AcmeFooBundle/Resources/public/css/*' filter='uglifycss' %}
    <link rel="stylesheet" href="{{ asset_url }}" />
{% endstylesheets %}
```

Just like with the `uglifyjs2` filter, if you prefix the filter name with `?` (i.e. `?uglifycss`), the minification will only happen when you're not in debug mode.

Chapter 13

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.

The YUI Compressor is going through a *deprecation process*[2]. But don't worry! See *How to Minify CSS/JS Files (using UglifyJs and UglifyCss)* for an alternative.

Download the YUI Compressor JAR

The YUI Compressor is written in Java and distributed as a JAR. *Download the JAR*[3] 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 13-1
```yaml
# app/config/config.yml
assetic:
    # java: "/usr/bin/java"
    filters:
        yui_css:
            jar: "%kernel.root_dir%/Resources/java/yuicompressor.jar"
```

1. http://developer.yahoo.com/yui/compressor/
2. http://www.yuiblog.com/blog/2012/10/16/state-of-yui-compressor/
3. http://yuilibrary.com/projects/yuicompressor/

```
7           yui_js:
8               jar: "%kernel.root_dir%/Resources/java/yuicompressor.jar"
```

Windows users need to remember to update config to proper java location. In Windows7 x64 bit by default it's `C:\Program Files (x86)\Java\jre6\bin\java.exe`.

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.

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 13-2
```
1 {% javascripts '@AcmeFooBundle/Resources/public/js/*' filter='yui_js' %}
2     <script src="{{ asset_url }}"></script>
3 {% 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 13-3
```
1 {% stylesheets '@AcmeFooBundle/Resources/public/css/*' filter='yui_css' %}
2     <link rel="stylesheet" type="text/css" media="screen" href="{{ asset_url }}" />
3 {% 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 by 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 13-4
```
1 {% javascripts '@AcmeFooBundle/Resources/public/js/*' filter='?yui_js' %}
2     <script src="{{ asset_url }}"></script>
3 {% 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 14

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 14-1
```yaml
# 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 14-2
```twig
{% 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 14-3
```yaml
# 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 14-4
```yaml
# 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 14-5
```yaml
# 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 14-6
```
<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 14-7
```yaml
# app/config/config.yml
assetic:
    filters:
        jpegoptim:
            bin: path/to/jpegoptim
    twig:
```

```
7           functions:
8               jpegoptim: { output: images/*.jpg }
```

Chapter 15

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, node and node_modules. An example configuration might look like this:

Listing 15-1
```yaml
# app/config/config.yml
assetic:
    filters:
        coffee:
            bin: /usr/bin/coffee
            node: /usr/bin/node
            node_paths: [ /usr/lib/node_modules/ ]
```

Filter a Single File

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

Listing 15-2
```
{% 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 15-3
```
{% 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 15-4
```
# app/config/config.yml
assetic:
    filters:
        coffee:
            bin: /usr/bin/coffee
            node: /usr/bin/node
            node_paths: [ /usr/lib/node_modules/ ]
            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 15-5
```
{% 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 16

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 16-1

```php
// 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)
```

```
21          * @Assert\NotBlank
22          */
23         public $name;
24
25         /**
26          * @ORM\Column(type="string", length=255, nullable=true)
27          */
28         public $path;
29
30         public function getAbsolutePath()
31         {
32             return null === $this->path
33                 ? null
34                 : $this->getUploadRootDir().'/'.$this->path;
35         }
36
37         public function getWebPath()
38         {
39             return null === $this->path
40                 ? null
41                 : $this->getUploadDir().'/'.$this->path;
42         }
43
44         protected function getUploadRootDir()
45         {
46             // the absolute directory path where uploaded
47             // documents should be saved
48             return __DIR__.'/../../../../web/'.$this->getUploadDir();
49         }
50
51         protected function getUploadDir()
52         {
53             // get rid of the __DIR__ so it doesn't screw up
54             // when displaying uploaded doc/image in the view.
55             return 'uploads/documents';
56         }
57     }
```

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 16-2

```php
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 16-3

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

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

    /**
     * Sets file.
     *
     * @param UploadedFile $file
     */
    public function setFile(UploadedFile $file = null)
    {
        $this->file = $file;
    }

    /**
     * Get file.
     *
     * @return UploadedFile
     */
    public function getFile()
    {
        return $this->file;
    }
}
```

Listing 16-4

```yaml
# src/Acme/DemoBundle/Resources/config/validation.yml
Acme\DemoBundle\Entity\Document:
    properties:
        file:
            - File:
                maxSize: 6000000
```

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 16-5
```php
// ...
use Acme\DemoBundle\Entity\Document;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Template;
use Symfony\Component\HttpFoundation\Request;
// ...

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

    $form->handleRequest($request);

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

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

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

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

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 16-6
```php
if ($form->isValid()) {
    $em = $this->getDoctrine()->getManager();

    $document->upload();

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

    return $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 16-7
```php
public function upload()
{
```

1. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/File/UploadedFile.html

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

    // 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->getFile()->move(
        $this->getUploadRootDir(),
        $this->getFile()->getClientOriginalName()
    );

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

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

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 16-8
```
/**
 * @ORM\Entity
 * @ORM\HasLifecycleCallbacks
 */
class Document
{
}
```

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

Listing 16-9
```
use Symfony\Component\HttpFoundation\File\UploadedFile;

/**
 * @ORM\Entity
 * @ORM\HasLifecycleCallbacks
 */
class Document
{
    private $temp;

```

```
    /**
     * Sets file.
     *
     * @param UploadedFile $file
     */
    public function setFile(UploadedFile $file = null)
    {
        $this->file = $file;
        // check if we have an old image path
        if (isset($this->path)) {
            // store the old name to delete after the update
            $this->temp = $this->path;
            $this->path = null;
        } else {
            $this->path = 'initial';
        }
    }

    /**
     * @ORM\PrePersist()
     * @ORM\PreUpdate()
     */
    public function preUpload()
    {
        if (null !== $this->getFile()) {
            // do whatever you want to generate a unique name
            $filename = sha1(uniqid(mt_rand(), true));
            $this->path = $filename.'.'.$this->getFile()->guessExtension();
        }
    }

    /**
     * @ORM\PostPersist()
     * @ORM\PostUpdate()
     */
    public function upload()
    {
        if (null === $this->getFile()) {
            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
        $this->getFile()->move($this->getUploadRootDir(), $this->path);

        // check if we have an old image
        if (isset($this->temp)) {
            // delete the old image
            unlink($this->getUploadRootDir().'/'.$this->temp);
            // clear the temp image path
            $this->temp = null;
        }
        $this->file = null;
    }

    /**
     * @ORM\PostRemove()
     */
```

```
70      public function removeUpload()
71      {
72          if ($file = $this->getAbsolutePath()) {
73              unlink($file);
74          }
75      }
76  }
```

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 16-10

```
1  if ($form->isValid()) {
2      $em = $this->getDoctrine()->getManager();
3
4      $em->persist($document);
5      $em->flush();
6
7      return $this->redirect(...);
8  }
```

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 16-11

```
1  use Symfony\Component\HttpFoundation\File\UploadedFile;
2
3  /**
4   * @ORM\Entity
5   * @ORM\HasLifecycleCallbacks
6   */
7  class Document
8  {
9      private $temp;
10
11     /**
12      * Sets file.
13      *
```

```
     * @param UploadedFile $file
     */
    public function setFile(UploadedFile $file = null)
    {
        $this->file = $file;
        // check if we have an old image path
        if (is_file($this->getAbsolutePath())) {
            // store the old name to delete after the update
            $this->temp = $this->getAbsolutePath();
        } else {
            $this->path = 'initial';
        }
    }

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

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

        // check if we have an old image
        if (isset($this->temp)) {
            // delete the old image
            unlink($this->temp);
            // clear the temp image path
            $this->temp = null;
        }

        // 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->getFile()->move(
            $this->getUploadRootDir(),
            $this->id.'.'.$this->getFile()->guessExtension()
        );

        $this->setFile(null);
    }

    /**
     * @ORM\PreRemove()
     */
    public function storeFilenameForRemove()
    {
```

```
73          $this->temp = $this->getAbsolutePath();
74      }
75
76      /**
77       * @ORM\PostRemove()
78       */
79      public function removeUpload()
80      {
81          if (isset($this->temp)) {
82              unlink($this->temp);
83          }
84      }
85
86      public function getAbsolutePath()
87      {
88          return null === $this->path
89              ? null
90              : $this->getUploadRootDir().'/'.$this->id.'.'.$this->path;
91      }
92  }
```

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 17

How to use 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 18

How to Register 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.

The Doctrine website also explains all existing events that can be listened to.

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 18-1

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

services:
    my.listener:
        class: Acme\SearchBundle\EventListener\SearchIndexer
        tags:
```

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

```
            - { name: doctrine.event_listener, event: postPersist }
    my.listener2:
        class: Acme\SearchBundle\EventListener\SearchIndexer2
        tags:
            - { name: doctrine.event_listener, event: postPersist, connection: default }
    my.subscriber:
        class: Acme\SearchBundle\EventListener\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`. The class behind that service must have a `postPersist` method, which will be called when the event is dispatched:

Listing 18-2

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

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 entity's class type in your method (as shown above).

Creating the Subscriber Class

A doctrine event subscriber must implement the `Doctrine\Common\EventSubscriber` interface and have an event method for each event it subscribes to:

Listing 18-3

```php
// src/Acme/SearchBundle/EventListener/SearchIndexerSubscriber.php
namespace Acme\SearchBundle\EventListener;

use Doctrine\Common\EventSubscriber;
use Doctrine\ORM\Event\LifecycleEventArgs;
```

```php
// for doctrine 2.4: Doctrine\Common\Persistence\Event\LifecycleEventArgs;
use Acme\StoreBundle\Entity\Product;

class SearchIndexerSubscriber implements EventSubscriber
{
    public function getSubscribedEvents()
    {
        return array(
            'postPersist',
            'postUpdate',
        );
    }

    public function postUpdate(LifecycleEventArgs $args)
    {
        $this->index($args);
    }

    public function postPersist(LifecycleEventArgs $args)
    {
        $this->index($args);
    }

    public function index(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
        }
    }
}
```

Doctrine event subscribers can not return a flexible array of methods to call for the events like the *Symfony event subscriber* can. Doctrine event subscribers must return a simple array of the event names they subscribe to. Doctrine will then expect methods on the subscriber with the same name as each subscribed event, just as when using an event listener.

For a full reference, see chapter *The Event System*[2] in the Doctrine documentation.

2. http://docs.doctrine-project.org/projects/doctrine-orm/en/latest/reference/events.html

Chapter 19
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 19-1
```yaml
# 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 19-2

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

```php
class UserController extends Controller
{
    public function indexAction()
    {
        $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 19-3
```yaml
# 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 supported by DBAL by default) to a the `string` mapping type:

Listing 19-4
```yaml
# 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 20

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 20-1

```
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,
```

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

```
    PRIMARY KEY (`id`),
    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.yml` 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 it's assumed 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 table fields.

Listing 20-2
```
$ php app/console doctrine:mapping:import --force AcmeBlogBundle xml
```

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

It's also possible to generate the metadata files in YAML format by changing the last argument to `yml`.

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

Listing 20-3
```
<?xml version="1.0" encoding="utf-8"?>
<doctrine-mapping xmlns="http://doctrine-project.org/schemas/orm/doctrine-mapping"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://doctrine-project.org/schemas/orm/doctrine-mapping
http://doctrine-project.org/schemas/orm/doctrine-mapping.xsd">
  <entity name="Acme\BlogBundle\Entity\BlogPost" table="blog_post">
    <id name="id" type="bigint" column="id">
      <generator strategy="IDENTITY"/>
    </id>
    <field name="title" type="string" column="title" length="100" nullable="false"/>
    <field name="content" type="text" column="content" nullable="false"/>
    <field name="createdAt" type="datetime" column="created_at" nullable="false"/>
  </entity>
</doctrine-mapping>
```

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

Listing 20-4
```
$ php app/console doctrine:mapping:convert annotation ./src
$ php app/console doctrine:generate:entities AcmeBlogBundle
```

The first command generates entity classes with annotation mappings. But if you want to use yml or xml mapping instead of annotations, you should execute the second command only.

If you want to use annotations, you can safely delete the XML (or YAML) files after running these two commands.

For example, the newly created `BlogComment` entity class looks as follow:

Listing 20-5

```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 integer $id
     *
     * @ORM\Column(name="id", type="bigint")
     * @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.

If you want to have a `oneToMany` relationship, you will need to add it manually into the entity or to the generated `xml` or `yml` files. Add a section on the specific entities for `oneToMany` defining the `inversedBy` and the `mappedBy` pieces.

The generated entities are now ready to be used. Have fun!

Chapter 21

How to work with Multiple Entity Managers and Connections

You can use multiple Doctrine entity managers or connections 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 21-1

```yaml
doctrine:
    dbal:
        default_connection:   default
        connections:
            default:
                driver:   "%database_driver%"
                host:     "%database_host%"
                port:     "%database_port%"
                dbname:   "%database_name%"
                user:     "%database_user%"
                password: "%database_password%"
                charset:  UTF8
            customer:
                driver:   "%database_driver2%"
                host:     "%database_host2%"
                port:     "%database_port2%"
                dbname:   "%database_name2%"
                user:     "%database_user2%"
                password: "%database_password2%"
                charset:  UTF8
```

```
21
22      orm:
23          default_entity_manager:   default
24          entity_managers:
25              default:
26                  connection:       default
27                  mappings:
28                      AcmeDemoBundle: ~
29                      AcmeStoreBundle: ~
30              customer:
31                  connection:       customer
32                  mappings:
33                      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`. You've also defined two connections, one for each entity manager.

When working with multiple connections and entity managers, you should be explicit about which configuration you want. If you *do* omit the name of the connection or entity manager, the default (i.e. `default`) is used.

When working with multiple connections to create your databases:

Listing 21-2
```
1  # Play only with "default" connection
2  $ php app/console doctrine:database:create
3
4  # Play only with "customer" connection
5  $ php app/console doctrine:database:create --connection=customer
```

When working with multiple entity managers to update your schema:

Listing 21-3
```
1  # Play only with "default" mappings
2  $ php app/console doctrine:schema:update --force
3
4  # Play only with "customer" mappings
5  $ php app/console doctrine:schema:update --force --em=customer
```

If you *do* omit the entity manager's name when asking for it, the default entity manager (i.e. `default`) is returned:

Listing 21-4
```
1  class UserController extends Controller
2  {
3      public function indexAction()
4      {
5          // both return the "default" em
6          $em = $this->get('doctrine')->getManager();
7          $em = $this->get('doctrine')->getManager('default');
8
9          $customerEm =  $this->get('doctrine')->getManager('customer');
10     }
11 }
```

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.

The same applies to repository call:

Listing 21-5

```php
class UserController extends Controller
{
    public function indexAction()
    {
        // Retrieves a repository managed by the "default" em
        $products = $this->get('doctrine')
            ->getRepository('AcmeStoreBundle:Product')
            ->findAll()
        ;

        // Explicit way to deal with the "default" em
        $products = $this->get('doctrine')
            ->getRepository('AcmeStoreBundle:Product', 'default')
            ->findAll()
        ;

        // Retrieves a repository managed by the "customer" em
        $customers = $this->get('doctrine')
            ->getRepository('AcmeCustomerBundle:Customer', 'customer')
            ->findAll()
        ;
    }
}
```

Chapter 22
How to Register 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 22-1
```yaml
# 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/latest/cookbook/dql-user-defined-functions.html

Chapter 23

How to Define Relationships with Abstract Classes and Interfaces

One of the goals of bundles is to create discreet bundles of functionality that do not have many (if any) dependencies, allowing you to use that functionality in other applications without including unnecessary items.

Doctrine 2.2 includes a new utility called the `ResolveTargetEntityListener`, that functions by intercepting certain calls inside Doctrine and rewriting `targetEntity` parameters in your metadata mapping at runtime. It means that in your bundle you are able to use an interface or abstract class in your mappings and expect correct mapping to a concrete entity at runtime.

This functionality allows you to define relationships between different entities without making them hard dependencies.

Background

Suppose you have an *InvoiceBundle* which provides invoicing functionality and a *CustomerBundle* that contains customer management tools. You want to keep these separated, because they can be used in other systems without each other, but for your application you want to use them together.

In this case, you have an `Invoice` entity with a relationship to a non-existent object, an `InvoiceSubjectInterface`. The goal is to get the `ResolveTargetEntityListener` to replace any mention of the interface with a real object that implements that interface.

Set up

Let's use the following basic entities (which are incomplete for brevity) to explain how to set up and use the RTEL.

A Customer entity:

Listing 23-1

```php
// src/Acme/AppBundle/Entity/Customer.php

namespace Acme\AppBundle\Entity;

use Doctrine\ORM\Mapping as ORM;
use Acme\CustomerBundle\Entity\Customer as BaseCustomer;
use Acme\InvoiceBundle\Model\InvoiceSubjectInterface;

/**
 * @ORM\Entity
 * @ORM\Table(name="customer")
 */
class Customer extends BaseCustomer implements InvoiceSubjectInterface
{
    // In our example, any methods defined in the InvoiceSubjectInterface
    // are already implemented in the BaseCustomer
}
```

An Invoice entity:

Listing 23-2

```php
// src/Acme/InvoiceBundle/Entity/Invoice.php

namespace Acme\InvoiceBundle\Entity;

use Doctrine\ORM\Mapping AS ORM;
use Acme\InvoiceBundle\Model\InvoiceSubjectInterface;

/**
 * Represents an Invoice.
 *
 * @ORM\Entity
 * @ORM\Table(name="invoice")
 */
class Invoice
{
    /**
     * @ORM\ManyToOne(targetEntity="Acme\InvoiceBundle\Model\InvoiceSubjectInterface")
     * @var InvoiceSubjectInterface
     */
    protected $subject;
}
```

An InvoiceSubjectInterface:

Listing 23-3

```php
// src/Acme/InvoiceBundle/Model/InvoiceSubjectInterface.php

namespace Acme\InvoiceBundle\Model;

/**
 * An interface that the invoice Subject object should implement.
 * In most circumstances, only a single object should implement
 * this interface as the ResolveTargetEntityListener can only
 * change the target to a single object.
 */
interface InvoiceSubjectInterface
{
    // List any additional methods that your InvoiceBundle
```

```
    // will need to access on the subject so that you can
    // be sure that you have access to those methods.

    /**
     * @return string
     */
    public function getName();
}
```

Next, you need to configure the listener, which tells the DoctrineBundle about the replacement:

Listing 23-4

```
# app/config/config.yml
doctrine:
    # ....
    orm:
        # ....
        resolve_target_entities:
            Acme\InvoiceBundle\Model\InvoiceSubjectInterface: Acme\AppBundle\Entity\Customer
```

Final Thoughts

With the `ResolveTargetEntityListener`, you are able to decouple your bundles, keeping them usable by themselves, but still being able to define relationships between different objects. By using this method, your bundles will end up being easier to maintain independently.

Chapter 24

How to provide model classes for several Doctrine implementations

When building a bundle that could be used not only with Doctrine ORM but also the CouchDB ODM, MongoDB ODM or PHPCR ODM, you should still only write one model class. The Doctrine bundles provide a compiler pass to register the mappings for your model classes.

For non-reusable bundles, the easiest option is to put your model classes in the default locations: `Entity` for the Doctrine ORM or `Document` for one of the ODMs. For reusable bundles, rather than duplicate model classes just to get the auto mapping, use the compiler pass.

New in version 2.3: The base mapping compiler pass was added in Symfony 2.3. The Doctrine bundles support it from DoctrineBundle >= 1.2.1, MongoDBBundle >= 3.0.0, PHPCRBundle >= 1.0.0-alpha2 and the (unversioned) CouchDBBundle supports the compiler pass since the *CouchDB Mapping Compiler Pass pull request*[1] was merged.

If you want your bundle to support older versions of Symfony and Doctrine, you can provide a copy of the compiler pass in your bundle. See for example the *FOSUserBundle mapping configuration*[2] `addRegisterMappingsPass`.

In your bundle class, write the following code to register the compiler pass. This one is written for the FOSUserBundle, so parts of it will need to be adapted for your case:

Listing 24-1

```
use Doctrine\Bundle\DoctrineBundle\DependencyInjection\Compiler\DoctrineOrmMappingsPass;
use Doctrine\Bundle\MongoDBBundle\DependencyInjection\Compiler\DoctrineMongoDBMappingsPass;
use Doctrine\Bundle\CouchDBBundle\DependencyInjection\Compiler\DoctrineCouchDBMappingsPass;
use Doctrine\Bundle\PHPCRBundle\DependencyInjection\Compiler\DoctrinePhpcrMappingsPass;

class FOSUserBundle extends Bundle
{
```

1. https://github.com/doctrine/DoctrineCouchDBBundle/pull/27
2. https://github.com/FriendsOfSymfony/FOSUserBundle/blob/master/FOSUserBundle.php

```
    public function build(ContainerBuilder $container)
    {
        parent::build($container);
        // ...

        $modelDir = realpath(__DIR__.'/Resources/config/doctrine/model');
        $mappings = array(
            $modelDir => 'FOS\UserBundle\Model',
        );

        $ormCompilerClass =
'Doctrine\Bundle\DoctrineBundle\DependencyInjection\Compiler\DoctrineOrmMappingsPass';
        if (class_exists($ormCompilerClass)) {
            $container->addCompilerPass(
                DoctrineOrmMappingsPass::createXmlMappingDriver(
                    $mappings,
                    array('fos_user.model_manager_name'),
                    'fos_user.backend_type_orm'
            ));
        }

        $mongoCompilerClass =
'Doctrine\Bundle\MongoDBBundle\DependencyInjection\Compiler\DoctrineMongoDBMappingsPass';
        if (class_exists($mongoCompilerClass)) {
            $container->addCompilerPass(
                DoctrineMongoDBMappingsPass::createXmlMappingDriver(
                    $mappings,
                    array('fos_user.model_manager_name'),
                    'fos_user.backend_type_mongodb'
            ));
        }

        $couchCompilerClass =
'Doctrine\Bundle\CouchDBBundle\DependencyInjection\Compiler\DoctrineCouchDBMappingsPass';
        if (class_exists($couchCompilerClass)) {
            $container->addCompilerPass(
                DoctrineCouchDBMappingsPass::createXmlMappingDriver(
                    $mappings,
                    array('fos_user.model_manager_name'),
                    'fos_user.backend_type_couchdb'
            ));
        }

        $phpcrCompilerClass =
'Doctrine\Bundle\PHPCRBundle\DependencyInjection\Compiler\DoctrinePhpcrMappingsPass';
        if (class_exists($phpcrCompilerClass)) {
            $container->addCompilerPass(
                DoctrinePhpcrMappingsPass::createXmlMappingDriver(
                    $mappings,
                    array('fos_user.model_manager_name'),
                    'fos_user.backend_type_phpcr'
            ));
        }
    }
}
```

Note the *class_exists*[3] check. This is crucial, as you do not want your bundle to have a hard dependency on all Doctrine bundles but let the user decide which to use.

The compiler pass provides factory methods for all drivers provided by Doctrine: Annotations, XML, Yaml, PHP and StaticPHP. The arguments are:

- a map/hash of absolute directory path to namespace;
- an array of container parameters that your bundle uses to specify the name of the Doctrine manager that it is using. In the above example, the FOSUserBundle stores the manager name that's being used under the `fos_user.model_manager_name` parameter. The compiler pass will append the parameter Doctrine is using to specify the name of the default manager. The first parameter found is used and the mappings are registered with that manager;
- an optional container parameter name that will be used by the compiler pass to determine if this Doctrine type is used at all (this is relevant if your user has more than one type of Doctrine bundle installed, but your bundle is only used with one type of Doctrine.

The factory method is using the `SymfonyFileLocator` of Doctrine, meaning it will only see XML and YML mapping files if they do not contain the full namespace as the filename. This is by design: the `SymfonyFileLocator` simplifies things by assuming the files are just the "short" version of the class as their filename (e.g. `BlogPost.orm.xml`)

If you also need to map a base class, you can register a compiler pass with the `DefaultFileLocator` like this. This code is simply taken from the `DoctrineOrmMappingsPass` and adapted to use the `DefaultFileLocator` instead of the `SymfonyFileLocator`:

Listing 24-2
```php
private function buildMappingCompilerPass()
{
    $arguments = array(array(realpath(__DIR__ . '/Resources/config/doctrine-base')),
'.orm.xml');
    $locator = new
Definition('Doctrine\Common\Persistence\Mapping\Driver\DefaultFileLocator',
$arguments);
    $driver = new Definition('Doctrine\ORM\Mapping\Driver\XmlDriver',
array($locator));

    return new DoctrineOrmMappingsPass(
        $driver,
        array('Full\Namespace'),
        array('your_bundle.manager_name'),
        'your_bundle.orm_enabled'
    );
}
```

Now place your mapping file into `/Resources/config/doctrine-base` with the fully qualified class name, separated by `.` instead of \, for example `Other.Namespace.Model.Name.orm.xml`. You may not mix the two as otherwise the SymfonyFileLocator will get confused.

Adjust accordingly for the other Doctrine implementations.

3. http://php.net/manual/en/function.class-exists.php

Chapter 25

How to implement a simple Registration Form

Some forms have extra fields whose values don't need to be stored in the database. For example, you may want to create a registration form with some extra fields (like a "terms accepted" checkbox field) and embed the form that actually stores the account information.

The simple User model

You have a simple `User` entity mapped to the database:

Listing 25-1

```php
// src/Acme/AccountBundle/Entity/User.php
namespace Acme\AccountBundle\Entity;

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

/**
 * @ORM\Entity
 * @UniqueEntity(fields="email", message="Email already taken")
 */
class User
{
    /**
     * @ORM\Id
     * @ORM\Column(type="integer")
     * @ORM\GeneratedValue(strategy="AUTO")
     */
    protected $id;

    /**
     * @ORM\Column(type="string", length=255)
     * @Assert\NotBlank()
     * @Assert\Email()
     */
    protected $email;
```

```
27
28      /**
29       * @ORM\Column(type="string", length=255)
30       * @Assert\NotBlank()
31       * @Assert\Length(max = 4096)
32       */
33      protected $plainPassword;
34
35      public function getId()
36      {
37          return $this->id;
38      }
39
40      public function getEmail()
41      {
42          return $this->email;
43      }
44
45      public function setEmail($email)
46      {
47          $this->email = $email;
48      }
49
50      public function getPlainPassword()
51      {
52          return $this->plainPassword;
53      }
54
55      public function setPlainPassword($password)
56      {
57          $this->plainPassword = $password;
58      }
59  }
```

This `User` entity contains three fields and two of them (`email` and `plainPassword`) should display on the form. The email property must be unique in the database, this is enforced by adding this validation at the top of the class.

If you want to integrate this User within the security system, you need to implement the *UserInterface* of the security component.

Why the 4096 Password Limit?

Notice that the `plainPassword` has a max length of `4096` characters. For security purposes (*CVE-2013-5750*[1]), Symfony limits the plain password length to 4096 characters when encoding it. Adding this constraint makes sure that your form will give a validation error if anyone tries a super-long password.

You'll need to add this constraint anywhere in your application where your user submits a plaintext password (e.g. change password form). The only place where you don't need to worry about this is your login form, since Symfony's Security component handles this for you.

1. `http://symfony.com/blog/cve-2013-5750-security-issue-in-fosuserbundle-login-form`

Create a Form for the Model

Next, create the form for the `User` model:

Listing 25-2
```php
// src/Acme/AccountBundle/Form/Type/UserType.php
namespace Acme\AccountBundle\Form\Type;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolverInterface;

class UserType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder->add('email', 'email');
        $builder->add('plainPassword', 'repeated', array(
           'first_name'  => 'password',
           'second_name' => 'confirm',
           'type'        => 'password',
        ));
    }

    public function setDefaultOptions(OptionsResolverInterface $resolver)
    {
        $resolver->setDefaults(array(
            'data_class' => 'Acme\AccountBundle\Entity\User'
        ));
    }

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

There are just two fields: `email` and `plainPassword` (repeated to confirm the entered password). The `data_class` option tells the form the name of data class (i.e. your `User` entity).

To explore more things about the form component, read *Forms*.

Embedding the User form into a Registration Form

The form that you'll use for the registration page is not the same as the form used to simply modify the `User` (i.e. `UserType`). The registration form will contain further fields like "accept the terms", whose value won't be stored in the database.

Start by creating a simple class which represents the "registration":

Listing 25-3
```php
// src/Acme/AccountBundle/Form/Model/Registration.php
namespace Acme\AccountBundle\Form\Model;

use Symfony\Component\Validator\Constraints as Assert;
```

```php

use Acme\AccountBundle\Entity\User;

class Registration
{
    /**
     * @Assert\Type(type="Acme\AccountBundle\Entity\User")
     * @Assert\Valid()
     */
    protected $user;

    /**
     * @Assert\NotBlank()
     * @Assert\True()
     */
    protected $termsAccepted;

    public function setUser(User $user)
    {
        $this->user = $user;
    }

    public function getUser()
    {
        return $this->user;
    }

    public function getTermsAccepted()
    {
        return $this->termsAccepted;
    }

    public function setTermsAccepted($termsAccepted)
    {
        $this->termsAccepted = (Boolean) $termsAccepted;
    }
}
```

Next, create the form for this `Registration` model:

Listing 25-4

```php
// src/Acme/AccountBundle/Form/Type/RegistrationType.php
namespace Acme\AccountBundle\Form\Type;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;

class RegistrationType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder->add('user', new UserType());
        $builder->add(
            'terms',
            'checkbox',
            array('property_path' => 'termsAccepted')
        );
    }

```

```
19      public function getName()
20      {
21          return 'registration';
22      }
23  }
```

You don't need to use special method for embedding the `UserType` form. A form is a field, too - so you can add this like any other field, with the expectation that the `Registration.user` property will hold an instance of the `User` class.

Handling the Form Submission

Next, you need a controller to handle the form. Start by creating a simple controller for displaying the registration form:

Listing 25-5
```
1  // src/Acme/AccountBundle/Controller/AccountController.php
2  namespace Acme\AccountBundle\Controller;
3
4  use Symfony\Bundle\FrameworkBundle\Controller\Controller;
5  use Symfony\Component\HttpFoundation\Response;
6
7  use Acme\AccountBundle\Form\Type\RegistrationType;
8  use Acme\AccountBundle\Form\Model\Registration;
9
10 class AccountController extends Controller
11 {
12     public function registerAction()
13     {
14         $registration = new Registration();
15         $form = $this->createForm(new RegistrationType(), $registration, array(
16             'action' => $this->generateUrl('account_create'),
17         ));
18
19         return $this->render(
20             'AcmeAccountBundle:Account:register.html.twig',
21             array('form' => $form->createView())
22         );
23     }
24 }
```

and its template:

Listing 25-6
```
1  {# src/Acme/AccountBundle/Resources/views/Account/register.html.twig #}
2  {{ form(form) }}
```

Next, create the controller which handles the form submission. This performs the validation and saves the data into the database:

Listing 25-7
```
1  public function createAction(Request $request)
2  {
3      $em = $this->getDoctrine()->getEntityManager();
4
5      $form = $this->createForm(new RegistrationType(), new Registration());
6
```

```
    $form->handleRequest($request);

    if ($form->isValid()) {
        $registration = $form->getData();

        $em->persist($registration->getUser());
        $em->flush();

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

    return $this->render(
        'AcmeAccountBundle:Account:register.html.twig',
        array('form' => $form->createView())
    );
}
```

Add New Routes

Next, update your routes. If you're placing your routes inside your bundle (as shown here), don't forget to make sure that the routing file is being *imported*.

Listing 25-8

```
# src/Acme/AccountBundle/Resources/config/routing.yml
account_register:
    pattern:  /register
    defaults: { _controller: AcmeAccountBundle:Account:register }

account_create:
    pattern:  /register/create
    defaults: { _controller: AcmeAccountBundle:Account:create }
```

Update your Database Schema

Of course, since you've added a `User` entity during this tutorial, make sure that your database schema has been updated properly:

```
$ php app/console doctrine:schema:update --force
```

That's it! Your form now validates, and allows you to save the `User` object to the database. The extra `terms` checkbox on the `Registration` model class is used during validation, but not actually used afterwards when saving the User to the database.

Chapter 26

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:

Listing 26-1
```
{{ form_row(form.age) }}
```

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

Listing 26-2
```
<div>
    {{ form_label(form.age) }}
    {{ form_errors(form.age) }}
    {{ form_widget(form.age) }}
</div>
```

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:

Listing 26-3
```
<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>
```

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

Listing 26-4

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

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 each 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 an `integer` type field is rendered, an `input number` field is generated

Listing 26-5
```twig
{{ form_widget(form.age) }}
```

renders:

Listing 26-6
```twig
<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 the `FrameworkBundle/Resources/views/Form` folder.

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

Listing 26-7
```twig
{# form_div_layout.html.twig #}
{% block integer_widget %}
    {% set type = type|default('number') %}
    {{ block('form_widget_simple') }}
{% endblock integer_widget %}
```

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

Listing 26-8
```twig
{# form_div_layout.html.twig #}
{% block form_widget_simple %}
    {% set type = type|default('text') %}
    <input type="{{ type }}" {{ block('widget_attributes') }} {% if value is not empty
%}value="{{ value }}" {% endif %}/>
{% endblock form_widget_simple %}
```

1. https://github.com/symfony/symfony/blob/2.3/src/Symfony/Bridge/Twig/Resources/views/Form/form_div_layout.html.twig
2. https://github.com/symfony/symfony/blob/2.3/src/Symfony/Bridge/Twig/Resources/views/Form/form_div_layout.html.twig

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 fragments are individual template files in this folder.

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 `form_errors` fragment. This takes advantage of field type inheritance. Specifically, since the `text` type extends from the `form` 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. `form_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 26-9
```
1 {% extends '::base.html.twig' %}
2
```

```twig
{% form_theme form _self %}

{% block integer_widget %}
    <div class="integer_widget">
        {% set type = type|default('number') %}
        {{ block('form_widget_simple') }}
    </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 26-10
```twig
{# src/Acme/DemoBundle/Resources/views/Form/fields.html.twig #}
{% block integer_widget %}
    <div class="integer_widget">
        {% set type = type|default('number') %}
        {{ block('form_widget_simple') }}
    </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 26-11
```twig
{% 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.

Child Forms

You can also apply a form theme to a specific child of your form:

Listing 26-12
```twig
{% form_theme form.child 'AcmeDemoBundle:Form:fields.html.twig' %}
```

This is useful when you want to have a custom theme for a nested form that's different than the one of your main form. Just specify both your themes:

Listing 26-13
```
{% form_theme form 'AcmeDemoBundle:Form:fields.html.twig' %}

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

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 26-14
```
<!-- src/Acme/DemoBundle/Resources/views/Form/integer_widget.html.php -->
<div class="integer_widget">
    <?php echo $view['form']->block($form, 'form_widget_simple', 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 26-15
```
<?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.

If you want to apply a theme to a specific child form, pass it to the `setTheme` method:

Listing 26-16
```
<?php $view['form']->setTheme($form['child'], 'AcmeDemoBundle:Form/Child'); ?>
```

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 then 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 26-17
```
{% use 'form_div_layout.html.twig' with integer_widget as base_integer_widget %}
```

3. https://github.com/symfony/symfony/blob/2.3/src/Symfony/Bridge/Twig/Resources/views/Form/form_div_layout.html.twig

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 26-18
```
{% 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 26-19
```
{# 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 %}
```

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 26-20
```
# 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 26-21

4. https://github.com/symfony/symfony/blob/2.3/src/Symfony/Bridge/Twig/Resources/views/Form/form_div_layout.html.twig

```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 26-22
```twig
{% 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 26-23
```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 26-24
```yaml
# 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 26-25
```php
<?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 26-26

```twig
{% form_theme form _self %}

{% block _product_name_widget %}
    <div class="text_widget">
        {{ block('form_widget_simple') }}
    </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 26-27
```twig
{# _product_name_row.html.twig #}
{% 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 26-28
```twig
{{ form_errors(form.age) }}
```

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

Listing 26-29
```
<ul>
    <li>This field is required</li>
</ul>
```

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

Listing 26-30
```
{# form_errors.html.twig #}
{% block form_errors %}
    {% spaceless %}
        {% if errors|length > 0 %}
        <ul>
            {% for error in errors %}
                <li>{{ error.message }}</li>
            {% endfor %}
        </ul>
        {% endif %}
    {% endspaceless %}
{% endblock form_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 26-31
```
{{ 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 `form_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 `form_row` fragment. For example, suppose you want to add a class to the `div` element around each row:

Listing 26-32
```
{# form_row.html.twig #}
{% block form_row %}
    <div class="form_row">
        {{ form_label(form) }}
        {{ form_errors(form) }}
        {{ form_widget(form) }}
    </div>
{% endblock form_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 `form_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 26-33
```twig
{% use 'form_div_layout.html.twig' with form_label as base_form_label %}

{% block form_label %}
    {{ block('base_form_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 26-34
```twig
{% extends 'form_div_layout.html.twig' %}

{% block form_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 26-35
```php
<!-- form_label.html.php -->

<!-- original content -->
<?php if ($required) { $label_attr['class'] = trim((isset($label_attr['class']) ?
$label_attr['class'] : '').' required'); } ?>
<?php if (!$compound) { $label_attr['for'] = $id; } ?>
<?php if (!$label) { $label = $view['form']->humanize($name); } ?>
<label <?php foreach ($label_attr as $k => $v) { printf('%s="%s" ', $view->escape($k),
$view->escape($v)); } ?>><?php echo $view->escape($view['translator']->trans($label,
array(), $translation_domain)) ?></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 26-36
```
{% use 'form_div_layout.html.twig' with form_widget_simple as base_form_widget_simple %}

{% block form_widget_simple %}
    {{ block('base_form_widget_simple') }}

    {% 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 26-37
```
{% extends 'form_div_layout.html.twig' %}

{% block form_widget_simple %}
    {{ 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 26-38
```
<!-- form_widget_simple.html.php -->

<!-- Original content -->
<input
    type="<?php echo isset($type) ? $view->escape($type) : 'text' ?>"
    <?php if (!empty($value)): ?>value="<?php echo $view->escape($value) ?>"<?php endif ?>
    <?php echo $view['form']->block($form, 'widget_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 26-39
```
{{ form_widget(form.title, {'help': 'foobar'}) }}
```

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

Using Form Variables

Most of the functions available for rendering different parts of a form (e.g. the form widget, form label, form errors, etc) also allow you to make certain customizations directly. Look at the following example:

Listing 26-40
```
{# render a widget, but add a "foo" class to it #}
{{ form_widget(form.name, { 'attr': {'class': 'foo'} }) }}
```

The array passed as the second argument contains form "variables". For more details about this concept in Twig, see *More about Form Variables*.

Chapter 27

How to use 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 might want to add a textbox instead, where the user can simply enter the issue number.

You could try to do this in your controller, but it's not the best solution. It would be better if this issue were automatically converted to an Issue object. This is where Data Transformers come into play.

Creating the Transformer

First, create an *IssueToNumberTransformer* class - this class will be responsible for converting to and from the issue number and the Issue object:

Listing 27-1

```php
// 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;
    }
}
```

If you want a new issue to be created when an unknown number is entered, you can instantiate it rather than throwing the `TransformationFailedException`.

Using the Transformer

Now that you have the transformer built, you just need to add it to your issue field in some form.

> You can also use transformers without creating a new custom form type by calling `addModelTransformer` (or `addViewTransformer` - see Model and View Transformers) on any field builder:

Listing 27-2

```php
use Symfony\Component\Form\FormBuilderInterface;
use Acme\TaskBundle\Form\DataTransformer\IssueToNumberTransformer;

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

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

        // add a normal text field, but add your transformer to it
        $builder->add(
            $builder->create('issue', 'text')
                ->addModelTransformer($transformer)
        );
    }

    public function setDefaultOptions(OptionsResolverInterface $resolver)
    {
        $resolver->setDefaults(array(
            'data_class' => 'Acme\TaskBundle\Entity\Task',
        ));

        $resolver->setRequired(array(
            'em',
        ));

        $resolver->setAllowedTypes(array(
            'em' => 'Doctrine\Common\Persistence\ObjectManager',
        ));

        // ...
    }

    // ...
}
```

This example requires that you pass in the entity manager as an option when creating your form. Later, you'll learn how you could create a custom `issue` field type to avoid needing to do this in your controller:

Listing 27-3

```php
$taskForm = $this->createForm(new TaskType(), $task, array(
    'em' => $this->getDoctrine()->getManager(),
));
```

Cool, you're done! Your user will be able to enter an issue number into the text field and it will be transformed back into an Issue object. This means that, after a successful submission, the Form framework will pass a real Issue object to `Task::setIssue()` instead of the issue number.

If the issue isn't found, a form error will be created for that field and its error message can be controlled with the `invalid_message` field option.

Notice that adding a transformer requires using a slightly more complicated syntax when adding the field. The following is **wrong**, as the transformer would be applied to the entire form, instead of just this field:

Listing 27-4
```php
// THIS IS WRONG - TRANSFORMER WILL BE APPLIED TO THE ENTIRE FORM
// see above example for correct code
$builder->add('issue', 'text')
    ->addModelTransformer($transformer);
```

Model and View Transformers

In the above example, the transformer was used as a "model" transformer. In fact, there are two different types of transformers and three different types of underlying data.

../../_images/DataTransformersTypes.png

In any form, the 3 different types of data are:

1) **Model data** - This is the data in the format used in your application (e.g. an `Issue` object). If you call `Form::getData` or `Form::setData`, you're dealing with the "model" data.

2) **Norm Data** - This is a normalized version of your data, and is commonly the same as your "model" data (though not in our example). It's not commonly used directly.

3) **View Data** - This is the format that's used to fill in the form fields themselves. It's also the format in which the user will submit the data. When you call `Form::submit($data)`, the `$data` is in the "view" data format.

The 2 different types of transformers help convert to and from each of these types of data:
Model transformers:

- `transform`: "model data" => "norm data"
- `reverseTransform`: "norm data" => "model data"

View transformers:

- `transform`: "norm data" => "view data"
- `reverseTransform`: "view data" => "norm data"

Which transformer you need depends on your situation.

To use the view transformer, call `addViewTransformer`.

So why use the model transformer?

In this example, the field is a `text` field, and a text field is always expected to be a simple, scalar format in the "norm" and "view" formats. For this reason, the most appropriate transformer was the "model" transformer (which converts to/from the *norm* format - string issue number - to the *model* format - Issue object).

The difference between the transformers is subtle and you should always think about what the "norm" data for a field should really be. For example, the "norm" data for a `text` field is a string, but is a `DateTime` object for a `date` field.

Using Transformers in a custom field type

In the above example, you applied the transformer to a normal `text` field. This was easy, but has two downsides:

1) You need to always remember to apply the transformer whenever you're adding a field for issue numbers

2) You need to worry about passing in the `em` option whenever you're creating a form that uses the transformer.

Because of these, you may choose to *create a custom field type*. First, create the custom field type class:

Listing 27-5

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

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Acme\TaskBundle\Form\DataTransformer\IssueToNumberTransformer;
use Doctrine\Common\Persistence\ObjectManager;
use Symfony\Component\OptionsResolver\OptionsResolverInterface;

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

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

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

    public function setDefaultOptions(OptionsResolverInterface $resolver)
    {
        $resolver->setDefaults(array(
            'invalid_message' => 'The selected issue does not exist',
        ));
    }

    public function getParent()
    {
        return 'text';
    }

```

```
43      public function getName()
44      {
45          return 'issue_selector';
46      }
47  }
```

Next, register your type as a service and tag it with `form.type` so that it's recognized as a custom field type:

Listing 27-6
```
1  services:
2      acme_demo.type.issue_selector:
3          class: Acme\TaskBundle\Form\Type\IssueSelectorType
4          arguments: ["@doctrine.orm.entity_manager"]
5          tags:
6              - { name: form.type, alias: issue_selector }
```

Now, whenever you need to use your special `issue_selector` field type, it's quite easy:

Listing 27-7
```
1  // src/Acme/TaskBundle/Form/Type/TaskType.php
2  namespace Acme\TaskBundle\Form\Type;
3
4  use Symfony\Component\Form\AbstractType;
5  use Symfony\Component\Form\FormBuilderInterface;
6
7  class TaskType extends AbstractType
8  {
9      public function buildForm(FormBuilderInterface $builder, array $options)
10     {
11         $builder
12             ->add('task')
13             ->add('dueDate', null, array('widget' => 'single_text'))
14             ->add('issue', 'issue_selector');
15     }
16
17     public function getName()
18     {
19         return 'task';
20     }
21 }
```

Chapter 28

How to Dynamically Modify Forms Using Form Events

Often times, a form can't be created statically. In this entry, you'll learn how to customize your form based on three common use-cases:

1. *Customizing your Form based on the underlying Data*

Example: you have a "Product" form and need to modify/add/remove a field based on the data on the underlying Product being edited.

2. *How to Dynamically Generate Forms based on user Data*

Example: you create a "Friend Message" form and need to build a drop-down that contains only users that are friends with the *current* authenticated user.

3. *Dynamic generation for submitted Forms*

Example: on a registration form, you have a "country" field and a "state" field which should populate dynamically based on the value in the "country" field.

Customizing your Form based on the underlying Data

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

Listing 28-1

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

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolverInterface;

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

```
        $builder->add('price');
    }

    public function setDefaultOptions(OptionsResolverInterface $resolver)
    {
        $resolver->setDefaults(array(
            'data_class' => 'Acme\DemoBundle\Entity\Product'
        ));
    }

    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.

Assume for a moment that this form utilizes an imaginary "Product" class that has only two properties ("name" and "price"). The form generated from this class will look the exact same regardless if 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 your ProductType form class, let's delegate the responsibility of creating that particular field to an Event Subscriber:

Listing 28-2

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

// ...
use Acme\DemoBundle\Form\EventListener\AddNameFieldSubscriber;

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

        $builder->addEventSubscriber(new AddNameFieldSubscriber());
    }

    // ...
}
```

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:

New in version 2.2: The ability to pass a string into *`FormInterface::add`*[1] was added in Symfony 2.2.

Listing 28-3

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

use Symfony\Component\Form\FormEvent;
use Symfony\Component\Form\FormEvents;
use Symfony\Component\EventDispatcher\EventSubscriberInterface;

class AddNameFieldSubscriber implements EventSubscriberInterface
{
    public static function getSubscribedEvents()
    {
        // Tells the dispatcher that you 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(FormEvent $event)
    {
        $data = $event->getData();
        $form = $event->getForm();

        // check if the product object is "new"
        // If you didn't pass any data to the form, the data is "null".
        // This should be considered a new "Product"
        if (!$data || !$data->getId()) {
            $form->add('name', 'text');
        }
    }
}
```

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

You can view the full list of form events via the *`FormEvents`*[3] class.

1. http://api.symfony.com/2.3/Symfony/Component/Form/FormInterface.html#add()
2. http://api.symfony.com/2.3/Symfony/Component/Form/FormEvents.html
3. http://api.symfony.com/2.3/Symfony/Component/Form/FormEvents.html

How to Dynamically Generate Forms based on user Data

Sometimes you want a form to be generated dynamically based not only on data from the form but also on something else - like some data from the current user. Suppose you have a social website where a user can only message people who are his friends on the website. In this case, a "choice list" of whom to message should only contain users that are the current user's friends.

Creating the Form Type

Using an event listener, your form might look like this:

Listing 28-4
```php
// src/Acme/DemoBundle/Form/Type/FriendMessageFormType.php
namespace Acme\DemoBundle\Form\Type;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\Form\FormEvents;
use Symfony\Component\Form\FormEvent;
use Symfony\Component\Security\Core\SecurityContext;
use Symfony\Component\OptionsResolver\OptionsResolverInterface;

class FriendMessageFormType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('subject', 'text')
            ->add('body', 'textarea')
        ;
        $builder->addEventListener(FormEvents::PRE_SET_DATA, function(FormEvent $event){
            // ... add a choice list of friends of the current application user
        });
    }

    public function getName()
    {
        return 'acme_friend_message';
    }

    public function setDefaultOptions(OptionsResolverInterface $resolver)
    {
    }
}
```

The problem is now to get the current user and create a choice field that contains only this user's friends. Luckily it is pretty easy to inject a service inside of the form. This can be done in the constructor:

Listing 28-5
```php
private $securityContext;

public function __construct(SecurityContext $securityContext)
{
    $this->securityContext = $securityContext;
}
```

You might wonder, now that you have access to the User (through the security context), why not just use it directly in `buildForm` and omit the event listener? This is because doing so in the `buildForm` method would result in the whole form type being modified and not just this one form instance. This may not usually be a problem, but technically a single form type could be used on a single request to create many forms or fields.

Customizing the Form Type

Now that you have all the basics in place you an take advantage of the `securityContext` and fill in the listener logic:

Listing 28-6

```
// src/Acme/DemoBundle/FormType/FriendMessageFormType.php

use Symfony\Component\Security\Core\SecurityContext;
use Doctrine\ORM\EntityRepository;
// ...

class FriendMessageFormType extends AbstractType
{
    private $securityContext;

    public function __construct(SecurityContext $securityContext)
    {
        $this->securityContext = $securityContext;
    }

    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('subject', 'text')
            ->add('body', 'textarea')
        ;

        // grab the user, do a quick sanity check that one exists
        $user = $this->securityContext->getToken()->getUser();
        if (!$user) {
            throw new \LogicException(
                'The FriendMessageFormType cannot be used without an authenticated user!'
            );
        }

        $builder->addEventListener(
            FormEvents::PRE_SET_DATA,
            function(FormEvent $event) use ($user) {
                $form = $event->getForm();

                $formOptions = array(
                    'class' => 'Acme\DemoBundle\Entity\User',
                    'property' => 'fullName',
                    'query_builder' => function(EntityRepository $er) use ($user) {
                        // build a custom query
                        // return $er->createQueryBuilder('u')->addOrderBy('fullName',
'DESC');

                        // or call a method on your repository that returns the query
builder
                        // the $er is an instance of your UserRepository
```

```
                        // return $er->createOrderByFullNameQueryBuilder();
                    },
                );

                // create the field, this is similar the $builder->add()
                // field name, field type, data, options
                $form->add('friend', 'entity', $formOptions);
            }
        );
    }

    // ...
}
```

The `multiple` and `expanded` form options will default to false because the type of the friend field is `entity`.

Using the Form

Our form is now ready to use and there are two possible ways to use it inside of a controller:

1. create it manually and remember to pass the security context to it;

or

2. define it as a service.

a) Creating the Form manually

This is very simple, and is probably the better approach unless you're using your new form type in many places or embedding it into other forms:

Listing 28-7
```
class FriendMessageController extends Controller
{
    public function newAction(Request $request)
    {
        $securityContext = $this->container->get('security.context');
        $form = $this->createForm(
            new FriendMessageFormType($securityContext)
        );

        // ...
    }
}
```

b) Defining the Form as a Service

To define your form as a service, just create a normal service and then tag it with *form.type*.

Listing 28-8
```
# app/config/config.yml
services:
    acme.form.friend_message:
        class: Acme\DemoBundle\Form\Type\FriendMessageFormType
        arguments: [@security.context]
        tags:
            -
```

```
            name: form.type
            alias: acme_friend_message
```

If you wish to create it from within a controller or any other service that has access to the form factory, you then use:

Listing 28-9
```php
use Symfony\Component\DependencyInjection\ContainerAware;

class FriendMessageController extends ContainerAware
{
    public function newAction(Request $request)
    {
        $form = $this->get('form.factory')->create('acme_friend_message');

        // ...
    }
}
```

If you extend the `Symfony\Bundle\FrameworkBundle\Controller\Controller` class, you can simply call:

Listing 28-10
```php
$form = $this->createForm('acme_friend_message');
```

You can also easily embed the form type into another form:

Listing 28-11
```php
// inside some other "form type" class
public function buildForm(FormBuilderInterface $builder, array $options)
{
    $builder->add('message', 'acme_friend_message');
}
```

Dynamic generation for submitted Forms

Another case that can appear is that you want to customize the form specific to the data that was submitted by the user. For example, imagine you have a registration form for sports gatherings. Some events will allow you to specify your preferred position on the field. This would be a `choice` field for example. However the possible choices will depend on each sport. Football will have attack, defense, goalkeeper etc... Baseball will have a pitcher but will not have a goalkeeper. You will need the correct options in order for validation to pass.

The meetup is passed as an entity field to the form. So we can access each sport like this:

Listing 28-12
```php
// src/Acme/DemoBundle/Form/Type/SportMeetupType.php
namespace Acme\DemoBundle\Form\Type;

use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\Form\FormEvent;
use Symfony\Component\Form\FormEvents;
// ...

class SportMeetupType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
```

```
            ->add('sport', 'entity', array(...))
        ;

        $builder->addEventListener(
            FormEvents::PRE_SET_DATA,
            function(FormEvent $event) {
                $form = $event->getForm();

                // this would be your entity, i.e. SportMeetup
                $data = $event->getData();

                $positions = $data->getSport()->getAvailablePositions();

                $form->add('position', 'entity', array('choices' => $positions));
            }
        );
    }
}
```

When you're building this form to display to the user for the first time, then this example works perfectly.

However, things get more difficult when you handle the form submission. This is because the `PRE_SET_DATA` event tells us the data that you're starting with (e.g. an empty `SportMeetup` object), *not* the submitted data.

On a form, we can usually listen to the following events:

- `PRE_SET_DATA`
- `POST_SET_DATA`
- `PRE_SUBMIT`
- `SUBMIT`
- `POST_SUBMIT`

New in version 2.3: The events `PRE_SUBMIT`, `SUBMIT` and `POST_SUBMIT` were added in Symfony 2.3. Before, they were named `PRE_BIND`, `BIND` and `POST_BIND`.

New in version 2.2.6: The behavior of the `POST_SUBMIT` event changed slightly in 2.2.6, which the below example uses.

The key is to add a `POST_SUBMIT` listener to the field that your new field depends on. If you add a `POST_SUBMIT` listener to a form child (e.g. `sport`), and add new children to the parent form, the Form component will detect the new field automatically and map it to the submitted client data.

The type would now look like:

Listing 28-13

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

// ...
use Acme\DemoBundle\Entity\Sport;
use Symfony\Component\Form\FormInterface;

class SportMeetupType extends AbstractType
{
```

```
10      public function buildForm(FormBuilderInterface $builder, array $options)
11      {
12          $builder
13              ->add('sport', 'entity', array(...))
14          ;
15
16          $formModifier = function(FormInterface $form, Sport $sport) {
17              $positions = $sport->getAvailablePositions();
18
19              $form->add('position', 'entity', array('choices' => $positions));
20          };
21
22          $builder->addEventListener(
23              FormEvents::PRE_SET_DATA,
24              function(FormEvent $event) use ($formModifier) {
25                  // this would be your entity, i.e. SportMeetup
26                  $data = $event->getData();
27
28                  $formModifier($event->getForm(), $data->getSport());
29              }
30          );
31
32          $builder->get('sport')->addEventListener(
33              FormEvents::POST_SUBMIT,
34              function(FormEvent $event) use ($formModifier) {
35                  // It's important here to fetch $event->getForm()->getData(), as
36                  // $event->getData() will get you the client data (that is, the ID)
37                  $sport = $event->getForm()->getData();
38
39                  // since we've added the listener to the child, we'll have to pass on
40                  // the parent to the callback functions!
41                  $formModifier($event->getForm()->getParent(), $sport);
42              }
43          );
44      }
45  }
```

You can see that you need to listen on these two events and have different callbacks only because in two different scenarios, the data that you can use is available in different events. Other than that, the listeners always perform exactly the same things on a given form.

One piece that may still be missing is the client-side updating of your form after the sport is selected. This should be handled by making an AJAX call back to your application. In that controller, you can submit your form, but instead of processing it, simply use the submitted form to render the updated fields. The response from the AJAX call can then be used to update the view.

Chapter 29

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, it's loosely assumed 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` association mapping definition 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 29-1

```
1  // src/Acme/TaskBundle/Entity/Task.php
2  namespace Acme\TaskBundle\Entity;
3
4  use Doctrine\Common\Collections\ArrayCollection;
5
6  class Task
7  {
8      protected $description;
9
10     protected $tags;
11
12     public function __construct()
13     {
14         $this->tags = new ArrayCollection();
15     }
16
17     public function getDescription()
18     {
19         return $this->description;
20     }
21
```

```
22      public function setDescription($description)
23      {
24          $this->description = $description;
25      }
26
27      public function getTags()
28      {
29          return $this->tags;
30      }
31  }
```

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 29-2
```
1  // src/Acme/TaskBundle/Entity/Tag.php
2  namespace Acme\TaskBundle\Entity;
3
4  class Tag
5  {
6      public $name;
7  }
```

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 29-3
```
1  // src/Acme/TaskBundle/Form/Type/TagType.php
2  namespace Acme\TaskBundle\Form\Type;
3
4  use Symfony\Component\Form\AbstractType;
5  use Symfony\Component\Form\FormBuilderInterface;
6  use Symfony\Component\OptionsResolver\OptionsResolverInterface;
7
8  class TagType extends AbstractType
9  {
10     public function buildForm(FormBuilderInterface $builder, array $options)
11     {
12         $builder->add('name');
13     }
14
15     public function setDefaultOptions(OptionsResolverInterface $resolver)
16     {
17         $resolver->setDefaults(array(
18             'data_class' => 'Acme\TaskBundle\Entity\Tag',
19         ));
20     }
21
22     public function getName()
23     {
```

```
        return 'tag';
    }
}
```

With this, you 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 you embed a collection of `TagType` forms using the *collection* field type:

Listing 29-4

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

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolverInterface;

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

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

    public function setDefaultOptions(OptionsResolverInterface $resolver)
    {
        $resolver->setDefaults(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 29-5

```php
// 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);

        $form->handleRequest($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 29-6

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

{# ... #}

{{ form_start(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_end(form) }}

{# ... #}
```

When the user submits the form, the submitted data for the `tags` field 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.

In this entry, you embed only one collection, but you are not limited to this. You can also embed nested collection as many level down as you like. But if you use Xdebug in your development setup, you may receive a `Maximum function nesting level of '100' reached, aborting!` error. This is due to the `xdebug.max_nesting_level` PHP setting, which defaults to `100`.

This directive limits recursion to 100 calls which may not be enough for rendering the form in the template if you render the whole form at once (e.g `form_widget(form)`). To fix this you can set this directive to a higher value (either via a PHP ini file or via *`ini_set`*[1], for example in `app/autoload.php`) or render each form field by hand using `form_row`.

Allowing "new" tags with the "prototype"

Allowing the user to dynamically add new tags means that you'll need to use some JavaScript. Previously you added two tags to your form in the controller. Now 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 you need to do is to let the form collection know that it will receive an unknown number of tags. So far you'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, add the `allow_add` option to your collection field:

Listing 29-7
```php
// src/Acme/TaskBundle/Form/Type/TaskType.php

// ...
use Symfony\Component\Form\FormBuilderInterface;

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

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

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 29-8
```
<ul class="tags" data-prototype="{{ form_widget(form.tags.vars.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.

1. http://php.net/manual/en/function.ini-set.php

The `form.tags.vars.prototype` is a form element that looks and feels just like the individual `form_widget(tag)` elements inside your `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 29-9
```
1 {{ form_widget(form.tags.vars.prototype.name)|e }}
```

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

Listing 29-10
```
1 <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, this example uses jQuery and assumes you have it included somewhere on your page.

Add a `script` tag somewhere on your page so you 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 you can add a new tag form (`addTagForm` will be show next):

Listing 29-11
```
1  // Get the ul that holds the collection of tags
2  var collectionHolder = $('ul.tags');
3
4  // setup an "add a tag" link
5  var $addTagLink = $('<a href="#" class="add_tag_link">Add a tag</a>');
6  var $newLinkLi = $('<li></li>').append($addTagLink);
7
8  jQuery(document).ready(function() {
9      // add the "add a tag" anchor and li to the tags ul
10     collectionHolder.append($newLinkLi);
11
12     // count the current form inputs we have (e.g. 2), use that as the new
13     // index when inserting a new item (e.g. 2)
14     collectionHolder.data('index', collectionHolder.find(':input').length);
15
16     $addTagLink.on('click', function(e) {
17         // prevent the link from creating a "#" on the URL
18         e.preventDefault();
19
20         // add a new tag form (see next code block)
21         addTagForm(collectionHolder, $newLinkLi);
22     });
23 });
```

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 you'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 29-12
```
function addTagForm(collectionHolder, $newLinkLi) {
    // Get the data-prototype explained earlier
    var prototype = collectionHolder.data('prototype');

    // get the new index
    var index = collectionHolder.data('index');

    // Replace '__name__' in the prototype's HTML to
    // instead be a number based on how many items we have
    var newForm = prototype.replace(/__name__/g, index);

    // increase the index with one for the next item
    collectionHolder.data('index', index + 1);

    // 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);
}
```

It is better to separate your javascript in real JavaScript files than to write it inside the HTML as is done here.

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

To make handling these new tags easier, add an "adder" and a "remover" method for the tags in the `Task` class:

Listing 29-13
```
// src/Acme/TaskBundle/Entity/Task.php
namespace Acme\TaskBundle\Entity;

// ...
class Task
{
    // ...

    public function addTag(Tag $tag)
    {
        $this->tags->add($tag);
    }

    public function removeTag(Tag $tag)
    {
        // ...
    }
}
```

Next, add a `by_reference` option to the `tags` field and set it to `false`:

Listing 29-14
```
// src/Acme/TaskBundle/Form/Type/TaskType.php

// ...
public function buildForm(FormBuilderInterface $builder, array $options)
{
```

```
6       // ...
7
8       $builder->add('tags', 'collection', array(
9           // ...
10          'by_reference' => false,
11      ));
12  }
```

With these two changes, when the form is submitted, each new `Tag` object is added to the `Task` class by calling the `addTag` method. Before this change, they were added internally by the form by calling `$task->getTags()->add($tag)`. That was just fine, but forcing the use of the "adder" method makes handling these new `Tag` objects easier (especially if you're using Doctrine, which we talk about next!).

If no `addTag` **and** `removeTag` method is found, the form will still use `setTag` even if `by_reference` is `false`. You'll learn more about the `removeTag` method later in this article.

Doctrine: Cascading Relations and saving the "Inverse" side

To save the new tags with 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 `Acme\TaskBundle\Entity\Task#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 29-15
```php
// src/Acme/TaskBundle/Entity/Task.php

// ...

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

A second potential issue deals with the *Owning Side and Inverse Side*[2] 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 `addTag()`, which is called by the form type since `by_reference` is set to `false`:

Listing 29-16
```php
// src/Acme/TaskBundle/Entity/Task.php

// ...
public function addTag(Tag $tag)
{
    $tag->addTask($this);

    $this->tags->add($tag);
}
```

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

Listing 29-17
```php
// 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 `addTag`.

2. `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 29-18
```php
// src/Acme/TaskBundle/Form/Type/TaskType.php

// ...
public function buildForm(FormBuilderInterface $builder, array $options)
{
    // ...

    $builder->add('tags', 'collection', array(
        // ...
        'allow_delete' => true,
    ));
}
```

Now, you need to put some code into the `removeTag` method of `Task`:

Listing 29-19
```php
// src/Acme/TaskBundle/Entity/Task.php

// ...
class Task
{
    // ...

    public function removeTag(Tag $tag)
    {
        $this->tags->removeElement($tag);
    }
}
```

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 29-20
```javascript
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 29-21
```
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 29-22

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

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

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

    $originalTags = array();

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

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

    $editForm->handleRequest($request);

    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
```

```
46              // $em->remove($tag);
47          }
48
49          $em->persist($task);
50          $em->flush();
51
52          // redirect back to some edit page
53          return $this->redirect($this->generateUrl('task_edit', array('id' => $id)));
54      }
55
56      // render some form template
57  }
```

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 30

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 you may want to create a custom form field type for a specific purpose. This recipe assumes you 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 you can customize its layout and finally, how you can register it for use in your application.

Defining the Field Type

In order to create the custom field type, first you have to create the class representing the field. In this 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 30-1

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

use Symfony\Component\Form\AbstractType;
use Symfony\Component\OptionsResolver\OptionsResolverInterface;

class GenderType extends AbstractType
{
    public function setDefaultOptions(OptionsResolverInterface $resolver)
    {
        $resolver->setDefaults(array(
            'choices' => array(
                'm' => 'Male',
                'f' => 'Female',
            )
        ));
    }

    public function getParent()
    {
```

1. http://api.symfony.com/2.3/Symfony/Component/Form/AbstractType.html

```
21          return 'choice';
22      }
23
24      public function getName()
25      {
26          return 'gender';
27      }
28  }
```

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 you're extending the `choice` field type. This means that, by default, you 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.
- `setDefaultOptions()` - 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 *form Field Type*), 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 `finishView()` 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 this 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 the parent field is `choice`, you don't *need* to do any work as the custom field type will automatically be rendered like a `choice` type. But for the sake of this example, let's suppose that when your field is "expanded" (i.e. radio buttons or checkboxes, instead of a select field), you 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 30-2

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

```twig
{# src/Acme/DemoBundle/Resources/views/Form/fields.html.twig #}
{% block gender_widget %}
    {% 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 30-3

```yaml
# 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 30-4

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

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;

class AuthorType extends AbstractType
{
    public function buildForm(FormBuilderInterface $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 you're storing the gender parameters in configuration:

Listing 30-5
```yaml
# app/config/config.yml
parameters:
    genders:
        m: Male
        f: Female
```

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

Listing 30-6
```yaml
# src/Acme/DemoBundle/Resources/config/services.yml
services:
    acme_demo.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. You'll see the importance of this in a moment when you use the custom field type. But first, add a `__construct` method to `GenderType`, which receives the gender configuration:

Listing 30-7
```php
// src/Acme/DemoBundle/Form/Type/GenderType.php
namespace Acme\DemoBundle\Form\Type;

use Symfony\Component\OptionsResolver\OptionsResolverInterface;

// ...

// ...
class GenderType extends AbstractType
{
    private $genderChoices;

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

    public function setDefaultOptions(OptionsResolverInterface $resolver)
    {
        $resolver->setDefaults(array(
            'choices' => $this->genderChoices,
        ));
    }
```

```
24
25      // ...
26  }
```

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

Listing 30-8

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

use Symfony\Component\Form\FormBuilderInterface;

// ...

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

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

Chapter 31

How to Create a Form Type Extension

Custom form field types are great when you need field types with a specific purpose, such as a gender selector, or a VAT number input.

But sometimes, you don't really need to add new field types - you want to add features on top of existing types. This is where form type extensions come in.

Form type extensions have 2 main use-cases:

1. You want to add a **generic feature to several types** (such as adding a "help" text to every field type);
2. You want to add a **specific feature to a single type** (such as adding a "download" feature to the "file" field type).

In both those cases, it might be possible to achieve your goal with custom form rendering, or custom form field types. But using form type extensions can be cleaner (by limiting the amount of business logic in templates) and more flexible (you can add several type extensions to a single form type).

Form type extensions can achieve most of what custom field types can do, but instead of being field types of their own, **they plug into existing types**.

Imagine that you manage a `Media` entity, and that each media is associated to a file. Your `Media` form uses a file type, but when editing the entity, you would like to see its image automatically rendered next to the file input.

You could of course do this by customizing how this field is rendered in a template. But field type extensions allow you to do this in a nice DRY fashion.

Defining the Form Type Extension

Your first task will be to create the form type extension class. Let's call it `ImageTypeExtension`. By standard, form extensions usually live in the `Form\Extension` directory of one of your bundles.

When creating a form type extension, you can either implement the *`FormTypeExtensionInterface`*[1] interface or extend the *`AbstractTypeExtension`*[2] class. In most cases, it's easier to extend the abstract class:

1. http://api.symfony.com/2.3/Symfony/Component/Form/FormTypeExtensionInterface.html
2. http://api.symfony.com/2.3/Symfony/Component/Form/AbstractTypeExtension.html

Listing 31-1
```php
// src/Acme/DemoBundle/Form/Extension/ImageTypeExtension.php
namespace Acme\DemoBundle\Form\Extension;

use Symfony\Component\Form\AbstractTypeExtension;

class ImageTypeExtension extends AbstractTypeExtension
{
    /**
     * Returns the name of the type being extended.
     *
     * @return string The name of the type being extended
     */
    public function getExtendedType()
    {
        return 'file';
    }
}
```

The only method you **must** implement is the `getExtendedType` function. It is used to indicate the name of the form type that will be extended by your extension.

The value you return in the `getExtendedType` method corresponds to the value returned by the `getName` method in the form type class you wish to extend.

In addition to the `getExtendedType` function, you will probably want to override one of the following methods:

- `buildForm()`
- `buildView()`
- `setDefaultOptions()`
- `finishView()`

For more information on what those methods do, you can refer to the *Creating Custom Field Types* cookbook article.

Registering your Form Type Extension as a Service

The next step is to make Symfony aware of your extension. All you need to do is to declare it as a service by using the `form.type_extension` tag:

Listing 31-2
```yaml
services:
    acme_demo_bundle.image_type_extension:
        class: Acme\DemoBundle\Form\Extension\ImageTypeExtension
        tags:
            - { name: form.type_extension, alias: file }
```

The `alias` key of the tag is the type of field that this extension should be applied to. In your case, as you want to extend the `file` field type, you will use `file` as an alias.

Adding the extension Business Logic

The goal of your extension is to display nice images next to file inputs (when the underlying model contains images). For that purpose, let's assume that you use an approach similar to the one described in *How to handle File Uploads with Doctrine*: you have a Media model with a file property (corresponding to the file field in the form) and a path property (corresponding to the image path in the database):

Listing 31-3

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

use Symfony\Component\Validator\Constraints as Assert;

class Media
{
    // ...

    /**
     * @var string The path - typically stored in the database
     */
    private $path;

    /**
     * @var \Symfony\Component\HttpFoundation\File\UploadedFile
     * @Assert\File(maxSize="2M")
     */
    public $file;

    // ...

    /**
     * Get the image url
     *
     * @return null|string
     */
    public function getWebPath()
    {
        // ... $webPath being the full image url, to be used in templates

        return $webPath;
    }
}
```

Your form type extension class will need to do two things in order to extend the `file` form type:

1. Override the `setDefaultOptions` method in order to add an image_path option;
2. Override the `buildForm` and `buildView` methods in order to pass the image url to the view.

The logic is the following: when adding a form field of type `file`, you will be able to specify a new option: `image_path`. This option will tell the file field how to get the actual image url in order to display it in the view:

Listing 31-4

```php
// src/Acme/DemoBundle/Form/Extension/ImageTypeExtension.php
namespace Acme\DemoBundle\Form\Extension;

use Symfony\Component\Form\AbstractTypeExtension;
use Symfony\Component\Form\FormView;
use Symfony\Component\Form\FormInterface;
use Symfony\Component\PropertyAccess\PropertyAccess;
use Symfony\Component\OptionsResolver\OptionsResolverInterface;
```

```
9
10  class ImageTypeExtension extends AbstractTypeExtension
11  {
12      /**
13       * Returns the name of the type being extended.
14       *
15       * @return string The name of the type being extended
16       */
17      public function getExtendedType()
18      {
19          return 'file';
20      }
21
22      /**
23       * Add the image_path option
24       *
25       * @param OptionsResolverInterface $resolver
26       */
27      public function setDefaultOptions(OptionsResolverInterface $resolver)
28      {
29          $resolver->setOptional(array('image_path'));
30      }
31
32      /**
33       * Pass the image url to the view
34       *
35       * @param FormView $view
36       * @param FormInterface $form
37       * @param array $options
38       */
39      public function buildView(FormView $view, FormInterface $form, array $options)
40      {
41          if (array_key_exists('image_path', $options)) {
42              $parentData = $form->getParent()->getData();
43
44              if (null !== $parentData) {
45                  $accessor = PropertyAccess::createPropertyAccessor();
46                  $imageUrl = $accessor->getValue($parentData, $options['image_path']);
47              } else {
48                   $imageUrl = null;
49              }
50
51              // set an "image_url" variable that will be available when rendering this field
52              $view->vars['image_url'] = $imageUrl;
53          }
54      }
55
56  }
```

Override the File Widget Template Fragment

Each field type is rendered by a template fragment. Those template fragments can be overridden in order to customize form rendering. For more information, you can refer to the *What are Form Themes?* article.

In your extension class, you have added a new variable (`image_url`), but you still need to take advantage of this new variable in your templates. Specifically, you need to override the `file_widget` block:

Listing 31-5

```twig
{# src/Acme/DemoBundle/Resources/views/Form/fields.html.twig #}
{% extends 'form_div_layout.html.twig' %}

{% block file_widget %}
    {% spaceless %}

    {{ block('form_widget') }}
    {% if image_url is not null %}
        <img src="{{ asset(image_url) }}"/>
    {% endif %}

    {% endspaceless %}
{% endblock %}
```

You will need to change your config file or explicitly specify how you want your form to be themed in order for Symfony to use your overridden block. See *What are Form Themes?* for more information.

Using the Form Type Extension

From now on, when adding a field of type `file` in your form, you can specify an `image_path` option that will be used to display an image next to the file field. For example:

Listing 31-6

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

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;

class MediaType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('name', 'text')
            ->add('file', 'file', array('image_path' => 'webPath'));
    }

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

When displaying the form, if the underlying model has already been associated with an image, you will see it displayed next to the file input.

Chapter 32

How to Reduce Code Duplication with "inherit_data"

New in version 2.3: This `inherit_data` option was known as `virtual` before Symfony 2.3.

The `inherit_data` 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 32-1
```php
// 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 32-2
```php
// src/Acme/HelloBundle/Entity/Customer.php
namespace Acme\HelloBundle\Entity;

class Customer
{
    private $firstName;
    private $lastName;

```

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

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

Let's build two forms for these entities, `CompanyType` and `CustomerType`:

Listing 32-3
```
// src/Acme/HelloBundle/Form/Type/CompanyType.php
namespace Acme\HelloBundle\Form\Type;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;

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

Listing 32-4
```
// src/Acme/HelloBundle/Form/Type/CustomerType.php
namespace Acme\HelloBundle\Form\Type;

use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\Form\AbstractType;

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

Instead of including the duplicated fields `address`, `zipcode`, `city` and `country` in both of these forms, we will create a third form for that. We will call this form simply `LocationType`:

Listing 32-5
```
// src/Acme/HelloBundle/Form/Type/LocationType.php
namespace Acme\HelloBundle\Form\Type;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolverInterface;

class LocationType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
```

```
            ->add('address', 'textarea')
            ->add('zipcode', 'text')
            ->add('city', 'text')
            ->add('country', 'text');
    }

    public function setDefaultOptions(OptionsResolverInterface $resolver)
    {
        $resolver->setDefaults(array(
            'inherit_data' => true
        ));
    }

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

The location form has an interesting option set, namely `inherit_data`. This option lets the form inherit its data from its parent form. If embedded in the company form, the fields of the location form will access the properties of the `Company` instance. If embedded in the customer form, the fields will access the properties of the `Customer` instance instead. Easy, eh?

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

Let's make this work by adding the location form to our two original forms:

Listing 32-6
```
// src/Acme/HelloBundle/Form/Type/CompanyType.php
public function buildForm(FormBuilderInterface $builder, array $options)
{
    // ...

    $builder->add('foo', new LocationType(), array(
        'data_class' => 'Acme\HelloBundle\Entity\Company'
    ));
}
```

Listing 32-7
```
// src/Acme/HelloBundle/Form/Type/CustomerType.php
public function buildForm(FormBuilderInterface $builder, array $options)
{
    // ...

    $builder->add('bar', new LocationType(), array(
        'data_class' => 'Acme\HelloBundle\Entity\Customer'
    ));
}
```

That's it! You have extracted duplicated field definitions to a separate location form that you can reuse wherever you need it.

Chapter 33

How to Unit Test your Forms

The Form Component consists of 3 core objects: a form type (implementing *FormTypeInterface*[1]), the *Form*[2] and the *FormView*[3].

The only class that is usually manipulated by programmers is the form type class which serves as a form blueprint. It is used to generate the `Form` and the `FormView`. You could test it directly by mocking its interactions with the factory but it would be complex. It is better to pass it to FormFactory like it is done in a real application. It is simple to bootstrap and you can trust the Symfony components enough to use them as a testing base.

There is already a class that you can benefit from for simple FormTypes testing: *TypeTestCase*[4]. It is used to test the core types and you can use it to test your types too.

New in version 2.3: The `TypeTestCase` has moved to the `Symfony\Component\Form\Test` namespace in 2.3. Previously, the class was located in `Symfony\Component\Form\Tests\Extension\Core\Type`.

The Basics

The simplest `TypeTestCase` implementation looks like the following:

Listing 33-1

```
1  // src/Acme/TestBundle/Tests/Form/Type/TestedTypeTests.php
2  namespace Acme\TestBundle\Tests\Form\Type;
3
4  use Acme\TestBundle\Form\Type\TestedType;
5  use Acme\TestBundle\Model\TestObject;
6  use Symfony\Component\Form\Test\TypeTestCase;
7
8  class TestedTypeTest extends TypeTestCase
```

1. http://api.symfony.com/2.3/Symfony/Component/Form/FormTypeInterface.html
2. http://api.symfony.com/2.3/Symfony/Component/Form/Form.html
3. http://api.symfony.com/2.3/Symfony/Component/Form/FormView.html
4. http://api.symfony.com/2.3/Symfony/Component/Form/Test/TypeTestCase.html

```
 9  {
10      public function testSubmitValidData()
11      {
12          $formData = array(
13              'test' => 'test',
14              'test2' => 'test2',
15          );
16
17          $type = new TestedType();
18          $form = $this->factory->create($type);
19
20          $object = new TestObject();
21          $object->fromArray($formData);
22
23          // submit the data to the form directly
24          $form->submit($formData);
25
26          $this->assertTrue($form->isSynchronized());
27          $this->assertEquals($object, $form->getData());
28
29          $view = $form->createView();
30          $children = $view->children;
31
32          foreach (array_keys($formData) as $key) {
33              $this->assertArrayHasKey($key, $children);
34          }
35      }
36  }
```

So, what does it test? Let's explain it line by line.

First you verify if the `FormType` compiles. This includes basic class inheritance, the `buildForm` function and options resolution. This should be the first test you write:

Listing 33-2
```
1  $type = new TestedType();
2  $form = $this->factory->create($type);
```

This test checks that none of your data transformers used by the form failed. The *`isSynchronized()`*[5] method is only set to `false` if a data transformer throws an exception:

Listing 33-3
```
1  $form->submit($formData);
2  $this->assertTrue($form->isSynchronized());
```

Don't test the validation: it is applied by a listener that is not active in the test case and it relies on validation configuration. Instead, unit test your custom constraints directly.

Next, verify the submission and mapping of the form. The test below checks if all the fields are correctly specified:

Listing 33-4
```
1  $this->assertEquals($object, $form->getData());
```

5. http://api.symfony.com/2.3/Symfony/Component/Form/FormInterface.html#isSynchronized()

Finally, check the creation of the `FormView`. You should check if all widgets you want to display are available in the children property:

Listing 33-5
```php
$view = $form->createView();
$children = $view->children;

foreach (array_keys($formData) as $key) {
    $this->assertArrayHasKey($key, $children);
}
```

Adding a Type your Form depends on

Your form may depend on other types that are defined as services. It might look like this:

Listing 33-6
```php
// src/Acme/TestBundle/Form/Type/TestedType.php

// ... the buildForm method
$builder->add('acme_test_child_type');
```

To create your form correctly, you need to make the type available to the form factory in your test. The easiest way is to register it manually before creating the parent form using PreloadedExtension class:

Listing 33-7
```php
// src/Acme/TestBundle/Tests/Form/Type/TestedTypeTests.php
namespace Acme\TestBundle\Tests\Form\Type;

use Acme\TestBundle\Form\Type\TestedType;
use Acme\TestBundle\Model\TestObject;
use Symfony\Component\Form\Test\TypeTestCase;
use Symfony\Component\Form\PreloadedExtension;

class TestedTypeTest extends TypeTestCase
{
    protected function getExtensions()
    {
        $childType = new TestChildType();
        return array(new PreloadedExtension(array(
            $childType->getName() => $childType,
        ), array()));
    }

    public function testSubmitValidData()
    {
        $type = new TestedType();
        $form = $this->factory->create($type);

        // ... your test
    }
}
```

Make sure the child type you add is well tested. Otherwise you may be getting errors that are not related to the form you are currently testing but to its children.

Adding custom Extensions

It often happens that you use some options that are added by *form extensions*. One of the cases may be the `ValidatorExtension` with its `invalid_message` option. The `TypeTestCase` loads only the core form extension so an "Invalid option" exception will be raised if you try to use it for testing a class that depends on other extensions. You need add those extensions to the factory object:

Listing 33-8

```php
// src/Acme/TestBundle/Tests/Form/Type/TestedTypeTests.php
namespace Acme\TestBundle\Tests\Form\Type;

use Acme\TestBundle\Form\Type\TestedType;
use Acme\TestBundle\Model\TestObject;
use Symfony\Component\Form\Test\TypeTestCase;

class TestedTypeTest extends TypeTestCase
{
    protected function setUp()
    {
        parent::setUp();

        $this->factory = Forms::createFormFactoryBuilder()
            ->addExtensions($this->getExtensions())
            ->addTypeExtension(
                new FormTypeValidatorExtension(
                    $this->getMock('Symfony\Component\Validator\ValidatorInterface')
                )
            )
            ->addTypeGuesser(
                $this->getMockBuilder(
                    'Symfony\Component\Form\Extension\Validator\ValidatorTypeGuesser'
                )
                    ->disableOriginalConstructor()
                    ->getMock()
            )
            ->getFormFactory();

        $this->dispatcher =
$this->getMock('Symfony\Component\EventDispatcher\EventDispatcherInterface');
        $this->builder = new FormBuilder(null, null, $this->dispatcher, $this->factory);
    }

    // ... your tests
}
```

Testing against different Sets of Data

If you are not familiar yet with PHPUnit's *data providers*[6], this might be a good opportunity to use them:

Listing 33-9

```php
// src/Acme/TestBundle/Tests/Form/Type/TestedTypeTests.php
namespace Acme\TestBundle\Tests\Form\Type;

use Acme\TestBundle\Form\Type\TestedType;
use Acme\TestBundle\Model\TestObject;
```

6. http://www.phpunit.de/manual/current/en/writing-tests-for-phpunit.html#writing-tests-for-phpunit.data-providers

```php
use Symfony\Component\Form\Test\TypeTestCase;

class TestedTypeTest extends TypeTestCase
{

    /**
     * @dataProvider getValidTestData
     */
    public function testForm($data)
    {
        // ... your test
    }

    public function getValidTestData()
    {
        return array(
            array(
                'data' => array(
                    'test' => 'test',
                    'test2' => 'test2',
                ),
            ),
            array(
                'data' => array(),
            ),
            array(
                'data' => array(
                    'test' => null,
                    'test2' => null,
                ),
            ),
        );
    }
}
```

The code above will run your test three times with 3 different sets of data. This allows for decoupling the test fixtures from the tests and easily testing against multiple sets of data.

You can also pass another argument, such as a boolean if the form has to be synchronized with the given set of data or not etc.

Chapter 34

How to configure Empty Data for a Form Class

The `empty_data` option allows you to specify an empty data set for your form class. This empty data set would be used if you submit your form, but haven't called `setData()` on your form or passed in data when you created you form. For example:

Listing 34-1
```php
public function indexAction()
{
    $blog = ...;

    // $blog is passed in as the data, so the empty_data option is not needed
    $form = $this->createForm(new BlogType(), $blog);

    // no data is passed in, so empty_data is used to get the "starting data"
    $form = $this->createForm(new BlogType());
}
```

By default, `empty_data` is set to `null`. Or, if you have specified a `data_class` option for your form class, it will default to a new instance of that class. That instance will be created by calling the constructor with no arguments.

If you want to override this default behavior, there are two ways to do this.

Option 1: Instantiate a new Class

One reason you might use this option is if you want to use a constructor that takes arguments. Remember, the default `data_class` option calls that constructor with no arguments:

Listing 34-2
```php
// src/Acme/DemoBundle/Form/Type/BlogType.php

// ...
use Symfony\Component\Form\AbstractType;
use Acme\DemoBundle\Entity\Blog;
use Symfony\Component\OptionsResolver\OptionsResolverInterface;

```

```php
class BlogType extends AbstractType
{
    private $someDependency;

    public function __construct($someDependency)
    {
        $this->someDependency = $someDependency;
    }
    // ...

    public function setDefaultOptions(OptionsResolverInterface $resolver)
    {
        $resolver->setDefaults(array(
            'empty_data' => new Blog($this->someDependency),
        ));
    }
}
```

You can instantiate your class however you want. In this example, we pass some dependency into the `BlogType` when we instantiate it, then use that to instantiate the `Blog` object. The point is, you can set `empty_data` to the exact "new" object that you want to use.

Option 2: Provide a Closure

Using a closure is the preferred method, since it will only create the object if it is needed.

The closure must accept a `FormInterface` instance as the first argument:

Listing 34-3

```php
use Symfony\Component\OptionsResolver\OptionsResolverInterface;
use Symfony\Component\Form\FormInterface;
// ...

public function setDefaultOptions(OptionsResolverInterface $resolver)
{
    $resolver->setDefaults(array(
        'empty_data' => function (FormInterface $form) {
            return new Blog($form->get('title')->getData());
        },
    ));
}
```

Chapter 35

How to use the submit() Function to handle Form Submissions

New in version 2.3: The `handleRequest()` method was added in Symfony 2.3.

In Symfony 2.3, a new *`handleRequest()`*[1] method was added, which makes handling form submissions easier than ever:

Listing 35-1

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

public function newAction(Request $request)
{
    $form = $this->createFormBuilder()
        // ...
        ->getForm();

    $form->handleRequest($request);

    if ($form->isValid()) {
        // perform some action...

        return $this->redirect($this->generateUrl('task_success'));
    }

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

1. http://api.symfony.com/2.3/SymfonyComponentFormFormInterface.html#handleRequest()

To see more about this method, read *Handling Form Submissions*.

Calling Form::submit() manually

New in version 2.3: Before Symfony 2.3, the `submit()` method was known as `bind()`.

In some cases, you want better control over when exactly your form is submitted and what data is passed to it. Instead of using the *`handleRequest()`*[2] method, pass the submitted data directly to *`submit()`*[3]:

Listing 35-2

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

public function newAction(Request $request)
{
    $form = $this->createFormBuilder()
        // ...
        ->getForm();

    if ($request->isMethod('POST')) {
        $form->submit($request->request->get($form->getName()));

        if ($form->isValid()) {
            // perform some action...

            return $this->redirect($this->generateUrl('task_success'));
        }
    }

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

Forms consisting of nested fields expect an array in *`submit()`*[4]. You can also submit individual fields by calling *`submit()`*[5] directly on the field:

Listing 35-3

```
$form->get('firstName')->submit('Fabien');
```

2. http://api.symfony.com/2.3/SymfonyComponentFormFormInterface.html#handleRequest()
3. http://api.symfony.com/2.3/SymfonyComponentFormFormInterface.html#submit()
4. http://api.symfony.com/2.3/SymfonyComponentFormFormInterface.html#submit()
5. http://api.symfony.com/2.3/SymfonyComponentFormFormInterface.html#submit()

Passing a Request to Form::submit() (deprecated)

New in version Before: Symfony 2.3, the `submit` method was known as `bind`.

Before Symfony 2.3, the *`submit()`*[6] method accepted a *`Request`*[7] object as a convenient shortcut to the previous example:

Listing 35-4

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

public function newAction(Request $request)
{
    $form = $this->createFormBuilder()
        // ...
        ->getForm();

    if ($request->isMethod('POST')) {
        $form->submit($request);

        if ($form->isValid()) {
            // perform some action...

            return $this->redirect($this->generateUrl('task_success'));
        }
    }

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

Passing the *`Request`*[8] directly to *`submit()`*[9] still works, but is deprecated and will be removed in Symfony 3.0. You should use the method *`handleRequest()`*[10] instead.

6. http://api.symfony.com/2.3/SymfonyComponentFormFormInterface.html#submit()
7. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Request.html
8. http://api.symfony.com/2.3/Symfony/ComponentHttpFoundation/Request.html
9. http://api.symfony.com/2.3/Symfony/Component/Form/FormInterface.html#submit()
10. http://api.symfony.com/2.3/SymfonyComponentFormFormInterface.html#handleRequest()

Chapter 36

How to use the Virtual Form Field Option

As of Symfony 2.3, the `virtual` option is renamed to `inherit_data`. You can read everything about the new option in "*How to Reduce Code Duplication with "inherit_data"*".

Chapter 37

How to create a Custom Validation Constraint

You can create a custom constraint by extending the base constraint class, *Constraint*[1]. As an example you're going to create a simple validator that checks if a string contains only alphanumeric characters.

Creating Constraint class

First you need to create a Constraint class and extend *Constraint*[2]:

Listing 37-1

```
// src/Acme/DemoBundle/Validator/Constraints/ContainsAlphanumeric.php
namespace Acme\DemoBundle\Validator\Constraints;

use Symfony\Component\Validator\Constraint;

/**
 * @Annotation
 */
class ContainsAlphanumeric extends Constraint
{
    public $message = 'The string "%string%" contains an illegal character: it can only
contain letters or numbers.';
}
```

The `@Annotation` annotation is necessary for this new constraint in order to make it available for use in classes via annotations. Options for your constraint are represented as public properties on the constraint class.

1. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraint.html
2. http://api.symfony.com/2.3/Symfony/Component/Validator/Constraint.html

Creating the Validator itself

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

Listing 37-2
```php
// 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.

The validator class is also simple, and only has one required method `validate()`:

Listing 37-3
```php
// src/Acme/DemoBundle/Validator/Constraints/ContainsAlphanumericValidator.php
namespace Acme\DemoBundle\Validator\Constraints;

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

class ContainsAlphanumericValidator extends ConstraintValidator
{
    public function validate($value, Constraint $constraint)
    {
        if (!preg_match('/^[a-zA-Za0-9]+$/', $value, $matches)) {
            $this->context->addViolation($constraint->message, array('%string%' =>
$value));
        }
    }
}
```

The `validate` method does not return a value; instead, it adds violations to the validator's `context` property with an `addViolation` method call if there are validation failures. Therefore, a value could be considered as being valid if it causes no violations to be added to the context. The first parameter of the `addViolation` call is the error message to use for that violation.

Using the new Validator

Using custom validators is very easy, just as the ones provided by Symfony2 itself:

Listing 37-4
```yaml
# src/Acme/BlogBundle/Resources/config/validation.yml
Acme\DemoBundle\Entity\AcmeEntity:
    properties:
        name:
            - NotBlank: ~
            - Acme\DemoBundle\Validator\Constraints\ContainsAlphanumeric: ~
```

If your constraint contains options, then they should be public properties on the custom Constraint class you created earlier. These options can be configured like options on core Symfony constraints.

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 37-5
```yaml
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 37-6
```php
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 in its `Constraint` class:

Listing 37-7
```php
public function getTargets()
{
    return self::CLASS_CONSTRAINT;
}
```

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

Listing 37-8
```php
class ProtocolClassValidator extends ConstraintValidator
{
    public function validate($protocol, Constraint $constraint)
    {
        if ($protocol->getFoo() != $protocol->getBar()) {
            $this->context->addViolationAt('foo', $constraint->message, array(), null);
        }
    }
}
```

Note that a class constraint validator is applied to the class itself, and not to the property:

Listing 37-9
```yaml
# src/Acme/BlogBundle/Resources/config/validation.yml
Acme\DemoBundle\Entity\AcmeEntity:
    constraints:
        - Acme\DemoBundle\Validator\Constraints\ContainsAlphanumeric: ~
```

Chapter 38

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 38-1

```
// 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 38-2
```yaml
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 38-3
```yaml
# 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 38-4
```
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 38-5
```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 the 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 38-6
```
1  doctrine:
2     dbal:
3         logging:  "%kernel.debug%"
4         # ...
```

As of Symfony 2.3, showing errors or not no longer depends on the debug mode. You'll need to enable that in your front controller by calling *`enable()`*[1].

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 38-7
```
1  # app/config/config_benchmark.yml
2  imports:
3      - { resource: config_prod.yml }
4
5  framework:
6      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.

1. http://api.symfony.com/2.3/Symfony/Component/Debug/Debug.html#enable()

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 38-8
```php
<?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 38-9
```
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 38-10
```
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 38-11
```
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.

You can easily change the directory location and name. For more information read the article *How to override Symfony's Default Directory Structure*.

Going Further

Read the article on *How to Set External Parameters in the Service Container*.

Chapter 39

How to override Symfony's Default Directory Structure

Symfony automatically ships with a default directory structure. You can easily override this directory structure to create your own. The default directory structure is:

Listing 39-1
```
app/
    cache/
    config/
    logs/
    ...
src/
    ...
vendor/
    ...
web/
    app.php
    ...
```

Override the `cache` directory

You can override the cache directory by overriding the `getCacheDir` method in the `AppKernel` class of you application:

Listing 39-2
```
// app/AppKernel.php

// ...
class AppKernel extends Kernel
{
    // ...

    public function getCacheDir()
```

```
    {
        return $this->rootDir.'/'.$this->environment.'/cache';
    }
}
```

`$this->rootDir` is the absolute path to the `app` directory and `$this->environment` is the current environment (i.e. `dev`). In this case you have changed the location of the cache directory to `app/{environment}/cache`.

You should keep the `cache` directory different for each environment, otherwise some unexpected behaviour may happen. Each environment generates its own cached config files, and so each needs its own directory to store those cache files.

Override the `logs` directory

Overriding the `logs` directory is the same as overriding the `cache` directory, the only difference is that you need to override the `getLogDir` method:

Listing 39-3
```
// app/AppKernel.php

// ...
class AppKernel extends Kernel
{
    // ...

    public function getLogDir()
    {
        return $this->rootDir.'/'.$this->environment.'/logs';
    }
}
```

Here you have changed the location of the directory to `app/{environment}/logs`.

Override the `web` directory

If you need to rename or move your `web` directory, the only thing you need to guarantee is that the path to the `app` directory is still correct in your `app.php` and `app_dev.php` front controllers. If you simply renamed the directory, you're fine. But if you moved it in some way, you may need to modify the paths inside these files:

Listing 39-4
```
require_once __DIR__.'/../Symfony/app/bootstrap.php.cache';
require_once __DIR__.'/../Symfony/app/AppKernel.php';
```

Since Symfony 2.1 (in which Composer is introduced), you also need to change the `extra.symfony-web-dir` option in the `composer.json` file:

Listing 39-5
```
{
    ...
    "extra": {
        ...
        "symfony-web-dir": "my_new_web_dir"
```

```
    }
}
```

Some shared hosts have a `public_html` web directory root. Renaming your web directory from `web` to `public_html` is one way to make your Symfony project work on your shared host. Another way is to deploy your application to a directory outside of your web root, delete your `public_html` directory, and then replace it with a symbolic link to the `web` in your project.

If you use the AsseticBundle you need to configure this, so it can use the correct `web` directory:

Listing 39-6
```yaml
# app/config/config.yml

# ...
assetic:
    # ...
    read_from: "%kernel.root_dir%/../../public_html"
```

Now you just need to dump the assets again and your application should work:

Listing 39-7
```
$ php app/console assetic:dump --env=prod --no-debug
```

Chapter 40

Understanding how the Front Controller, Kernel and Environments work together

The section *How to Master and Create new Environments* explained the basics on how Symfony uses environments to run your application with different configuration settings. This section will explain a bit more in-depth what happens when your application is bootstrapped. To hook into this process, you need to understand three parts that work together:

- The Front Controller
- The Kernel Class
- The Environments

Usually, you will not need to define your own front controller or `AppKernel` class as the *Symfony2 Standard Edition*[1] provides sensible default implementations.

This documentation section is provided to explain what is going on behind the scenes.

The Front Controller

The *front controller*[2] is a well-known design pattern; it is a section of code that *all* requests served by an application run through.

In the *Symfony2 Standard Edition*[3], this role is taken by the *app.php*[4] and *app_dev.php*[5] files in the `web/` directory. These are the very first PHP scripts executed when a request is processed.

The main purpose of the front controller is to create an instance of the `AppKernel` (more on that in a second), make it handle the request and return the resulting response to the browser.

1. https://github.com/symfony/symfony-standard
2. http://en.wikipedia.org/wiki/Front_Controller_pattern
3. https://github.com/symfony/symfony-standard
4. https://github.com/symfony/symfony-standard/blob/master/web/app.php
5. https://github.com/symfony/symfony-standard/blob/master/web/app_dev.php

Because every request is routed through it, the front controller can be used to perform global initializations prior to setting up the kernel or to *decorate*[6] the kernel with additional features. Examples include:

- Configuring the autoloader or adding additional autoloading mechanisms;
- Adding HTTP level caching by wrapping the kernel with an instance of *AppCache*;
- Enabling (or skipping) the *ClassCache*
- Enabling the *Debug Component*.

The front controller can be chosen by requesting URLs like:

Listing 40-1
```
1 http://localhost/app_dev.php/some/path/...
```

As you can see, this URL contains the PHP script to be used as the front controller. You can use that to easily switch the front controller or use a custom one by placing it in the `web/` directory (e.g. `app_cache.php`).

When using Apache and the *RewriteRule shipped with the Standard Edition*[7], you can omit the filename from the URL and the RewriteRule will use `app.php` as the default one.

Pretty much every other web server should be able to achieve a behavior similar to that of the RewriteRule described above. Check your server documentation for details or see *Configuring a web server*.

Make sure you appropriately secure your front controllers against unauthorized access. For example, you don't want to make a debugging environment available to arbitrary users in your production environment.

Technically, the *app/console*[8] script used when running Symfony on the command line is also a front controller, only that is not used for web, but for command line requests.

The Kernel Class

The *`Kernel`*[9] is the core of Symfony2. It is responsible for setting up all the bundles that make up your application and providing them with the application's configuration. It then creates the service container before serving requests in its *`handle()`*[10] method.

There are two methods declared in the *`KernelInterface`*[11] that are left unimplemented in *`Kernel`*[12] and thus serve as *template methods*[13]:

- *`registerBundles()`*[14], which must return an array of all bundles needed to run the application;
- *`registerContainerConfiguration()`*[15], which loads the application configuration.

6. http://en.wikipedia.org/wiki/Decorator_pattern
7. https://github.com/symfony/symfony-standard/blob/master/web/.htaccess)
8. https://github.com/symfony/symfony-standard/blob/master/app/console
9. http://api.symfony.com/2.3/Symfony/Component/HttpKernel/Kernel.html
10. http://api.symfony.com/2.3/Symfony/Component/HttpKernel/HttpKernelInterface.html#handle()
11. http://api.symfony.com/2.3/Symfony/Component/HttpKernel/KernelInterface.html
12. http://api.symfony.com/2.3/Symfony/Component/HttpKernel/Kernel.html
13. http://en.wikipedia.org/wiki/Template_method_pattern
14. http://api.symfony.com/2.3/Symfony/Component/HttpKernel/KernelInterface.html#registerBundles()
15. http://api.symfony.com/2.3/Symfony/Component/HttpKernel/KernelInterface.html#registerContainerConfiguration()

To fill these (small) blanks, your application needs to subclass the Kernel and implement these methods. The resulting class is conventionally called the `AppKernel`.

Again, the Symfony2 Standard Edition provides an *AppKernel*[16] in the `app/` directory. This class uses the name of the environment - which is passed to the Kernel's *`constructor`*[17] method and is available via *`getEnvironment()`*[18] - to decide which bundles to create. The logic for that is in `registerBundles()`, a method meant to be extended by you when you start adding bundles to your application.

You are, of course, free to create your own, alternative or additional `AppKernel` variants. All you need is to adapt your (or add a new) front controller to make use of the new kernel.

The name and location of the `AppKernel` is not fixed. When putting multiple Kernels into a single application, it might therefore make sense to add additional sub-directories, for example `app/admin/AdminKernel.php` and `app/api/ApiKernel.php`. All that matters is that your front controller is able to create an instance of the appropriate kernel.

Having different `AppKernels` might be useful to enable different front controllers (on potentially different servers) to run parts of your application independently (for example, the admin UI, the frontend UI and database migrations).

There's a lot more the `AppKernel` can be used for, for example *overriding the default directory structure*. But odds are high that you don't need to change things like this on the fly by having several `AppKernel` implementations.

The Environments

We just mentioned another method the `AppKernel` has to implement - *`registerContainerConfiguration()`*[19]. This method is responsible for loading the application's configuration from the right *environment*.

Environments have been covered extensively *in the previous chapter*, and you probably remember that the Standard Edition comes with three of them - `dev`, `prod` and `test`.

More technically, these names are nothing more than strings passed from the front controller to the `AppKernel`'s constructor. This name can then be used in the *`registerContainerConfiguration()`*[20] method to decide which configuration files to load.

The Standard Edition's *AppKernel*[21] class implements this method by simply loading the `app/config/config_*environment*.yml` file. You are, of course, free to implement this method differently if you need a more sophisticated way of loading your configuration.

16. https://github.com/symfony/symfony-standard/blob/master/app/AppKernel.php
17. http://api.symfony.com/2.3/Symfony/Component/HttpKernel/Kernel.html#__construct()
18. http://api.symfony.com/2.3/Symfony/Component/HttpKernel/Kernel.html#getEnvironment()
19. http://api.symfony.com/2.3/Symfony/Component/HttpKernel/KernelInterface.html#registerContainerConfiguration()
20. http://api.symfony.com/2.3/Symfony/Component/HttpKernel/KernelInterface.html#registerContainerConfiguration()
21. https://github.com/symfony/symfony-standard/blob/master/app/AppKernel.php

Chapter 41

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 41-1

```
<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 41-2
```
$ 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 41-3
```
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. See *Constants as Parameters* for more details.

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`.

Listing 41-4
```
# app/config/config.yml
imports:
    - { resource: parameters.php }
```

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.

Listing 41-5

1. `http://httpd.apache.org/docs/current/env.html`

```php
// app/config/parameters.php
include_once('/path/to/drupal/sites/default/settings.php');
$container->setParameter('drupal.database.url', $db_url);
```

Chapter 42

How to use PdoSessionHandler 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 *`PdoSessionHandler`*[1]. To use it, you just need to change some parameters in `config.yml` (or the configuration format of your choice):

New in version 2.1: In Symfony2.1 the class and namespace are slightly modified. You can now find the session storage classes in the *Session\Storage* namespace: `Symfony\Component\HttpFoundation\Session\Storage`. Also note that in Symfony2.1 you should configure `handler_id` not `storage_id` like in Symfony2.0. Below, you'll notice that `%session.storage.options%` is not used anymore.

Listing 42-1

```yaml
# app/config/config.yml
framework:
    session:
        # ...
        handler_id:     session.handler.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
```

1. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Session/Storage/Handler/PdoSessionHandler.html

```
        arguments:
            dsn:      "mysql:dbname=mydatabase"
            user:     myuser
            password: mypassword
        calls:
            - [setAttribute, [3, 2]] # \PDO::ATTR_ERRMODE, \PDO::ERRMODE_EXCEPTION

    session.handler.pdo:
        class:
Symfony\Component\HttpFoundation\Session\Storage\Handler\PdoSessionHandler
        arguments: ["@pdo", "%pdo.db_options%"]
```

- `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 42-2
```
pdo:
    class: PDO
    arguments:
        - "mysql:host=%database_host%;port=%database_port%;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 42-3
```
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 42-4

```
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)
);
```

Microsoft SQL Server

For MSSQL, the statement might look like the following:

Listing 42-5

```
CREATE TABLE [dbo].[session](
    [session_id] [nvarchar](255) NOT NULL,
    [session_value] [ntext] NOT NULL,
    [session_time] [int] NOT NULL,
    PRIMARY KEY CLUSTERED(
        [session_id] ASC
    ) WITH (
        PAD_INDEX  = OFF,
        STATISTICS_NORECOMPUTE  = OFF,
        IGNORE_DUP_KEY = OFF,
        ALLOW_ROW_LOCKS  = ON,
        ALLOW_PAGE_LOCKS  = ON
    ) ON [PRIMARY]
) ON [PRIMARY] TEXTIMAGE_ON [PRIMARY]
```

Chapter 43

How to use the Apache Router

Symfony2, while fast out of the box, also provides various ways to increase that speed with a little bit of tweaking. One of these ways is by letting apache handle routes directly, rather than using Symfony2 for this task.

Change Router Configuration Parameters

To dump Apache routes you must first tweak some configuration parameters to tell Symfony2 to use the `ApacheUrlMatcher` instead of the default one:

Listing 43-1
```yaml
# app/config/config_prod.yml
parameters:
    router.options.matcher.cache_class: ~ # disable router cache
    router.options.matcher_class: Symfony\Component\Routing\Matcher\ApacheUrlMatcher
```

Note that *ApacheUrlMatcher*[1] extends *UrlMatcher*[2] so even if you don't regenerate the url_rewrite rules, everything will work (because at the end of `ApacheUrlMatcher::match()` a call to `parent::match()` is done).

Generating mod_rewrite rules

To test that it's working, let's create a very basic route for demo bundle:

Listing 43-2
```yaml
# app/config/routing.yml
hello:
    path:  /hello/{name}
    defaults: { _controller: AcmeDemoBundle:Demo:hello }
```

1. http://api.symfony.com/2.3/Symfony/Component/Routing/Matcher/ApacheUrlMatcher.html
2. http://api.symfony.com/2.3/Symfony/Component/Routing/Matcher/UrlMatcher.html

Now generate **url_rewrite** rules:

Listing 43-3
```
$ php app/console router:dump-apache -e=prod --no-debug
```

Which should roughly output the following:

Listing 43-4
```
# skip "real" requests
RewriteCond %{REQUEST_FILENAME} -f
RewriteRule .* - [QSA,L]

# hello
RewriteCond %{REQUEST_URI} ^/hello/([^/]+?)$
RewriteRule .* app.php
[QSA,L,E=_ROUTING__route:hello,E=_ROUTING_name:%1,E=_ROUTING__controller:AcmeDemoBundle\:Demo\:hello]
```

You can now rewrite *web/.htaccess* to use the new rules, so with this example it should look like this:

Listing 43-5
```
<IfModule mod_rewrite.c>
    RewriteEngine On

    # skip "real" requests
    RewriteCond %{REQUEST_FILENAME} -f
    RewriteRule .* - [QSA,L]

    # hello
    RewriteCond %{REQUEST_URI} ^/hello/([^/]+?)$
    RewriteRule .* app.php
[QSA,L,E=_ROUTING__route:hello,E=_ROUTING_name:%1,E=_ROUTING__controller:AcmeDemoBundle\:Demo\:hello]
</IfModule>
```

Procedure above should be done each time you add/change a route if you want to take full advantage of this setup

That's it! You're now all set to use Apache Route rules.

Additional tweaks

To save a little bit of processing time, change occurrences of `Request` to `ApacheRequest` in `web/app.php`:

Listing 43-6
```
// web/app.php

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

use Symfony\Component\HttpFoundation\ApacheRequest;

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

Chapter 44
Configuring a web server

The web directory is the home of all of your application's public and static files. Including images, stylesheets and JavaScript files. It is also where the front controllers live. For more details, see the *The Web Directory*.

The web directory services as the document root when configuring your web server. In the examples below, this directory is in `/var/www/project/web/`.

Apache2

For advanced Apache configuration options, see the official *Apache*[1] documentation. The minimum basics to get your application running under Apache2 are:

Listing 44-1
```
<VirtualHost *:80>
    ServerName domain.tld
    ServerAlias www.domain.tld

    DocumentRoot /var/www/project/web
    <Directory /var/www/project/web>
        # enable the .htaccess rewrites
        AllowOverride All
        Order allow,deny
        Allow from All
    </Directory>

    ErrorLog /var/log/apache2/project_error.log
    CustomLog /var/log/apache2/project_access.log combined
</VirtualHost>
```

For performance reasons, you will probably want to set `AllowOverride None` and implement the rewrite rules in the `web/.htaccess` into the virtualhost config.

1. `http://httpd.apache.org/docs/current/mod/core.html#documentroot`

If you are using **php-cgi**, Apache does not pass HTTP basic username and password to PHP by default. To work around this limitation, you should use the following configuration snippet:

Listing 44-2
```
RewriteRule .* - [E=HTTP_AUTHORIZATION:%{HTTP:Authorization}]
```

Nginx

For advanced Nginx configuration options, see the official *Nginx*[2] documentation. The minimum basics to get your application running under Nginx are:

Listing 44-3
```
server {
    server_name domain.tld www.domain.tld;
    root /var/www/project/web;

    location / {
        # try to serve file directly, fallback to rewrite
        try_files $uri @rewriteapp;
    }

    location @rewriteapp {
        # rewrite all to app.php
        rewrite ^(.*)$ /app.php/$1 last;
    }

    location ~ ^/(app|app_dev|config)\.php(/|$) {
        fastcgi_pass unix:/var/run/php5-fpm.sock;
        fastcgi_split_path_info ^(.+\.php)(/.*)$;
        include fastcgi_params;
        fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
        fastcgi_param HTTPS off;
    }

    error_log /var/log/nginx/project_error.log;
    access_log /var/log/nginx/project_access.log;
}
```

Depending on your PHP-FPM config, the `fastcgi_pass` can also be `fastcgi_pass 127.0.0.1:9000`.

This executes **only** `app.php`, `app_dev.php` and `config.php` in the web directory. All other files will be served as text. You **must** also make sure that if you *do* deploy `app_dev.php` or `config.php` that these files are secured and not available to any outside user (the IP checking code at the top of each file does this by default).

If you have other PHP files in your web directory that need to be executed, be sure to include them in the `location` block above.

2. `http://wiki.nginx.org/Symfony`

Chapter 45

How to use the Serializer

Serializing and deserializing to and from objects and different formats (e.g. JSON or XML) is a very complex topic. Symfony comes with a *Serializer Component*, which gives you some tools that you can leverage for your solution.

In fact, before you start, get familiar with the serializer, normalizers and encoders by reading the *Serializer Component*. You should also check out the *JMSSerializerBundle*[1], which expands on the functionality offered by Symfony's core serializer.

Activating the Serializer

New in version 2.3: The Serializer has always existed in Symfony, but prior to Symfony 2.3, you needed to build the `serializer` service yourself.

The `serializer` service is not available by default. To turn it on, activate it in your configuration:

Listing 45-1

```yaml
# app/config/config.yml
framework:
    # ...
    serializer:
        enabled: true
```

Adding Normalizers and Encoders

Once enabled, the `serializer` service will be available in the container and will be loaded with two *encoders* (`JsonEncoder`[2] and `XmlEncoder`[3]) but no *normalizers*, meaning you'll need to load your own.

1. http://jmsyst.com/bundles/JMSSerializerBundle
2. http://api.symfony.com/2.3/Symfony/Component/Serializer/Encoder/JsonEncoder.html
3. http://api.symfony.com/2.3/Symfony/Component/Serializer/Encoder/XmlEncoder.html

You can load normalizers and/or encoders by tagging them as *serializer.normalizer* and *serializer.encoder*. It's also possible to set the priority of the tag in order to decide the matching order.

Here an example on how to load the load the *GetSetMethodNormalizer*[4]:

Listing 45-2
```yaml
# app/config/config.yml
services:
    get_set_method_normalizer:
        class: Symfony\Component\Serializer\Normalizer\GetSetMethodNormalizer
        tags:
            - { name: serializer.normalizer }
```

The *GetSetMethodNormalizer*[5] is broken by design. As soon as you have a circular object graph, an infinite loop is created when calling the getters. You're encouraged to add your own normalizers that fit your use-case.

4. http://api.symfony.com/2.3/Symfony/Component/Serializer/Normalizer/GetSetMethodNormalizer.html
5. http://api.symfony.com/2.3/Symfony/Component/Serializer/Normalizer/GetSetMethodNormalizer.html

Chapter 46

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, you will create a service that will act as an Exception Listener, allowing you to modify how exceptions are shown by your application. The `KernelEvents::EXCEPTION` event is just one of the core kernel events:

Listing 46-1

```php
// src/Acme/DemoBundle/EventListener/AcmeExceptionListener.php
namespace Acme\DemoBundle\EventListener;

use Symfony\Component\HttpKernel\Event\GetResponseForExceptionEvent;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\HttpKernel\Exception\HttpExceptionInterface;

class AcmeExceptionListener
{
    public function onKernelException(GetResponseForExceptionEvent $event)
    {
        // You get the exception object from the received event
        $exception = $event->getException();
        $message = sprintf(
            'My Error says: %s with code: %s',
            $exception->getMessage(),
            $exception->getCode()
        );

        // Customize your response object to display the exception details
        $response = new Response();
        $response->setContent($message);

        // HttpExceptionInterface is a special type of exception that
        // holds status code and header details
        if ($exception instanceof HttpExceptionInterface) {
```

1. http://api.symfony.com/2.3/Symfony/Component/HttpKernel/KernelEvents.html

```
            $response->setStatusCode($exception->getStatusCode());
            $response->headers->replace($exception->getHeaders());
        } else {
            $response->setStatusCode(500);
        }

        // Send the 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].

Now that the class is created, you just need to register it as a service and notify Symfony that it is a "listener" on the `kernel.exception` event by using a special "tag":

Listing 46-2
```
# app/config/config.yml
services:
    kernel.listener.your_listener_name:
        class: Acme\DemoBundle\EventListener\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.

Request events, checking types

A single page can make several requests (one master request, and then multiple sub-requests), which is why when working with the `KernelEvents::REQUEST` event, you might need to check the type of the request. This can be easily done as follow:

Listing 46-3
```
// src/Acme/DemoBundle/EventListener/AcmeRequestListener.php
namespace Acme\DemoBundle\EventListener;

use Symfony\Component\HttpKernel\Event\GetResponseEvent;
use Symfony\Component\HttpKernel\HttpKernel;

class AcmeRequestListener
{
    public function onKernelRequest(GetResponseEvent $event)
    {
        if (HttpKernel::MASTER_REQUEST != $event->getRequestType()) {
            // don't do anything if it's not the master request
```

2. http://api.symfony.com/2.3/Symfony/Component/HttpKernel/Event/GetResponseForExceptionEvent.html
3. http://api.symfony.com/2.3/Symfony/Component/HttpKernel/KernelEvents.html

```
13                 return;
14             }
15
16             // ...
17         }
18     }
```

Two types of request are available in the *HttpKernelInterface*[4] interface: `HttpKernelInterface::MASTER_REQUEST` and `HttpKernelInterface::SUB_REQUEST`.

4. http://api.symfony.com/2.3/Symfony/Component/HttpKernel/HttpKernelInterface.html

Chapter 47

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 *`ContainerAwareHttpKernel`*[1] 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` service, you will receive a *`ScopeWideningInjectionException`*[2] when compiling the container. Read the sidebar below for more details.

1. http://api.symfony.com/2.3/Symfony/Component/HttpKernel/DependencyInjection/ContainerAwareHttpKernel.html
2. http://api.symfony.com/2.3/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, every time 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.

Using a Service from a narrower Scope

If your service has a dependency on a scoped service (like the `request`), you have three ways to deal with it:

- Use setter injection if the dependency is "synchronized"; this is the recommended way and the best solution for the `request` instance as it is synchronized with the `request` scope (see *Using a synchronized Service*).
- Put your service in the same scope as the dependency (or a narrower one). If you depend on the `request` service, this means putting your new service in the `request` scope (see *Changing the Scope of your Service*);
- Pass the entire container to your service and retrieve your dependency from the container each time you need it to be sure you have the right instance -- your service can live in the default `container` scope (see *Passing the Container as a Dependency of your Service*);

Each scenario is detailed in the following sections.

Using a synchronized Service

New in version 2.3: Synchronized services are new in Symfony 2.3.

Injecting the container or setting your service to a narrower scope have drawbacks. For synchronized services (like the `request`), using setter injection is the best option as it has no drawbacks and everything works without any special code in your service or in your definition:

Listing 47-1
```php
// src/Acme/HelloBundle/Mail/Mailer.php
namespace Acme\HelloBundle\Mail;

use Symfony\Component\HttpFoundation\Request;

class Mailer
{
    protected $request;

    public function setRequest(Request $request = null)
    {
        $this->request = $request;
    }

    public function sendEmail()
    {
        if (null === $this->request) {
            // throw an error?
        }

        // ... do something using the request here
    }
}
```

Whenever the `request` scope is entered or left, the service container will automatically call the `setRequest()` method with the current `request` instance.

You might have noticed that the `setRequest()` method accepts `null` as a valid value for the `request` argument. That's because when leaving the `request` scope, the `request` instance can be `null` (for the master request for instance). Of course, you should take care of this possibility in your code. This should also be taken into account when declaring your service:

Listing 47-2
```yaml
# src/Acme/HelloBundle/Resources/config/services.yml
services:
    greeting_card_manager:
        class: Acme\HelloBundle\Mail\GreetingCardManager
        calls:
            - [setRequest, ['@?request=']]
```

You can declare your own `synchronized` services very easily; here is the declaration of the `request` service for reference:

Listing 47-3
```yaml
services:
    request:
        scope: request
        synthetic: true
        synchronized: true
```

Changing the Scope of your Service

Changing the scope of a service should be done in its definition:

Listing 47-4
```yaml
# src/Acme/HelloBundle/Resources/config/services.yml
services:
    greeting_card_manager:
        class: Acme\HelloBundle\Mail\GreetingCardManager
        scope: request
        arguments: [@request]
```

Passing the Container as a Dependency of your Service

Setting the scope to a narrower one 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 can pass the entire container into your service:

Listing 47-5
```php
// src/Acme/HelloBundle/Mail/Mailer.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 cause 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 47-6

```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).

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 48

How to work with Compiler Passes in Bundles

Compiler passes give you an opportunity to manipulate other service definitions that have been registered with the service container. You can read about how to create them in the components section "*Compiling the Container*". To register a compiler pass from a bundle you need to add it to the build method of the bundle definition class:

Listing 48-1

```php
// src/Acme/MailerBundle/AcmeMailerBundle.php
namespace Acme\MailerBundle;

use Symfony\Component\HttpKernel\Bundle\Bundle;
use Symfony\Component\DependencyInjection\ContainerBuilder;

use Acme\MailerBundle\DependencyInjection\Compiler\CustomCompilerPass;

class AcmeMailerBundle extends Bundle
{
    public function build(ContainerBuilder $container)
    {
        parent::build($container);

        $container->addCompilerPass(new CustomCompilerPass());
    }
}
```

One of the most common use-cases of compiler passes is to work with tagged services (read more about tags in the components section "*Working with Tagged Services*"). If you are using custom tags in a bundle then by convention, tag names consist of the name of the bundle (lowercase, underscores as separators), followed by a dot, and finally the "real" name. For example, if you want to introduce some sort of "transport" tag in your AcmeMailerBundle, you should call it `acme_mailer.transport`.

Chapter 49
Session Proxy Examples

The session proxy mechanism has a variety of uses and this example demonstrates two common uses. Rather than injecting the session handler as normal, a handler is injected into the proxy and registered with the session storage driver:

Listing 49-1
```php
use Symfony\Component\HttpFoundation\Session\Session;
use Symfony\Component\HttpFoundation\Session\Storage\NativeSessionStorage;
use Symfony\Component\HttpFoundation\Session\Storage\Handler\PdoSessionStorage;

$proxy = new YourProxy(new PdoSessionStorage());
$session = new Session(new NativeSessionStorage($proxy));
```

Below, you'll learn two real examples that can be used for `YourProxy`: encryption of session data and readonly guest session.

Encryption of Session Data

If you wanted to encrypt the session data, you could use the proxy to encrypt and decrypt the session as required:

Listing 49-2
```php
use Symfony\Component\HttpFoundation\Session\Storage\Proxy\SessionHandlerProxy;

class EncryptedSessionProxy extends SessionHandlerProxy
{
    private $key;

    public function __construct(\SessionHandlerInterface $handler, $key)
    {
        $this->key = $key;

        parent::__construct($handler);
    }

    public function read($id)
```

```
    {
        $data = parent::write($id, $data);

        return mcrypt_decrypt(\MCRYPT_3DES, $this->key, $data);
    }

    public function write($id, $data)
    {
        $data = mcrypt_encrypt(\MCRYPT_3DES, $this->key, $data);

        return parent::write($id, $data);
    }
}
```

Readonly Guest Sessions

There are some applications where a session is required for guest users, but there is no particular need to persist the session. In this case you can intercept the session before it writes:

Listing 49-3

```
use Foo\User;
use Symfony\Component\HttpFoundation\Session\Storage\Proxy\SessionHandlerProxy;

class ReadOnlyGuestSessionProxy extends SessionHandlerProxy
{
    private $user;

    public function __construct(\SessionHandlerInterface $handler, User $user)
    {
        $this->user = $user;

        parent::__construct($handler);
    }

    public function write($id, $data)
    {
        if ($this->user->isGuest()) {
            return;
        }

        return parent::write($id, $data);
    }
}
```

Chapter 50

Making the Locale "Sticky" during a User's Session

Prior to Symfony 2.1, the locale was stored in a session called `_locale`. Since 2.1, it is stored in the Request, which means that it's not "sticky" during a user's request. In this article, you'll learn how to make the locale of a user "sticky" so that once it's set, that same locale will be used for every subsequent request.

Creating LocaleListener

To simulate that the locale is stored in a session, you need to create and register a *new event listener*. The listener will look something like this. Typically, `_locale` is used as a routing parameter to signify the locale, though it doesn't really matter how you determine the desired locale from the request:

Listing 50-1

```php
// src/Acme/LocaleBundle/EventListener/LocaleListener.php
namespace Acme\LocaleBundle\EventListener;

use Symfony\Component\HttpKernel\Event\GetResponseEvent;
use Symfony\Component\HttpKernel\KernelEvents;
use Symfony\Component\EventDispatcher\EventSubscriberInterface;

class LocaleListener implements EventSubscriberInterface
{
    private $defaultLocale;

    public function __construct($defaultLocale = 'en')
    {
        $this->defaultLocale = $defaultLocale;
    }

    public function onKernelRequest(GetResponseEvent $event)
    {
        $request = $event->getRequest();
```

```
        if (!$request->hasPreviousSession()) {
            return;
        }

        // try to see if the locale has been set as a _locale routing parameter
        if ($locale = $request->attributes->get('_locale')) {
            $request->getSession()->set('_locale', $locale);
        } else {
            // if no explicit locale has been set on this request, use one from the session
            $request->setLocale($request->getSession()->get('_locale',
$this->defaultLocale));
        }
    }

    public static function getSubscribedEvents()
    {
        return array(
            // must be registered before the default Locale listener
            KernelEvents::REQUEST => array(array('onKernelRequest', 17)),
        );
    }
}
```

Then register the listener:

Listing 50-2
```
services:
    acme_locale.locale_listener:
        class: Acme\LocaleBundle\EventListener\LocaleListener
        arguments: ["%kernel.default_locale%"]
        tags:
            - { name: kernel.event_subscriber }
```

That's it! Now celebrate by changing the user's locale and seeing that it's sticky throughout the request. Remember, to get the user's locale, always use the *`Request::getLocale`*[1] method:

Listing 50-3
```
// from a controller...
$locale = $this->getRequest()->getLocale();
```

1. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/Request.html#getLocale()

Chapter 51

Configuring the Directory where Sessions Files are Saved

By default, Symfony stores the session data in the cache directory. This means that when you clear the cache, any current sessions will also be deleted.

Using a different directory to save session data is one method to ensure that your current sessions aren't lost when you clear Symfony's cache.

Using a different session save handler is an excellent (yet more complex) method of session management available within Symfony. See *Configuring Sessions and Save Handlers* for a discussion of session save handlers. There is also an entry in the cookbook about storing sessions in the *database*.

To change the directory in which Symfony saves session data, you only need change the framework configuration. In this example, you will change the session directory to `app/sessions`:

Listing 51-1
```yaml
# app/config/config.yml
framework:
    session:
        save_path: "%kernel.root_dir%/sessions"
```

Chapter 52
Bridge a legacy application with Symfony Sessions

New in version 2.3: The ability to integrate with a legacy PHP session was added in Symfony 2.3.

If you're integrating the Symfony full-stack Framework into a legacy application that starts the session with `session_start()`, you may still be able to use Symfony's session management by using the PHP Bridge session.

If the application has sets it's own PHP save handler, you can specify null for the `handler_id`:

Listing 52-1
```
framework:
    session:
        storage_id: session.storage.php_bridge
        handler_id: ~
```

Otherwise, if the problem is simply that you cannot avoid the application starting the session with `session_start()`, you can still make use of a Symfony based session save handler by specifying the save handler as in the example below:

Listing 52-2
```
framework:
    session:
        storage_id: session.storage.php_bridge
        handler_id: session.handler.native_file
```

If the legacy application requires its own session save-handler, do not override this. Instead set `handler_id: ~`. Note that a save handler cannot be changed once the session has been started. If the application starts the session before Symfony is initialized, the save-handler will have already been set. In this case, you will need `handler_id: ~`. Only override the save-handler if you are sure the legacy application can use the Symfony save-handler without side effects and that the session has not been started before Symfony is initialized.

For more details, see *Integrating with Legacy Sessions*.

Chapter 53

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 53-1

```
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/2.3/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 53-2
```php
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 53-3
```yaml
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 53-4
```twig
{% extends 'WebProfilerBundle:Profiler:layout.html.twig' %}

{% block toolbar %}
    {# the web debug toolbar content #}
{% endblock %}
```

2. http://api.symfony.com/2.3/Symfony/Component/HttpKernel/DataCollector/DataCollector.html

```
6
7  {% block head %}
8      {# if the web profiler panel needs some specific JS or CSS files #}
9  {% endblock %}
10
11 {% block menu %}
12     {# the menu content #}
13 {% endblock %}
14
15 {% block panel %}
16     {# the panel content #}
17 {% 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="..."`). 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 53-5
```
services:
    data_collector.your_collector_name:
        class: Acme\DebugBundle\Collector\Class\Name
        tags:
            - { name: data_collector, template: "AcmeDebugBundle:Collector:templatename",
id: "your_collector_name" }
```

Chapter 54

How to use Matchers to enable the Profiler Conditionally

By default, the profiler is only activated in the development environment. But it's imaginable that a developer may want to see the profiler even in production. Another situation may be that you want to show the profiler only when an admin has logged in. You can enable the profiler in these situations by using matchers.

Using the built-in Matcher

Symfony2 provides a *built-in matcher*[1] which can match paths and IPs. For example, if you want to only show the profiler when accessing the page with the `168.0.0.1` ip, then you can use this configuration:

Listing 54-1

```yaml
# app/config/config.yml
framework:
    # ...
    profiler:
        matcher:
            ip: 168.0.0.1
```

You can also set a `path` option to define the path on which the profiler should be enabled. For instance, setting it to `^/admin/` will enable the profiler only for the `/admin/` urls.

Creating a Custom Matcher

You can also create a custom matcher. This is a service that checks whether the profiler should be enabled or not. To create that service, create a class which implements *`RequestMatcherInterface`*[2]. This

1. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/RequestMatcher.html
2. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/RequestMatcherInterface.html

interface requires one method: *matches()*[3]. This method returns false to disable the profiler and true to enable the profiler.

To enable the profiler when a `ROLE_SUPER_ADMIN` is logged in, you can use something like:

Listing 54-2
```php
// src/Acme/DemoBundle/Profiler/SuperAdminMatcher.php
namespace Acme\DemoBundle\Profiler;

use Symfony\Component\Security\Core\SecurityContext;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\RequestMatcherInterface;

class SuperAdminMatcher implements RequestMatcherInterface
{
    protected $securityContext;

    public function __construct(SecurityContext $securityContext)
    {
        $this->securityContext = $securityContext;
    }

    public function matches(Request $request)
    {
        return $this->securityContext->isGranted('ROLE_SUPER_ADMIN');
    }
}
```

Then, you need to configure the service:

Listing 54-3
```yaml
parameters:
    acme_demo.profiler.matcher.super_admin.class: Acme\DemoBundle\Profiler\SuperAdminMatcher

services:
    acme_demo.profiler.matcher.super_admin:
        class: "%acme_demo.profiler.matcher.super_admin.class%"
        arguments: [@security.context]
```

Now the service is registered, the only thing left to do is configure the profiler to use this service as the matcher:

Listing 54-4
```yaml
# app/config/config.yml
framework:
    # ...
    profiler:
        matcher:
            service: acme_demo.profiler.matcher.super_admin
```

3. http://api.symfony.com/2.3/Symfony/Component/HttpFoundation/RequestMatcherInterface.html#matches()

Chapter 55

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 the PHP SOAP extension is used in these 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 55-1

```php
// src/Acme/SoapBundle/Services/HelloService.php
namespace Acme\SoapBundle\Services;

class HelloService
{
    private $mailer;

    public function __construct(\Swift_Mailer $mailer)
    {
        $this->mailer = $mailer;
    }

    public function hello($name)
    {
```

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

```
15
16          $message = \Swift_Message::newInstance()
17                                  ->setTo('me@example.com')
18                                  ->setSubject('Hello Service')
19                                  ->setBody($name . ' says hi!');
20
21          $this->mailer->send($message);
22
23          return 'Hello, '.$name;
24      }
25  }
```

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, you can configure Symfony to construct a `HelloService` object properly:

Listing 55-2

```
# app/config/config.yml
services:
    hello_service:
        class: Acme\SoapBundle\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 55-3

```
1  namespace Acme\SoapBundle\Controller;
2
3  use Symfony\Bundle\FrameworkBundle\Controller\Controller;
4  use Symfony\Component\HttpFoundation\Response;
5
6  class HelloServiceController extends Controller
7  {
8      public function indexAction()
9      {
10         $server = new \SoapServer('/path/to/hello.wsdl');
11         $server->setObject($this->get('hello_service'));
12
13         $response = new Response();
14         $response->headers->set('Content-Type', 'text/xml; charset=ISO-8859-1');
15
16         ob_start();
17         $server->handle();
18         $response->setContent(ob_get_clean());
19
20         return $response;
21     }
22 }
```

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`.

4. `http://php.net/manual/en/book.outcontrol.php`

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 55-4
```php
$client = new \Soapclient('http://example.com/app.php/soap?wsdl', true);

$result = $client->call('hello', array('name' => 'Scott'));
```

An example WSDL is below.

Listing 55-5
```xml
<?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>
```

5. http://sourceforge.net/projects/nusoap

```
        </operation>
    </binding>

    <service name="hellowsdl">
        <port name="hellowsdlPort" binding="tns:hellowsdlBinding">
            <soap:address location="http://example.com/app.php/soap" />
        </port>
    </service>
</definitions>
```

Chapter 56
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.

This chapter walks 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 you travel from "then" to "now".

Directory Structure

When looking at a Symfony2 project - for example, the *Symfony2 Standard Edition*[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 will live somewhere in the `vendor/` 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 the 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 live somewhere in the `vendor/`.

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 56-1
```
$ 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 tool named *Composer*[2] 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 56-2
```php
namespace Sensio\Bundle\FrameworkExtraBundle;

use Symfony\Component\HttpKernel\Bundle\Bundle;
// ...

class SensioFrameworkExtraBundle extends Bundle
{
    // ...
}
```

The file itself lives at `vendor/sensio/framework-extra-bundle/Sensio/Bundle/FrameworkExtraBundle/SensioFrameworkExtraBundle.php`. As you can see, the second part of the path follows the namespace of the class. The first part is equal to the package name of the SensioFrameworkExtraBundle.

The namespace, `Sensio\Bundle\FrameworkExtraBundle`, and package name, `sensio/framework-extra-bundle`, spells out the directory that the file should live in (`vendor/sensio/framework-extra-bundle/Sensio/Bundle/FrameworkExtraBundle/`). Composer can then look for the file at this specific place and load it very fast.

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" error means that the namespace of the class 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/sensio/framework-extra-bundle` directory and that, for example, the `Doctrine` namespace lives in the `vendor/doctrine/orm/lib/` directory. This mapping is entirely controlled by Composer. Each third-party library you load through Composer has its settings defined and Composer takes care of everything for you.

For this to work, all third-party libraries used by your project must be defined in the `composer.json` 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 `AcmeDemoBundle` is not defined in your `composer.json` file. Nonetheless are the files autoloaded. This is because you can tell composer to autoload files from specific directories without defining a dependency:

Listing 56-3
```json
"autoload": {
    "psr-0": { "": "src/" }
}
```

This means that if a class is not found in the `vendor` directory, Composer will search in the `src` directory before throwing a "class does not exist" exception. Read more about configuring the Composer Autoloader in *the Composer documentation*[3]

2. http://getcomposer.org
3. http://getcomposer.org/doc/04-schema.md#autoload

Using the Console

In symfony1, the console is in the root directory of your project and is called `symfony`:

Listing 56-4
```
$ php symfony
```

In Symfony2, the console is now in the app sub-directory and is called `console`:

Listing 56-5
```
$ 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 56-6
```
// config/ProjectConfiguration.class.php
public function setup()
{
    // some plugins here
    $this->enableAllPluginsExcept(array(...));
}
```

In Symfony2, the bundles are activated inside the application kernel:

Listing 56-7

```php
// 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 56-8
```yaml
# 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 56-9
```yaml
# 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 56-10
```yaml
# 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 56-11
```yaml
parameters:
    email.from_address: foo.bar@example.com
```

You can now access this from a controller, for example:

Listing 56-12
```php
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*".

Chapter 57

How to deploy a Symfony2 application

Deploying can be a complex and varied task depending on your setup and needs. This entry doesn't try to explain everything, but rather offers the most common requirements and ideas for deployment.

Symfony2 Deployment Basics

The typical steps taken while deploying a Symfony2 application include:

1. Upload your modified code to the live server;
2. Update your vendor dependencies (typically done via Composer, and may be done before uploading);
3. Running database migrations or similar tasks to update any changed data structures;
4. Clearing (and perhaps more importantly, warming up) your cache.

A deployment may also include other things, such as:

- Tagging a particular version of your code as a release in your source control repository;
- Creating a temporary staging area to build your updated setup "offline";
- Running any tests available to ensure code and/or server stability;
- Removal of any unnecessary files from `web` to keep your production environment clean;
- Clearing of external cache systems (like *Memcached*[1] or *Redis*[2]).

How to deploy a Symfony2 application

There are several ways you can deploy a Symfony2 application.

Let's start with a few basic deployment strategies and build up from there.

1. `http://memcached.org/`
2. `http://redis.io/`

Basic File Transfer

The most basic way of deploying an application is copying the files manually via ftp/scp (or similar method). This has its disadvantages as you lack control over the system as the upgrade progresses. This method also requires you to take some manual steps after transferring the files (see Common Post-Deployment Tasks)

Using Source Control

If you're using source control (e.g. git or svn), you can simplify by having your live installation also be a copy of your repository. When you're ready to upgrade it is as simple as fetching the latest updates from your source control system.

This makes updating your files *easier*, but you still need to worry about manually taking other steps (see Common Post-Deployment Tasks).

Using Build scripts and other Tools

There are also high-quality tools to help ease the pain of deployment. There are even a few tools which have been specifically tailored to the requirements of Symfony2, and which take special care to ensure that everything before, during, and after a deployment has gone correctly.

See The Tools for a list of tools that can help with deployment.

Common Post-Deployment Tasks

After deploying your actual source code, there are a number of common things you'll need to do:

A) Configure your `app/config/parameters.yml` file

This file should be customized on each system. The method you use to deploy your source code should *not* deploy this file. Instead, you should set it up manually (or via some build process) on your server(s).

B) Update your vendors

Your vendors can be updated before transferring your source code (i.e. update the `vendor/` directory, then transfer that with your source code) or afterwards on the server. Either way, just update your vendors as your normally do:

Listing 57-1
```
$ php composer.phar install --optimize-autoloader
```

The `--optimize-autoloader` flag makes Composer's autoloader more performant by building a "class map".

C) Clear your Symfony cache

Make sure you clear (and warm-up) your Symfony cache:

Listing 57-2
```
$ php app/console cache:clear --env=prod --no-debug
```

D) Dump your Assetic assets

If you're using Assetic, you'll also want to dump your assets:

Listing 57-3
```
$ php app/console assetic:dump --env=prod --no-debug
```

E) Other things!

There may be lots of other things that you need to do, depending on your setup:

- Running any database migrations
- Clearing your APC cache
- Running `assets:install` (taken care of already in `composer.phar install`)
- Add/edit CRON jobs
- Pushing assets to a CDN
- ...

Application Lifecycle: Continuous Integration, QA, etc

While this entry covers the technical details of deploying, the full lifecycle of taking code from development up to production may have a lot more steps (think deploying to staging, QA, running tests, etc).

The use of staging, testing, QA, continuous integration, database migrations and the capability to roll back in case of failure are all strongly advised. There are simple and more complex tools and one can make the deployment as easy (or sophisticated) as your environment requires.

Don't forget that deploying your application also involves updating any dependency (typically via Composer), migrating your database, clearing your cache and other potential things like pushing assets to a CDN (see Common Post-Deployment Tasks).

The Tools

Capifony[3]:

> This tool provides a specialized set of tools on top of Capistrano, tailored specifically to symfony and Symfony2 projects.

sf2debpkg[4]:

> This tool helps you build a native Debian package for your Symfony2 project.

Magallanes[5]:

> This Capistrano-like deployment tool is built in PHP, and may be easier for PHP developers to extend for their needs.

Bundles:

> There are many *bundles that add deployment features*[6] directly into your Symfony2 console.

3. http://capifony.org/
4. https://github.com/liip/sf2debpkg
5. https://github.com/andres-montanez/Magallanes

Basic scripting:

> You can of course use shell, *Ant*[7], or any other build tool to script the deploying of your project.

Platform as a Service Providers:

> PaaS is a relatively new way to deploy your application. Typically a PaaS will use a single configuration file in your project's root directory to determine how to build an environment on the fly that supports your software. One provider with confirmed Symfony2 support is *PagodaBox*[8].

Looking for more? Talk to the community on the *Symfony IRC channel*[9] #symfony (on freenode) for more information.

6. http://knpbundles.com/search?q=deploy
7. http://blog.sznapka.pl/deploying-symfony2-applications-with-ant
8. https://github.com/jmather/pagoda-symfony-sonata-distribution/blob/master/Boxfile
9. http://webchat.freenode.net/?channels=symfony