



Symfony

The Symfony CMF Book

Version: 1.0

generated on July 13, 2016

The Symfony CMF Book (1.0)

This work is licensed under the “Attribution-Share Alike 3.0 Unported” license (<http://creativecommons.org/licenses/by-sa/3.0/>).

You are free **to share** (to copy, distribute and transmit the work), and **to remix** (to adapt the work) under the following conditions:

- **Attribution:** You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).
- **Share Alike:** If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license. For any reuse or distribution, you must make clear to others the license terms of this work.

The information in this book is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor SensioLabs shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

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

- The Big Picture 4
- The Model 10
- The Router 14
- The Third Party Bundles 17
- Installing the Standard Edition 20
- First look at the internals 24
- Routing 27
- The Database Layer: PHPCR-ODM 33
- Static Content 38
- Structuring Content 40



Chapter 1

The Big Picture

Start using the Symfony CMF in 10 minutes! This chapter will walk you through the base concepts of the Symfony CMF and get you started with it.

It's important to know that the Symfony CMF is a collection of bundles which provide common functionality needed when building a CMS with the Symfony Framework. Before you read further, you should at least have a basic knowledge of the Symfony Framework. If you don't know Symfony, start by reading the *Symfony Framework Quick Tour*¹.

Solving the framework versus CMS dilemma

Before starting a new project, there is a difficult decision on whether it will be based on a framework or on a CMS. When choosing to use a framework, you need to spend much time creating CMS features for the project. On the other hand, when choosing to use a CMS, it's more difficult to build custom application functionality. It is impossible or at least very hard to customize the core parts of the CMS.

The CMF is created to solve this framework versus CMS dilemma. It provides bundles, so you can easily add CMS features to your project. But it also provides flexibility and in all cases you are using the framework, so you can build custom functionality the way you want. This is called a *decoupled CMS*².

The bundles provided by the Symfony CMF can work together, but they are also able to work standalone. This means that you don't need to add all bundles, you can decide to only use one of them (e.g. only the RoutingBundle or the MediaBundle).

Downloading the Symfony CMF Standard Edition

When you want to start using the CMF for a new project, you can download the Symfony CMF Standard Edition. The Symfony CMF Standard Edition is similar to the *Symfony Standard Edition*³, but contains and configures essential Symfony CMF bundles. It also adds a very simple bundle to show some of the basic Symfony CMF features.

1. http://symfony.com/doc/current/quick_tour/the_big_picture.html

2. <http://decoupledcms.org>

3. <https://github.com/symfony/symfony-standard>

The best way to download the Symfony CMF Standard Edition is using *Composer*⁴:

Listing 1-1 1 \$ php composer.phar create-project symfony-cmf/standard-edition <path-to-install>

Setting up the Database

Now, the only thing left to do is setting up the database. This is not something you are used to doing when creating Symfony applications, but to make a lot of things configurable using an admin, the Symfony CMF needs a database.

To get quickly started, it is expected that you have enabled the sqlite extension. After that, run these commands:

Listing 1-2

```
1 $ php app/console doctrine:database:create
2 $ php app/console doctrine:phpcr:init:dbal
3 $ php app/console doctrine:phpcr:repository:init
4 $ php app/console doctrine:phpcr:fixtures:load
```



You are going to learn more about the Database layer of the Symfony CMF later in the Quick Tour.

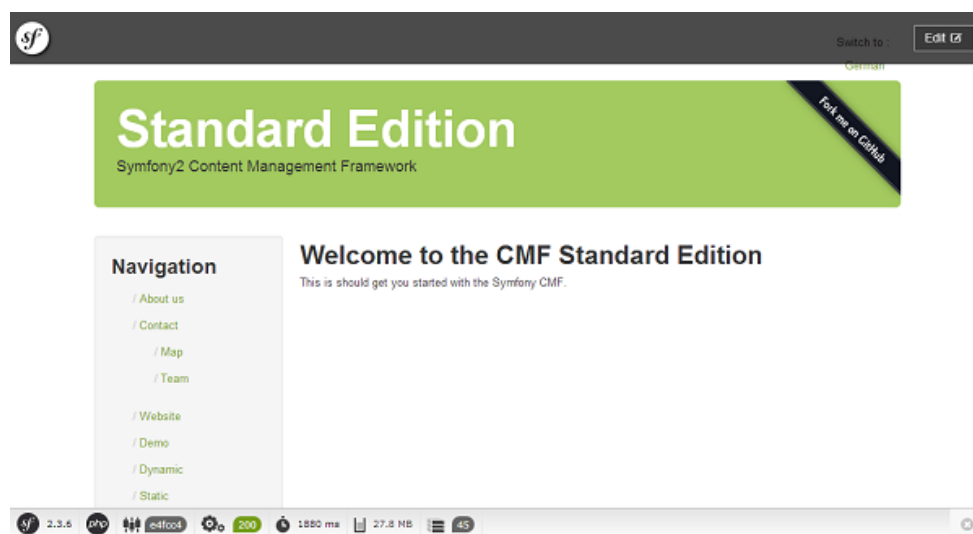
For a complete installation guide, see the "Installing the Standard Edition" chapter of the Book.

The Request Flow



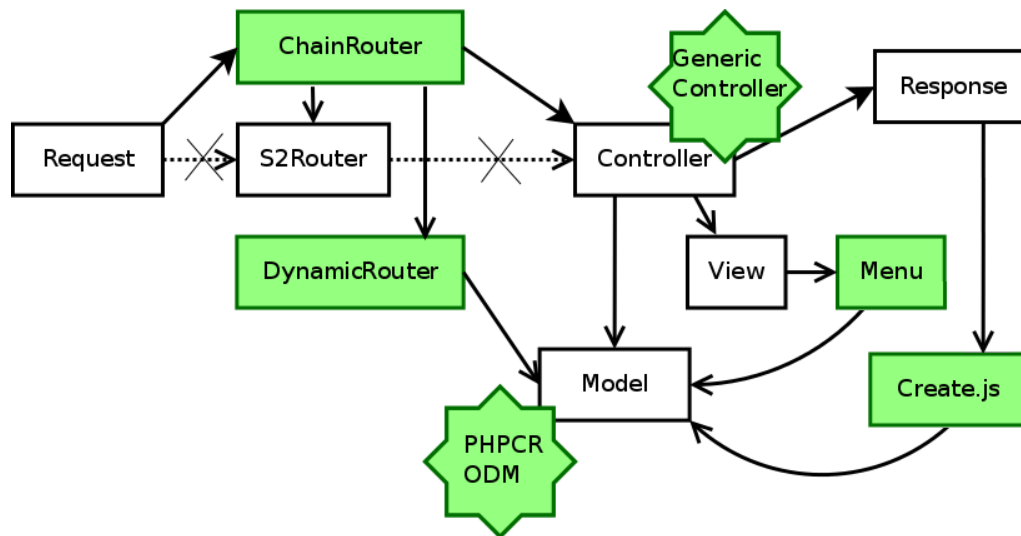
When you have at least PHP 5.4, use the `server:run` command to run a local server for the demo. Otherwise, use a localhost and prefix the URLs with `/path-to-project/web/app_dev.php/`.

Now the standard edition is ready to use, navigate to the homepage (<http://localhost:8000/>) to see the demo:



4. <http://getcomposer.org/>

You see that we already have a complete website in our demo. Let's take a closer look at the request flow for a Symfony CMF application:



First of all, you see a typical symfony request flow following the white blocks. It creates a `Request` object which will be passed to a router, which executes the controller and that controller uses models to generate a view to put in the response.

On top of this, the CMF adds the green blocks. In the coming sections, you'll learn more about these separately.

The Model

Before creating the CMF, the team had done a lot of research on which database to use. They ended up finding *JCR*⁵, a Content Repository for Java. Together with some other developers they created *PHPCR*⁶, a PHP port of the JCR specification.

PHPCR uses a directory-like structure. It stores elements in a big tree. Elements have a parent and can have children.



Although PHPCR is the first choice of the CMF team, the bundles are not tied to a specific storage system. Some bundles also provide ORM integration and you can also add your own models easily.

The Router

In Symfony, the routes are stored in a configuration file. This means only a developer can change routes. In a CMS, you want the admin to change the routes of their site. This is why the Symfony CMF introduces a `DynamicRouter`.

The `DynamicRouter` loads some routes which possibly match the request from the database and then tries to find an exact match. The routes in the database can be edited, deleted and created using an admin interface, so everything is fully under the control of the admin.

Because you may also want other Routers, like the normal Symfony router, the CMF also provides a `ChainRouter`. A chain router contains a chain of other routers and executes them in a given order to find a match.

5. http://en.wikipedia.org/wiki/Content_repository_API_for_Java

6. <http://phpcr.github.io/>

Using a database to store the routes makes it also possible to reference other documents from the route. This means that a route can have a Content object.



You'll learn more about the router in the next chapter of the Quick Tour.

The Controller

When a Route matches, a Controller is executed. This Controller normally just gets the Content object from the Route and renders it. Because it is almost always the same, the CMF uses a generic Controller which it will execute. This can be overridden by setting a specific controller for a Route or Content object.

The View

Using the RoutingBundle, you can configure which Content objects are rendered by a specific template or controller. The generic controller will then render this template.

A view also uses a Menu, provided by the *KnpmenuBundle*⁷, and it can have integration with Create.js, for live editing.

Adding a New Page

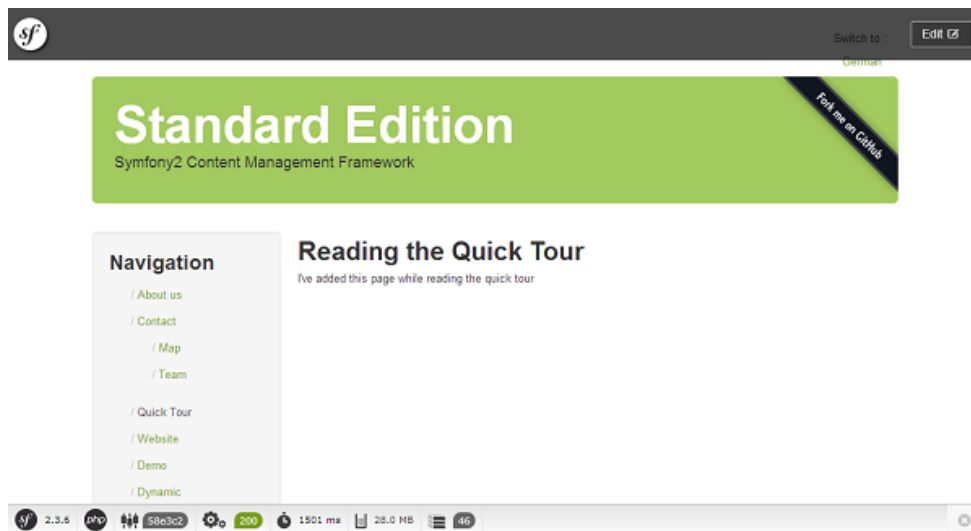
Now you know the request flow, you can start adding a new page. In the Symfony CMF Standard Edition, the data is stored in data files, which are loaded when executing the `doctrine:phpcr:fixtures:load` command. To add a new page, you just need to edit such a data file, which is located in the `Resources/data` directory:

Listing 1-3

```
1 # src/Acme/MainBundle/Resources/data/page.yml
2 static:
3   # ...
4
5   quick_tour:
6     name: "quick_tour"
7     label: "Quick Tour"
8     title: "Reading the Quick Tour"
9     body: "I've added this page while reading the quick tour"
```

After this, you need to run the `doctrine:phpcr:fixtures:load` to reflect the changes on the database and after refreshing, you can see your new page!

7. <http://knpbundles.com/KnpLabs/KnpMenuBundle>



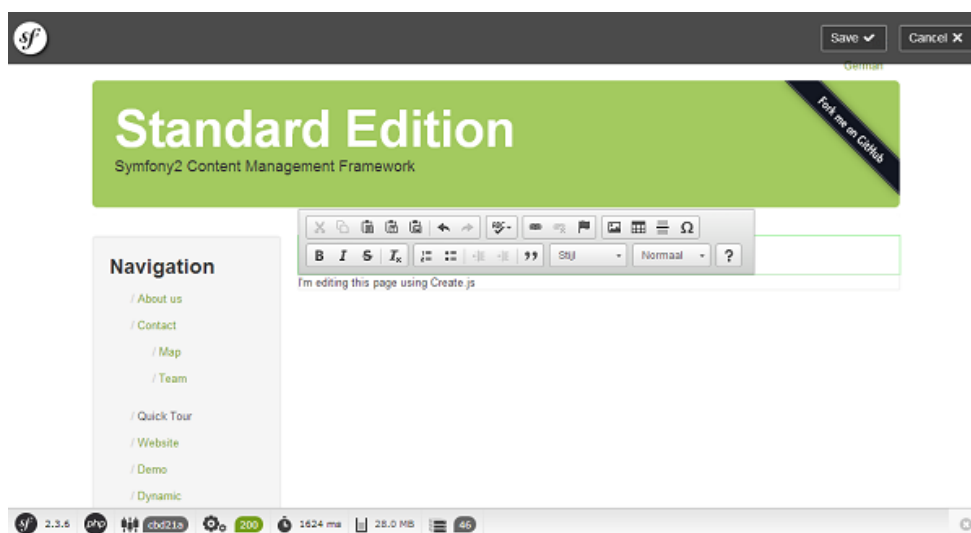
Live Editing

You may already noticed the big black bar at the top of the site:



This bar is generated by the *Create.js*⁸ library. The Symfony CMF integrates the *CreatePHP*⁹ and *Create.js*¹⁰ libraries using a *CreateBundle*. This enables you to edit a page using a full WYSIWYG editor when you are reading the page.

Now you can change the content of our new page using *Create.js*:



8. <http://createjs.org/>

9. <http://demo.createphp.org/>

10. <http://createjs.org/>

After clicking "save", the changes are saved using the CreateBundle and the content is updated.

Final Thoughts

Congratulations! You've come to the end of your first introduction into the Symfony CMF. There is a lot more to discover, but you should already see how the Symfony CMF tries to make your life as a developer better by providing some CMS bundles. If you want to discover more, you can dive into the next section: "*The Model*".



Chapter 2

The Model

You decided to continue reading 10 more minutes about the Symfony CMF? That's great news! In this part, you will learn more about the default database layer of the CMF.



Again, this chapter is talking about the PHPCR storage layer. But the CMF is storage agnostically created, meaning it is not tied to specific storage system.

Getting Familiar with PHPCR

*PHPCR*¹ stores all data into one big tree structure. You can compare this to a filesystem where each file and directory contains data. This means that all data stored with PHPCR has a relationship with at least one other data: its parent. The inverse relation also exists, you can also get the children of a data element. Let's take a look at the dump of the tree of the Standard Edition you downloaded in the previous chapter. Go to your directory and execute the following command:

Listing 2-1 1 `$ php app/console doctrine:phpcr:node:dump`

The result will be the PHPCR tree:

Listing 2-2

```
1  ROOT:
2    cms:
3      simple:
4        about:
5        contact:
6          map:
7          team:
8        quick_tour:
9        dynamic:
10       docs:
11       demo:
12       demo_redirect:
13       hardcoded_dynamic:
14       hardcoded_static:
```

1. <http://phpcr.github.io/>

Each data is called a *node* in PHPCR. In this tree, there are 13 nodes and one ROOT node (created by PHPCR). You may have already seen the document you created in the previous section, it's called `quick_tour` (and its path is `/cms/simple/quick_tour`). When using the SimpleCmsBundle, all nodes are stored in the `/cms/simple` path.

Each node has properties, which contain the data. The content, title and label you set for your page are saved in such properties for the `quick_tour` node. You can view these properties by adding the `--props` switch to the dump command.



Previously, the PHPCR tree was compared with a `FileSystem`. While this gives you a good image of what happens, it's not the truth. You can better compare it to an XML file, where each node is an element and its properties are attributes.

Doctrine PHPCR-ODM

The Symfony CMF uses the *Doctrine PHPCR-ODM*² to interact with PHPCR. Doctrine allows a user to create objects (called *documents*) which are directly persisted into and retrieved from the PHPCR tree. This is similar to the Doctrine ORM used by the Symfony2 Framework, but then for PHPCR.

Creating a Page with code

Now you know a little bit more about PHPCR and you know the tool to interact with it, you can start using it yourself. In the previous chapter, you created a page by using a yml file which was parsed by the SimpleCmsBundle. This time, you'll create a page by doing it yourself.

First, you have to create a new DataFixture to add your new page. You do this by creating a new class in the AcmeMainBundle:

```
Listing 2-3 1 // src/Acme/MainBundle/DataFixtures/PHPCR/LoadPageData.php
2 namespace Acme\MainBundle\DataFixtures\PHPCR;
3
4 use Doctrine\Common\Persistence\ObjectManager;
5 use Doctrine\Common\DataFixtures\FixtureInterface;
6 use Doctrine\Common\DataFixtures\OrderedFixtureInterface;
7
8 class LoadPageData implements FixtureInterface, OrderedFixtureInterface
9 {
10     public function getOrder()
11     {
12         // refers to the order in which the class' load function is called
13         // (lower return values are called first)
14         return 10;
15     }
16
17     public function load(ObjectManager $documentManager)
18     {
19     }
20 }
```

The `$documentManager` is the object which will persist the document to PHPCR. But first, you have to create a new Page document:

```
Listing 2-4 1 use Symfony\Cmf\Bundle\SimpleCmsBundle\Doctrine\Phpcr\Page;
2
3 // ...
4 public function load(ObjectManager $documentManager)
5 {
```

2. <http://docs.doctrine-project.org/projects/doctrine-phpcr-odm/en/latest/>

```

6     $page = new Page(); // create a new Page object (document)
7     $page->setName('new_page'); // the name of the node
8     $page->setLabel('Another new Page');
9     $page->setTitle('Another new Page');
10    $page->setBody('I have added this page myself!');
11 }

```

Each document needs a parent. In this case, the parent should just be the root node. To do this, we first retrieve the root document from PHPCR and then set it as its parent:

Listing 2-5

```

1 // ...
2 public function load(ObjectManager $documentManager)
3 {
4     // ...
5
6     // get root document (/cms/simple)
7     $simpleCmsRoot = $documentManager->find(null, '/cms/simple');
8
9     $page->setParent($simpleCmsRoot); // set the parent to the root
10 }

```

And at last, we have to tell the Document Manager to persist our Page document using the Doctrine API:

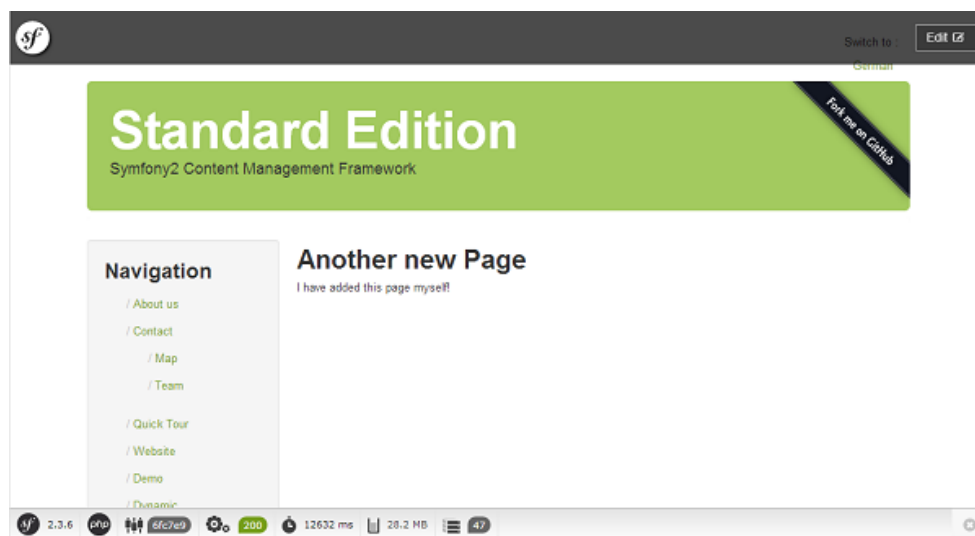
Listing 2-6

```

1 // ...
2 public function load(ObjectManager $documentManager)
3 {
4     // ...
5     $documentManager->persist($page); // add the Page in the queue
6     $documentManager->flush(); // add the Page to PHPCR
7 }

```

Now you need to execute the `doctrine:phpcr:fixtures:load` command again and then you can visit your website again. You'll see your new page you added!



See "The Database Layer: PHPCR-ODM" if you want to know more about using PHPCR in a Symfony application.

Final Thoughts

PHPCR is a powerfull way to store your pages in a CMS. But, if you're not comfortable with it, you can always *switch to another storage layer*.

When looking back at these 20 minutes, you should have learned how to work with a new storage layer and you have added 2 new pages. Do you see how easy the CMF works when making your application editable? It provides most of the things you previously had to do yourself.

But you have now only seen a small bit of the CMF, there is much more to learn about and many other bundles are waiting for you. Before you can do all this, you should meet the backbone of the CMF: The routing system. You can read about that in *the next chapter*. Ready for another 10 minutes?



Chapter 3

The Router

Welcome to the third part of the Quick Tour. You seem to have fallen in love with the CMF, getting this far! And that's a good thing, as you will learn about the backbone of the CMF in this chapter: The Router.

The Backbone of the CMF

As already said, the router is the backbone. To understand this, you have a good view of what a CMS tries to do. In a normal Symfony application, a route refers to a controller which can handle a specific entity. Another route refers to another controller which can handle another entity. This way, a route is tied to a controller. In fact, using the Symfony core you are also limited at this.

But if you look at the base of a CMS, it only needs to handle 1 type of entity: The Content. So most of the routes don't have to be tied to a controller anymore, as only one controller is needed. The Route has to be tied to a specific Content object, which - on its side - can reference a specific template and controller.

Other parts of the CMF are also related to the Router. To give 2 examples: The menu is created by generating specific routes using the Router and the blocks are displayed to specific routes (as they are related to a template).

Loading Routes from the PHPCR tree

In the first chapter, you have already learned that routes are loaded from the database using a special `DynamicRouter`. This way, not all routes need to be loaded each request.

Matching routes from a PHPCR is really simple. If you remember the previous chapter, you know that you can get the `quick_tour` page from PHPCR using `/cms/simple/quick_tour`. The URL to get this page is `quick_tour`. Some other examples:

Listing 3-1

```
1 /cms
2   /simple
3     /about      # /about Route
4     /contact    # /contact Route
5     /team       # /contact/team Route
6     /docs       # /docs Route
```

OK, you got it? The only thing the Router has to do is prefix the route with a specific path prefix and load that document. In the case of the SimpleCmsBundle, all routes are prefixed with `/cms/simple`.

You see that a route like `/contact/team`, which consist of 2 "path units", has 2 documents in the PHPCR tree: `contact` and `team`.

Chaining multiple Routers

You may need to have several prefixes or several routes. For instance, you may want to use both the `DynamicRouter` for the page routes, but also the static routing files from `Symfony` for your custom logic. To be able to do that, the CMF provides a `ChainRouter`. This router chains over multiple router and stops whenever a router matches.

By default, the `ChainRouter` overrides the `Symfony` router and only has the core router in its chain. You can add more routers to the chain in the configuration or by tagging the router services. For instance, the router used by the `SimpleCmsBundle` is a service registered by that bundle and tagged with `cmf_routing.router`.

Creating a new Route

Now you know the basics of routing, you can add a new route to the tree. In the configuration file, configure a new chain router so that you can put your new routes in `/cms/routes`:

```
Listing 3-2 1 # app/config/config.yml
2
3 # ...
4 cmf_routing:
5     chain:
6         routers_by_id:
7             # the standard DynamicRouter
8             cmf_routing.dynamic_router: 200
9
10            # the core symfony router
11            router.default: 100
12     dynamic:
13         persistence:
14             phpcr:
15                 route_basepath: /cms/routes
```

Now you can add a new Route to the tree using `Doctrine`:

```
Listing 3-3 1 // src/Acme/MainBundle/DataFixtures/PHPCR/LoadRoutingData.php
2 namespace Acme\MainBundle\DataFixtures\PHPCR;
3
4 use Doctrine\Common\Persistence\ObjectManager;
5 use Doctrine\Common\DataFixtures\FixtureInterface;
6
7 use Symfony\Cmf\Bundle\RoutingBundle\Doctrine\Phpcr\Route;
8
9 class LoadRoutingData implements FixtureInterface
10 {
11     public function load(ObjectManager $documentManager)
12     {
13         $routesRoot = $documentManager->find(null, '/cms/routes');
14
15         $route = new Route();
16         // set $routesRoot as the parent and 'new-route' as the node name,
17         // this is equal to:
18         // $route->setName('new-route');
19         // $route->setParent($routesRoot);
20         $route->setPosition($routesRoot, 'new-route');
21     }
22 }
```

```
22     $page = $documentManager->find(null, '/cms/routes/quick_tour');
23     $route->setContent($page);
24
25     $documentManager->persist($route); // put $route in the queue
26     $documentManager->flush(); // save it
27 }
28 }
```

This creates a new node called `/cms/routes/new-route` which will display our `quick_tour` page when you go to `/new-route`.



When doing this in a real app, you may want to use a `RedirectRoute` instead.

Final Thoughts

Now you reached the end of this article, you can say you really know the basics of the Symfony CMF. First, you have learned about the Request flow and quickly learned each new step in this process. After that, you have learned more about the default storage layer and the routing system.

The Routing system is created together with some developers from Drupal8. In fact, Drupal 8 uses the Routing component of the Symfony CMF. The Symfony CMF also uses some 3rd party bundles from others and integrated them into PHPCR. In *the next chapter* you'll learn more about those bundles and other projects the Symfony CMF is helping.



Chapter 4

The Third Party Bundles

You're still here? You already learned the basics of the Symfony CMF and you just wanted to learn more and more? Then you can read this chapter! This chapter will walk you quickly through some other CMF bundles. Most of the other bundles are based on the shoulders of some giants, like the *KnplabsMenuBundle*¹ or *SonataBlockBundle*².

The MenuBundle

Let's start with the MenuBundle. If you visit the page, you can see a nice menu in the sidebar. You can find the layout view in the AcmeMainBundle. Somewhere in the middle of the document, you find this code:

```
Listing 4-1 1 <!-- src/Acme/MainBundle/Resources/views/layout.html.twig -->
2
3 <!-- ... -->
4 <div class="row">
5     <div id="content-container" class="span12 boxed">
6         <div class="row">
7             <div id="navigation" class="span3">
8                 <div class="well">
9                     {% block navigation %}
10                    <h2>Navigation</h2>
11                    {{ knp_menu_render('simple') }}
12                    {% endblock %}
13                </div>
14            </div>
15
16            <!-- ... -->
17        </div>
18    </div>
19 </div>
```

As you can see, the menu is rendered by the `knp_menu_render` tag. This seems a bit strange, we are talking about the *CmfMenuBundle* and not the *KnplabsMenuBundle*, aren't we? That's correct, but as a matter of fact the *CmfMenuBundle* is just a tiny layer on top of the *KnplabsMenuBundle*.

1. <https://github.com/Knplabs/KnplabsMenuBundle>

2. <http://sonata-project.org/bundles/block/master/doc/index.html>

Normally, the argument of `knp_menu_render()` is the menu name to render, but when using the `CmfMenuBundle`, it's a node id. In this case, the menu contains all items implementing the `NodeInterface` inside the `/cms/simple` (since the basepath in the Standard Edition is `/cms`).



Apart from including a PHPCR menu provider, the `CmfMenuBundle` also provides Admin classes. See the section about Sonata Admin to learn more about this.

The CreateBundle

You've already seen this bundle in the first chapter. This bundle integrates the `CreatePHP`³ library (which uses the `Create.js`⁴ library) into Symfony2 using the `FOSRestBundle`⁵.

Sonata Admin

We have explained you that the CMF is based on a database, in order to make it editable by an admin without changing the code. But we haven't told you how that admin will be able to maintain the website. Now it's time to reveal how to do that: Using the `SonataAdminBundle`⁶. All the CMF bundles that define editable elements also provide integration to make those elements editable in Sonata Admin.

By default, all Admin classes in the CMF bundles will be activated when the `SonataDoctrinePHPCRAdminBundle`⁷ is installed. You can switch off the Admin class in the configuration. For instance, to disable the `MenuBundle` Admin classes, you would do:

Listing 4-2

```
1 # app/config/config.yml
2 cmf_menu:
3     persistence:
4         phpcr:
5             use_sonata_admin: false
```

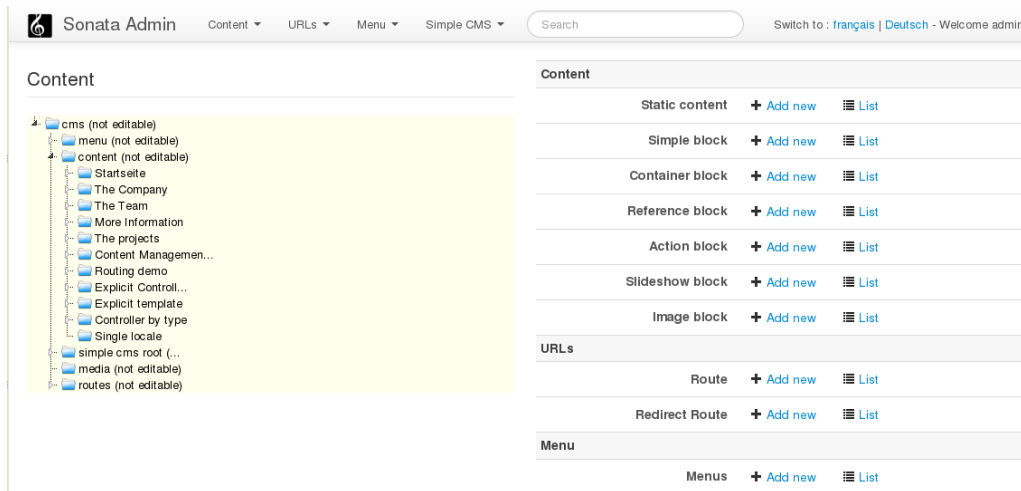
You can also disable/enable all CMF Admin classes by configuring this on the `cmf_core` bundle:

Listing 4-3

```
1 # app/config/config.yml
2 cmf_core:
3     persistence:
4         phpcr:
5             use_sonata_admin: false
```

When the Admin classes are activated, the admin can go to `/admin` (if you installed the `SonataAdminBundle` correctly) and find the well-known admin dashboard with all they need:

3. <http://demo.createphp.org/>
4. <http://createjs.org/>
5. <https://github.com/friendsofsymfony/FOSRestBundle>
6. <http://sonata-project.org/bundles/admin/master/doc/index.html>
7. <http://sonata-project.org/bundles/doctrine-phpcr-admin/master/doc/index.html>



As you can see on the left, the admin uses the *TreeBrowserBundle* to display a live admin tree, where the admin can click on the nodes to edit, remove or move them.

Final Thoughts

You made it! Let's summarize what you've learned in the Quick Tour:

- The Symfony CMF is build for highly customized Content Management Systems;
- The Symfony CMF team creates bundles with a specific CMS feature, which can be used both together and standalone;
- The Symfony CMF uses the database in order to make a lot of things editable by an Admin, however the configuration is kept in the filesystem to keep deployments simple and support version control;
- The PHP Content Repository (PHPCR) is a great database build for CMS systems, but you can use any other storage system for the Symfony CMF too;
- Instead of binding controllers to routes, the routes are bound to content objects.
- The Symfony CMF took care not to reinvent the wheel. That resulted in a lot of bundles integrating commonly known Symfony2 bundles.

I can't tell you more about the architecture and bundles of the Symfony CMF, but there is much more to explore. Take a look at *the book* and get started with your first project using the Symfony CMF!



Chapter 5

Installing the Standard Edition

The Symfony CMF Standard Edition (SE) is a distribution based on all the main components needed for most common use cases.

The goal of this tutorial is to install the CMF bundles, with the minimum necessary configuration and some very simple examples, into a working Symfony2 application.

After that, you get a quick introduction of the bundles you have installed. This can be used to familiarize yourself with the CMF or as a starting point for a new custom application.



You can also install the CMF Sandbox, this is a more complete demo instance of the Symfony CMF. You can view it online at cmf.liip.ch¹. You can also install it locally, so you can play with the code. For installation instructions for the sandbox, please read "*Installing the CMF sandbox*".

Preconditions

As Symfony CMF is based on Symfony2, you should make sure you meet the *Requirements for running Symfony2*². Additionally, you need to have *SQLite*³ PDO extension (`pdo_sqlite`) installed, since it is used as the default storage medium.



By default, Symfony CMF uses Jackalope + Doctrine DBAL and SQLite as the underlying DB. However, Symfony CMF is storage agnostic, which means you can use one of several available data storage mechanisms without having to rewrite your code. For more information on the different available mechanisms and how to install and configure them, refer to *Installing and Configuring Doctrine PHPCR-ODM*

Installation

You can install the Standard Edition in 2 ways:

-
1. <http://cmf.liip.ch>
 2. <http://symfony.com/doc/current/reference/requirements.html>
 3. <http://www.sqlite.org/>

1) Composer

The easiest way to install Symfony CMF is using *Composer*⁴. Get it using

```
Listing 5-1 1 $ curl -sS https://getcomposer.org/installer | php
           2 $ sudo mv composer.phar /usr/local/bin/composer
```

and then get the Symfony CMF code with it (this may take a while):

```
Listing 5-2 1 $ php composer.phar create-project symfony-cmf/standard-edition <path-to-install>
           2 $ cd <path-to-install>
```



The path `<path-to-install>` should either be inside your web server doc root or configure a virtual host for `<path-to-install>`.

This will clone the Standard Edition and install all the dependencies and run some initial commands. These commands require write permissions to the `app/cache` and `app/logs` directory. In case the final commands end up giving permissions errors, please follow the *guidelines in the Symfony Book*⁵ to configure the permissions and then run the `install` command:

```
Listing 5-3 1 $ php composer.phar install
```

2) GIT

You can also install the Standard Edition using GIT. Just clone the repository from github:

```
Listing 5-4 1 $ git clone git://github.com/symfony-cmf/symfony-cmf-standard.git <path-to-install>
           2 $ cd <path-to-install>
```

You still need Composer to get the dependencies. To install or update the dependencies, use the `install` command:

```
Listing 5-5 1 $ php composer.phar install
```

Set up the Database

The next step is to set up the database. If you want to use SQLite as your database backend just go ahead and run the following:

```
Listing 5-6 1 $ php app/console doctrine:database:create
           2 $ php app/console doctrine:phpcr:init:dbal
           3 $ php app/console doctrine:phpcr:repository:init
           4 $ php app/console doctrine:phpcr:fixtures:load
```

The first command will create a file called `app.sqlite` inside your `app` folder, containing the database content. The two commands after it will setup PHPCR and the final command will load some fixtures, so you can access the Standard Edition using a web server.

The project should now be accessible on your web server. If you have PHP 5.4 installed you can alternatively use the PHP internal web server:

Listing 5-7

4. <http://getcomposer.org/>
5. <http://symfony.com/doc/master/book/installation.html#configuration-and-setup>

```
1 $ php app/console server:run
```

And then access the CMF via:

Listing 5-8 1 <http://localhost:8000>



Using Other Database Backends

If you prefer to use another database backend, for example MySQL, run the configurator (point your browser to <http://localhost:8000/config.php>) or set your database connection parameters in `app/config/parameters.yml`. Make sure you leave the `database_path` property at `null` in order to use another driver than SQLite. Leaving the field blank in the web-configurator will set it to `null`.



The proper term to use for the default database of the CMF is *content repository*. The idea behind this name is essentially to describe a specialized database created specifically for content management systems. The acronym *PHPCR* actually stands for *PHP content repository*. But as mentioned before, the CMF is storage agnostic so its possible to combine the CMF with other storage mechanism, like Doctrine ORM, Propel etc.

Overview

This section will help you understand the basic parts of Symfony CMF Standard Edition (SE) and how they work together to provide the default pages you can see when browsing the Symfony CMF SE installation.

It assumes you have already installed Symfony CMF SE and have carefully read *the Symfony2 book*⁶.

AcmeMainBundle and SimpleCmsBundle

Symfony CMF SE comes with a default AcmeMainBundle to help you get started, similar to the AcmeDemoBundle provided by Symfony2. This gives you some demo pages viewable in your browser.



Where are the Controllers?

AcmeMainBundle doesn't include controllers or configuration files as you might expect. It contains little more than a Twig file and *Fixtures*⁷ data that was loaded into your database during installation. The controller logic is actually provided by the relevant CMF bundles, as described below.

There are several bundles working together in order to turn the fixture data into a browsable website. The overall, simplified process is:

- When a request is received, the *Symfony CMF Routing's Dynamic Router* is used to handle the incoming request;
- The Dynamic Router is able to match the requested URL to a `Page` document provided by SimpleCmsBundle and stored inside the database;
- The retrieved document information is used to determine which controller to pass it on to, and which template to use;
- As configured, the retrieved document is passed to `ContentController` provided by the ContentBundle, which render document into `layout.html.twig` of the AcmeMainBundle.

6. <http://symfony.com/doc/current/book/>

7. <http://symfony.com/doc/current/bundles/DoctrineFixturesBundle/index.html>

Again, this is simplified view of a very simple CMS built on top of Symfony CMF. To fully understand all the possibilities of the CMF, continue reading this Book section.

If you want to review the contents of the PHPCR database you can use the following commands:

```
Listing 5-9 1 $ php app/console doctrine:phpcr:node:dump
           2 $ php app/console doctrine:phpcr:node:dump --props
           3 $ php app/console doctrine:phpcr:node:dump /path/to/node
```

The above examples respectively show a summary, a detailed view, and a summary of a node and all its children (instead of starting at the root node).

Don't forget to look at the `--help` output for more possibilities:

```
Listing 5-10 1 $ php app/console doctrine:phpcr:node:dump --help
```

Adding new pages

Symfony CMF SE does not provide any admin tools to create new pages. If you are interested in adding an admin UI have a look at *Creating a CMS using CMF and Sonata*. However if all you want is a simple way to add new pages that you can then edit via the inline editing, then you can use the SimpleCmsBundle page migrator. The Symfony CMF SE ships with an example YAML file stored in `app/Resources/data/pages/test.yml`. The contents of this file can be loaded into the PHPCR database by calling:

```
Listing 5-11 1 $ php app/console doctrine:phpcr:migrator page --identifier=/cms/simple/test
```

Note that the above identifier is mapped to `app/Resources/data/pages/test.yml` by stripping off the basepath configuration of the SimpleCmsBundle (which defaults to `/cms/simple`).

Therefore if you want to define a child page `foo` for `/cms/simple/test` you would need to create a file `app/Resources/data/pages/test/foo.yml` and then run the following command:

```
Listing 5-12 1 $ php app/console doctrine:phpcr:migrator page --identifier=/cms/simple/test/foo
```



Chapter 6

First look at the internals

In most CMS use cases the most basic need is to associate content with a URL. In the Symfony CMF, this is done by using a powerful routing system, provided by the `RoutingBundle`, and a `ContentBundle`. The `RoutingBundle` provides a `Route` object which can be associated with the `Content` object of the `ContentBundle`.

Having two objects is the most flexible solution. You can have different routes (e.g. per language) for the same content. Or you can organize your content in a different logic than your URL tree. But in many situations, having the route and the content be one and the same simplifies things. That is exactly what the `SimpleCmsBundle` is doing, which is used by the Symfony CMF Standard Edition by default for routing, content and menus.



It's important to know that the `SimpleCmsBundle` is just a simple example how you can combine the CMF bundles into a complete CMS. Feel free to extend the `SimpleCmsBundle` or create your own bundle to do this task.



To learn more about the routing, see "[Routing](#)". To learn more about content storage, see "[Static Content](#)". Finally, to learn more about menus, see "[Structuring Content](#)".

Page Document

The `SimpleCmsBundle` provides a class called `Page` which extends from the core `Route` class and provides properties to store content and also implements the `NodeInterface`, so you can use inside the menu. This three-in-one approach is the key concept behind the bundle.

Creating a new Page

To create a page, use the `Symfony\Cmf\Bundle\SimpleCmsBundle\Doctrine\Phpcr\Page` object:

Listing 6-1


```

1 // // src/Acme/MainBundle/DataFixtures/PHPCR/LoadSimpleCms.php
2 namespace Acme\DemoBundle\DataFixtures\PHPCR;
3
4 use Symfony\Cmf\Bundle\SimpleCmsBundle\Doctrine\Phpcr\Page;
5 use Doctrine\ODM\PHPCR\DocumentManager;
6
7 class LoadRoutingData implements FixtureInterface
8 {
9     /**
10      * @param DocumentManager $dm
11      */
12     public function load(ObjectManager $dm)
13     {
14         $parent = $dm->find(null, '/cms/simple');
15         $page = new Page();
16         $page->setTitle('About Symfony CMF');
17         $page->setLabel('About');
18         $page->setBody(...);
19
20         // the tree position defines the URL
21         $page->setPosition($parent, 'about');
22
23         $dm->persist($page);
24         $dm->flush();
25     }
26 }

```

You can also set other options on the Page (e.g. tags).

All pages are stored in a simple tree structure. To set the position, use `setPosition`. The first argument is the parent document, the second the name for this page. The names are used for the URL. For instance, you may have the following tree structure:

Listing 6-2

```

1 /cmf/simple/
2   about/
3   blog/
4     symfony-cmf-is-great/

```

In this case, you have 4 pages: the page at `/cms/simple`, `about`, `blog` and `symfony-cmf-is-great`. The page at the home has the path `/`. The page `symfony-cmf-is-great` is a child of `blog` and thus has the path `/blog/symfony-cmf-is-great`. To create such a structure, you would do:

Listing 6-3

```

1 // // src/Acme/MainBundle/DataFixtures/PHPCR/LoadSimpleCms.php
2 namespace Acme\DemoBundle\DataFixtures\PHPCR;
3
4 use Symfony\Cmf\Bundle\SimpleCmsBundle\Doctrine\Phpcr\Page;
5 use Doctrine\ODM\PHPCR\DocumentManager;
6
7 class LoadRoutingData implements FixtureInterface
8 {
9     /**
10      * @param DocumentManager $dm
11      */
12     public function load(ObjectManager $dm)
13     {
14         $root = $dm->find(null, '/cms/simple');
15
16         $about = new Page();
17         // ... set up about
18         $about->setPosition($root, 'about');
19
20         $dm->persist($about);
21
22         $blog = new Page();
23         // ... set up blog
24         $blog->setPosition($root, 'blog');
25
26         $dm->persist($blog);

```

```
27
28     $blogPost = new Page();
29     // ... set up blog post
30     $blogPost->setPosition($blog, 'symfony-cmf-is-great');
31
32     $dm->persist($blogPost);
33
34     $dm->flush();
35 }
36 }
```

Every PHPCR-ODM document must have a parent document. Parents are never created automatically, so we use the PHPCR NodeHelper to ensure we have the root element (`/cmf/simple` in this case).



The Page at `/cmf/simple` is created by an initializer of the SimpleCmsBundle.

Summary

Congratulations! You are now able to create a simple web site using the Symfony CMF. From here, each chapter will tell you a bit more about the CMF and more about the things behind the SimpleCMSBundle. In the end, you'll be able to create more advanced blog systems and other CMS websites.



Chapter 7

Routing

This is an introduction to understand the concepts behind CMF routing. For the reference documentation please see the documentation for the *Routing component* and the *RoutingBundle*.

Concept

Why a new Routing Mechanism?

CMS are highly dynamic sites, where most of the content is managed by the administrators rather than developers. The number of available pages can easily reach the thousands, which is usually multiplied by the number of available translations. Best accessibility and SEO practices, as well as user preferences dictate that the URLs should be definable by the content managers.

The default Symfony2 routing mechanism, with its configuration file approach, is not the best solution for this problem. It does not provide a way of handling dynamic, user-defined routes, nor does it scale well to a large number of routes.

The Solution

In order to address these issues, a new routing system needed to be developed that takes into account the typical needs of CMS routing:

- User-defined URLs;
- Multi-site;
- Multi-language;
- Tree-like structure for easier management;
- Content, Menu and Route separation for added flexibility.

The Symfony CMF Routing component was created with these requirements in mind.

The ChainRouter

At the core of Symfony CMF's Routing component sits the `ChainRouter`. It is used as a replacement for Symfony2's default routing system and, like the Symfony2 router, is responsible for determining which Controller will handle each request.

The `ChainRouter` works by accepting a set of prioritized routing strategies, `RouterInterface`¹ implementations, commonly referred to as "Routers". The routers are responsible for matching an incoming request to an actual Controller and, to do so, the `ChainRouter` iterates over the configured Routers according to their configured priority:

```
Listing 7-1 1 # app/config/config.yml
2 cmf_routing:
3   chain:
4     routers_by_id:
5       # enable the DynamicRouter with a low priority
6       # this way the non dynamic routes take precedence
7       # to prevent needless database look ups
8       cmf_routing.dynamic_router: 20
9
10      # enable the symfony default router with a higher priority
11      router.default: 100
```

You can also load Routers using tagged services, by using the `router` tag and an optional `priority`. The higher the priority, the earlier your router will be asked to match the route. If you do not specify the priority, your router will come last. If there are several routers with the same priority, the order between them is undetermined. The tagged service will look like this:

```
Listing 7-2 1 services:
2   my_namespace.my_router:
3     class: "%my_namespace.my_router_class%"
4     tags:
5       - { name: router, priority: 300 }
```

The Symfony CMF Routing system adds a new `DynamicRouter`, which complements the default `Router` found in Symfony2.

The Default Symfony2 Router

Although it replaces the default routing mechanism, Symfony CMF Routing allows you to keep using the existing system. In fact, the standard Symfony2 routing is enabled by default, so you can keep using the routes you declared in your configuration files, or as declared by other bundles.

The DynamicRouter

This Router can dynamically load `Route` instances from a dynamic source via a so called *provider*. In fact it only loads candidate routes. The actual matching process is exactly the same as with the standard Symfony2 routing mechanism. However the `DynamicRouter` additionally is able to determine which Controller and Template to use based on the `Route` that is matched.

By default the `DynamicRouter` is disabled. To activate it, just add the following to your configuration file:

```
Listing 7-3 1 # app/config/config.yml
2 cmf_routing:
3   dynamic:
4     enabled: true
```

1. <http://api.symfony.com/1.0/Symfony/Component/Routing/RouterInterface.html>

This is the minimum configuration required to load the `DynamicRouter` as a service, thus making it capable of performing routing. Actually, when you browse the default pages that come with the Symfony CMF SE, it is the `DynamicRouter` that matches your requests with the `Controllers` and `Templates`.

Getting the Route Object

The provider to use can be configured to best suit each implementation's needs. As part of this bundle, an implementation for `Doctrine ORM`² and `PHPCR-ODM`³ is provided. Also, you can easily create your own by simply implementing the `RouteProviderInterface`. Providers are responsible for fetching an ordered subset of candidate routes that could match the request. For example the default `PHPCR-ODM`⁴ provider loads the `Route` at the path in the request and all parent paths to allow for some of the path segments being parameters.

For more detailed information on this implementation and how you can customize or extend it, refer to *RoutingBundle*.

The `DynamicRouter` is able to match the incoming request to a `Route` object from the underlying provider. The details on how this matching process is carried out can be found in the *component documentation*.



To have the route provider find routes, you also need to provide the data in your storage. With `PHPCR-ODM`, this is either done through the admin interface (see at the bottom) or with fixtures.

However, before we can explain how to do that, you need to understand how the `DynamicRouter` works. An example will come later in this document.

Getting the Controller and Template

A `Route` needs to specify which `Controller` should handle a specific `Request`. The `DynamicRouter` uses one of several possible methods to determine it (in order of precedence):

- **Explicit:** The `Route` document itself can explicitly declare the target `Controller` if one is returned from `getDefault('_controller')`.
- **By type:** The `Route` document returns a value from `getDefault('type')`, which is then matched against the provided configuration from `config.yml`
- **By class:** Requires the `Route` document to implement `RouteObjectInterface` and return an object for `getContent()`. The returned class type is then matched against the provided configuration from `config.yml`.
- **Default:** If configured, a default `Controller` will be used.

Apart from this, the `DynamicRouter` is also capable of dynamically specifying which `Template` will be used, in a similar way to the one used to determine the `Controller` (in order of precedence):

- **Explicit:** The stored `Route` document itself can explicitly declare the target `Template` by returning the name of the template via `getDefault('_template')`.
- **By class:** Requires the `Route` instance to implement `RouteObjectInterface` and return an object for `getContent()`. The returned class type is then matched against the provided configuration from `config.yml`.

Here's an example of how to configure the above mentioned options:

Listing 7-4

```
1 # app/config/config.yml
2 cmf_routing:
3     dynamic:
4         generic_controller: cmf_content.controller:indexAction
5         controllers_by_type:
6             editable_static: sandbox_main.controller:indexAction
7         controllers_by_class:
```

2. <http://www.doctrine-project.org/projects/orm.html>

3. <http://www.doctrine-project.org/projects/phpcr-odm.html>

4. <http://www.doctrine-project.org/projects/phpcr-odm.html>

```

8         Symfony\Cmf\Bundle\ContentBundle\Document\StaticContent: cmf_content.controller::indexAction
9         templates_by_class:
10        Symfony\Cmf\Bundle\ContentBundle\Document\StaticContent:
        CmfContentBundle:StaticContent:index.html.twig

```

Notice that `enabled: true` is no longer present. It's only required if no other configuration parameter is provided. The router is automatically enabled as soon as you add any other configuration to the `dynamic` entry.



Internally, the routing component maps these configuration options to several `RouteEnhancerInterface` instances. The actual scope of these enhancers is much wider, and you can find more information about them in the routing enhancers documentation section.

Linking a Route with a Model Instance

Depending on your application's logic, a requested URL may have an associated model instance from the database. Those Routes can implement the `RouteObjectInterface`, and optionally return a model instance, that will be automatically passed to the Controller as the `contentDocument` method parameter.

Note that a Route can implement the above mentioned interface but still not return any model instance, in which case no associated object will be provided.

Furthermore, Routes that implement this interface can also have a custom Route name, instead of the default Symfony core compatible name, and can contain any characters. This allows you, for example, to set a path as the route name.

Redirects

You can build redirects by implementing the `RedirectRouteInterface`. If you are using the default PHPCR-ODM route provider, a ready to use implementation is provided in the `RedirectRoute` Document. It can redirect either to an absolute URI, to a named Route that can be generated by any Router in the chain or to another Route object known to the route provider. The actual redirection is handled by a specific Controller that can be configured as follows:

Listing 7-5

```

1 # app/config/config.yml
2 cmf_routing:
3     dynamic:
4         controllers_by_class:
5             Symfony\Cmf\Component\Routing\RedirectRouteInterface:
                cmf_routing.redirect_controller:redirectAction

```



The actual configuration for this association exists as a service, not as part of a `config.yml` file. As discussed before, any of the approaches can be used.

URL Generation

Symfony CMF's Routing component uses the default Symfony2 components to handle route generation, so you can use the default methods for generating your URLs with a few added possibilities:

- Pass an implementation of either `RouteObjectInterface` or `RouteReferrersInterface` as the `name` parameter

- Alternatively, supply an implementation of `ContentRepositoryInterface` and the id of the model instance as parameter `content_id`

See URL generation with the `DynamicRouter` for code examples of all above cases.

The route generation handles locales as well, see "ContentAwareGenerator and Locales".

The PHPCR-ODM Route Document

As mentioned above, you can use any route provider. The example in this section applies if you use the default PHPCR-ODM route provider (`Symfony\Cmf\Bundle\RoutingBundle\Doctrine\Phpcr\RouteProvider`).

All routes are located under the base path configured in the application configuration `cmf_routing.persistence.phpcr.route_basepath`. By default this path is `/cms/routes`. A new route can be created in PHP code as follows:

```
Listing 7-6
1 // src/Acme/MainBundle/DataFixtures/PHPCR/LoadRoutingData.php
2 namespace Acme\DemoBundle\DataFixtures\PHPCR;
3
4 use Doctrine\ODM\PHPCR\DocumentManager;
5 use Symfony\Cmf\Bundle\RoutingBundle\Doctrine\Phpcr\Route;
6 use Symfony\Cmf\Bundle\ContentBundle\Doctrine\Phpcr\StaticContent;
7
8 class LoadRoutingData implements FixtureInterface
9 {
10     /**
11      * @param DocumentManager $dm
12      */
13     public function load(ObjectManager $dm)
14     {
15         $route = new Route();
16         $route->setParent($dm->find(null, '/cms/routes'));
17         $route->setName('projects');
18
19         // link a content to the route
20         $content = new StaticContent();
21         $content->setParent($dm->find(null, '/cms/content'));
22         $content->setName('my-content');
23         $dm->persist($content);
24         $route->setRouteContent($content);
25
26         // now define an id parameter; do not forget the leading slash if you
27         // want /projects/{id} and not /projects{id}
28         $route->setVariablePattern('/{id}');
29         $route->setRequirement('id', '\d+');
30         $route->setDefault('id', 1);
31
32         $dm->persist($route),
33         $dm->flush();
34     }
35 }
```

This will give you a document that matches the URL `/projects/<number>` but also `/projects` as there is a default for the id parameter.

Because you defined the `{id}` route parameter, your controller can expect an `$id` parameter. Additionally, because you called `setRouteContent` on the route, your controller can expect the `$contentDocument` parameter. The content could be used to define an intro section that is the same for each project or other shared data. If you don't need content, you can just not set it in the document.

For more details, see the route document section in the `RoutingBundle` documentation.

Further Notes

For more information on the Routing component of Symfony CMF, please refer to:

- *Routing* for most of the actual functionality implementation
- *RoutingBundle* for Symfony2 integration bundle for Routing Bundle
- Symfony2's *Routing*⁵ component page
- *Handling Multi-Language Documents* for some notes on multilingual routing

5. <http://symfony.com/doc/current/components/routing/introduction.html>



Chapter 8

The Database Layer: PHPCR-ODM

The Symfony CMF is storage layer agnostic, meaning that it can work with many storage layers. By default, the Symfony CMF works with the *Doctrine PHPCR-ODM*¹. In this chapter, you will learn how to work with the Doctrine PHPCR-ODM.



Read more about choosing the correct storage layer in *Choosing a Storage Layer*

PHPCR: A Tree Structure

The Doctrine PHPCR-ODM is a doctrine object-mapper on top of the *PHP Content Repository*² (PHPCR), which is a PHP adaption of the *JSR-283 specification*³. The most important feature of PHPCR is the tree structure to store the data. All data is stored in items of a tree, called nodes. You can think of this like a file system, that makes it perfect to use in a CMS.

On top of the tree structure, PHPCR also adds features like searching, versioning and access control.

Doctrine PHPCR-ODM has the same API as the other Doctrine libraries, like the *Doctrine ORM*⁴. The Doctrine PHPCR-ODM adds another great feature to PHPCR: Multilanguage support.



PHPCR Implementations

In order to let the Doctrine PHPCR-ODM communicate with the PHPCR, a PHPCR implementation is needed. *Jackalope*⁵ is the reference PHPCR implementation, which can work with *Apache Jackrabbit*⁶ (with the *jackalope-jackrabbit*⁷ package) and with Doctrine DBAL (providing support for postgres, sqlite and mysql) with the *jackalope-doctrine-dbal*⁸ package.

-
1. <http://docs.doctrine-project.org/projects/doctrine-phpcr-odm/en/latest/index.html>
 2. <http://phpcr.github.io/>
 3. <http://jcp.org/en/jsr/detail?id=283>
 4. <http://symfony.com/doc/current/book/doctrine.html>
 5. <http://jackalope.github.io/>
 6. <http://jackrabbit.apache.org/>
 7. <https://github.com/jackalope/jackalope-jackrabbit>

A Simple Example: A Task

The easiest way to get started with the PHPCR-ODM is to see it in action. In this section, you are going to create a Task object and learn how to persist it.

Creating a Document Class

Without thinking about Doctrine or PHPCR-ODM, you can create a Task object in PHP:

```
Listing 8-1 1 // src/Acme/TaskBundle/Document/Task.php
2 namespace Acme\TaskBundle\Document;
3
4 class Task
5 {
6     protected $description;
7
8     protected $done = false;
9 }
```

This class - often called a "document" in PHPCR-ODM, meaning *a basic class that holds data* - is simple and helps fulfill the business requirement of needing tasks in your application. This class can't be persisted to Doctrine PHPCR-ODM yet - it's just a simple PHP class.



A Document is analogous to the term Entity employed by the Doctrine ORM. You must add this object to the Document sub-namespace of you bundle, in order register the mapping data automatically.

Add Mapping Information

Doctrine allows you to work with PHPCR in a much more interesting way than just fetching data back and forth as an array. Instead, Doctrine allows you to persist entire objects to PHPCR and fetch entire *objects* out of PHPCR. This works by mapping a PHP class and its properties to the PHPCR tree.

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

```
Listing 8-2 1 // src/Acme/TaskBundle/Document/Task.php
2 namespace Acme\TaskBundle\Document;
3
4 use Doctrine\ODM\PHPCR\Mapping\Annotations as PHPCR;
5
6 /**
7  * @PHPCR\Document()
8  */
9 class Task
10 {
11     /**
12      * @PHPCR\Id()
13      */
14     protected $id;
15
16     /**
17      * @PHPCR\String()
18      */
19     protected $description;
20
21     /**
22      * @PHPCR\Boolean()
```

8. <https://github.com/jackalope/jackalope-doctrine-dbal>

```

23     */
24     protected $done = false;
25
26     /**
27      * @PHPCR\ParentDocument()
28      */
29     protected $parent;
30 }

```

After this, you have to create getters and setters for the properties.



This Document uses the parent document and a node name to determine its position in the tree. Because there isn't any name set, it is generated automatically. If you want to use a specific node name, such as a sluggified version of the title, you need to add a property mapped as `Nodename`.

A Document must have an `id` property. This represents the full path (parent + name) of the Document. This will be set by Doctrine by default and it is not recommend to use the `id` to determine the location of a Document.

For more information about identifier generation strategies, refer to the *doctrine documentation*⁹

You can also check out Doctrine's Basic Mapping Documentation¹⁰ for all details about mapping information. If you use annotations, you'll need to prepend all annotations with `PHPCR\`, which is the name of the imported namespace (e.g. `PHPCR\Document(..)`), this is not shown in Doctrine's documentation. You'll also need to include the `USE use Doctrine\ODM\PHPCR\Mapping\Annotations as PHPCR;` statement, which imports the PHPCR annotations prefix.

Persisting Documents to PHPCR

Now that you have a mapped Task document, complete with getter and setter methods, you're ready to persist data to PHPCR. From inside a controller, this is pretty easy, add the following method to the `DefaultController` of the `AcmeTaskBundle`:

Listing 8-3

```

1 // src/Acme/TaskBundle/Controller/DefaultController.php
2
3 // ...
4 use Acme\TaskBundle\Document\Task;
5 use Symfony\Component\HttpFoundation\Response;
6
7 // ...
8 public function createAction()
9 {
10     $documentManager = $this->get('doctrine_phpcr')->getManager();
11
12     $rootTask = $documentManager->find(null, '/tasks');
13
14     $task = new Task();
15     $task->setDescription('Finish CMF project');
16     $task->setParent($rootTask);
17
18     $documentManager->persist($task);
19
20     $documentManager->flush();
21
22     return new Response('Created task "'. $task->getDescription(). "'");
23 }

```

Take a look at the previous example in more detail:

9. <http://docs.doctrine-project.org/projects/doctrine-odm/en/latest/reference/basic-mapping.html#basicmapping-identifier-generation-strategies>

10. <http://docs.doctrine-project.org/projects/doctrine-odm/en/latest/reference/annotations-reference.html>

- **line 10** This line fetches Doctrine's *document manager* object, which is responsible for handling the process of persisting and fetching objects to and from PHPCR.
- **line 12** This line fetches the root document for the tasks, as each Document needs to have a parent. To create this root document, you can configure a Repository Initializer, which will be executed when running `doctrine:phpcr:repository:init`.
- **lines 14-16** In this section, you instantiate and work with the `$task` object like any other, normal PHP object.
- **line 18** The `persist()` method tells Doctrine to "manage" the `$task` object. This does not actually cause a query to be made to PHPCR (yet).
- **line 20** When the `flush()` method is called, Doctrine looks through all of the objects that it is managing to see if they need to be persisted to PHPCR. In this example, the `$task` object has not been persisted yet, so the document manager makes a query to PHPCR, which adds a new document.

When creating or updating objects, the workflow is always the same. In the next section, you'll see how Doctrine is smart enough to update documents if they already exist in PHPCR.

Fetching Objects from PHPCR

Fetching an object back out of PHPCR is even easier. For example, suppose you've configured a route to display a specific task by name:

Listing 8-4

```

1 public function showAction($name)
2 {
3     $repository = $this->get('doctrine_phpcr')->getRepository('AcmeTaskBundle:Task');
4     $task = $repository->find('/task/'.$name);
5
6     if (!$task) {
7         throw $this->createNotFoundException('No task found with name '.$name);
8     }
9
10    return new Response(['.($task->isDone() ? 'x' : ' ').''].$task->getDescription());
11 }

```

To retrieve objects from the document repository using both the `find` and `findMany` methods and all helper methods of a class-specific repository. In PHPCR, it's often unknown for developers which node has the data for a specific document, in that case you should use the document manager to find the nodes (for instance, when you want to get the root document). In example above, we know they are Task documents and so we can use the repository.

The repository contains all sorts of helpful methods:

Listing 8-5

```

1 // query by the id (full path)
2 $task = $repository->find($id);
3
4 // query for one task matching by name and done
5 $task = $repository->findOneBy(array('name' => 'foo', 'done' => false));
6
7 // query for all tasks matching the name, ordered by done
8 $tasks = $repository->findBy(
9     array('name' => 'foo'),
10    array('done' => 'ASC')
11 );

```



If you use the repository class, you can also create a custom repository for a specific document. This helps Separation of Concern when using more complex queries. This is similar to how it's done in Doctrine ORM, for more information read "*Custom Repository Classes*"¹¹ in the core documentation.

11. <http://symfony.com/doc/current/book/doctrine.html#custom-repository-classes>



You can also query objects by using the Query Builder provided by Doctrine PHPCR-ODM. For more information, read *the QueryBuilder documentation*¹².

Updating an Object

Once you've fetched an object from Doctrine, updating it is easy. Suppose you have a route that maps a task ID to an update action in a controller:

```
Listing 8-6 1 public function updateAction($name)
2 {
3     $documentManager = $this->get('doctrine_phpcr')->getManager();
4     $repository = $documentManager->getRepository('AcmeTaskBundle:Task');
5     $task = $repository->find('/tasks/'.$name);
6
7     if (!$task) {
8         throw $this->createNotFoundException('No task found for name '.$name);
9     }
10
11    if (!$task->isDone()) {
12        $task->setDone(true);
13    }
14
15    $documentManager->flush();
16
17    return new Response(['x' '.$task->getDescription()]);
18 }
```

Updating an object involves just three steps:

1. fetching the object from Doctrine;
2. modifying the object;
3. calling `flush()` on the document manager

Notice that calling `$documentManger->persist($task)` isn't necessary. Recall that this method simply tells Doctrine to manage or "watch" the `$task` object. In this case, since you fetched the `$task` object from Doctrine, it's already managed.

Deleting an Object

Deleting an object is very similar, but requires a call to the `remove()` method of the document manager after you fetched the document from PHPCR:

```
Listing 8-7 $documentManager->remove($task);
            $documentManager->flush();
```

As you might expect, the `remove()` method notifies Doctrine that you'd like to remove the given document from PHPCR. The actual delete operation however, is not actually executed until the `flush()` method is called.

Summary

With Doctrine, you can focus on your objects and how they're useful in your application and worry about database persistence second. This is because Doctrine allows you to use any PHP object to hold your data and relies on mapping metadata information to map an object's data to a particular database table.

And even though Doctrine revolves around a simple concept, it's incredibly powerful, allowing you to create complex queries and subscribe to events that allow you to take different actions as objects go through their persistence lifecycle.

12. <http://docs.doctrine-project.org/projects/doctrine-odm/en/latest/reference/query-builder.html>



Chapter 9

Static Content

Concept

At the heart of every CMS stands the content, an abstraction that the publishers can manipulate and that will later be presented to the page's users. The content's structure greatly depends on the project's needs, and it will have a significant impact on future development and use of the platform.

The `ContentBundle` provides a basic implementation of a content document classes, including support for multiple languages and association to Routes.

Static Content

The `StaticContent` class declares the basic content's structure. Its structure is very similar to the ones used on Symfony2's ORM systems. Most of its fields are self explanatory and are what you would expect from a basic CMS: title, body, publishing information and a parent reference, to accommodate a tree-like hierarchy. It also includes a Block reference (more on that later).

The two implemented interfaces reveal two of the features included in this implementation:

- `RouteReferrersInterface` means that the content has associated Routes.
- `PublishWorkflowInterface` **means that the content has publishing and** unpublishing dates, which will be handled by Symfony CMF's core to determine whether or not to display the content from `StaticContent`.

Multilang Static Content

The `MultilangStaticContent` class extends `StaticContent`, offering the same functionality with multi language support. It specifies which fields are to be translated (title, body and tags) as well as a variable to declare the locale.

It also specifies the translation strategy:

Listing 9-1

```
1 use Doctrine\ODM\PHPCR\Mapping\Annotations as PHPCR;
2
3 /**
4  * @PHPCR\Document(translator="child", referenceable=true)
5  */
```

For information on the available translation strategies, refer to the Doctrine page regarding *multilanguage support in PHPCR-ODM*¹.

Content Controller

A controller is also included that can render either of the above content document types. Its single action, `indexAction`, accepts a content instance and optionally the path of the template to be used for rendering. If no template path is provided, it uses a pre-configured default.

The controller action also takes into account the document's publishing status and language (for `MultilangStaticContent`). Both the content instance and the optional template are provided to the controller by the `DynamicRouter` of the `RoutingBundle`. More information on this is available on the `Routing system getting started` page.

Admin Support

The last component needed to handle the included content types is an administration panel. Symfony CMF can optionally support *SonataDoctrinePHPCRAdminBundle*², a back office generation tool.

In `ContentBundle`, the required administration panels are already declared in the `Admin` folder and configured in `Resources/config/admin.xml`, and will automatically be loaded if you install the `SonataDoctrinePHPCRAdminBundle` (refer to *Creating a CMS using CMF and Sonata* for instructions on that).

Configuration

The `ContentBundle` also supports a set of optional configuration parameters. Refer to *ContentBundle* for the full configuration reference.

Final Thoughts

While this small bundle includes some vital components to a fully working CMS, it often will not provide all you need. The main idea behind it is to provide developers with a small and easy to understand starting point you can extend or use as inspiration to develop your own content types, Controllers and Admin panels.

1. <http://docs.doctrine-project.org/projects/doctrine-phpcr-odm/en/latest/reference/multilang.html>

2. <https://github.com/sonata-project/SonataDoctrinePhpcrAdminBundle>



Chapter 10

Structuring Content

Menu Bundle

Concept

No CMS system is complete without a menu system that allows users to navigate between content pages and perform certain actions. While it usually maps the actual content tree structure, menus often have a logic of their own, include options not mapped by content or exist in multiple contexts with multiple options, thus making them a complex problem themselves.

Symfony CMF Menu System

Symfony CMF SE includes the MenuBundle, a tool that allow you to dynamically define your menus. It extends the *KnpmenuBundle*¹, with a set of hierarchical, multi language menu elements, along with the tools to persist them in the chosen content store. It also includes the administration panel definitions and related services needed for integration with the *SonataDoctrinePHPCRAAdminBundle*².



The MenuBundle extends and greatly relies on the *KnpmenuBundle*³, so you should carefully read *KnpmenuBundle's documentation*⁴. For the rest of this page we assume you have done so and are familiar with concepts like Menu Providers and Menu Factories.

Usage

The MenuBundle uses KnpmenuBundle's default renderers and helpers to print out menus. You can refer to the *respective documentation page*⁵ for more information on the subject, but a basic call would be:

Listing 10-1

-
1. <https://github.com/knplabs/KnpMenuBundle>
 2. <http://sonata-project.org/bundles/doctrine-phpcr-admin/master/doc/index.html>
 3. <https://github.com/knplabs/KnpMenuBundle>
 4. <https://github.com/KnpLabs/KnpMenuBundle/blob/master/Resources/doc/index.md>
 5. <https://github.com/KnpLabs/KnpMenuBundle/blob/master/Resources/doc/index.md#rendering-menus>


```
1 {{ knp_menu_render('simple') }}
```

The provided menu name will be passed on to `MenuProviderInterface` implementation, which will use it to identify which menu you want rendered in this specific section.

The Provider

The core of the `MenuBundle` is `PhpcrMenuProvider`, a `MenuProviderInterface` implementation that's responsible for dynamically loading menus from a PHPCR database. The default provider service is configured with a `menu_basepath` to know where in the PHPCR tree it will find menus. The menu name is given when rendering the menu and must be a direct child of the menu base path. This allows the `PhpcrMenuProvider` to handle several menu hierarchies using a single storage mechanism.

To give a concrete example, if we have the configuration as given below and render the menu `simple`, the menu root node must be stored at `/cms/menu/simple`.

Listing 10-2

```
1 cmf_menu:
2   menu_basepath: /cms/menu
```

If you need multiple menu roots, you can create further `PhpcrMenuProvider` instances and register them with `Knpmenu` - see the `CMF MenuBundle DependencyInjection` code for the details.

The menu element fetched using this process is used as the menu root node, and its children will be loaded progressively as the full menu structure is rendered by the `MenuFactory`.

The Factory

The `ContentAwareFactory` is a `FactoryInterface` implementation, which generates the full `MenuItem` hierarchy from the provided `MenuNode`. The data generated this way is later used to generate the actual HTML representation of the menu.

The included implementation focuses on generating `MenuItem` instances from `NodeInterface` instances, as this is usually the best approach to handle tree-like structures typically used by a CMS. Other approaches are implemented in the base classes, and their respective documentation pages can be found in `KnpmenuBundle`'s page.

`ContentAwareFactory` is responsible for loading the full menu hierarchy and transforming the `MenuNode` instances from the root node it receives from the `MenuProviderInterface` implementation. It is also responsible for determining which (if any) menu item is currently being viewed by the user. It supports a voter mechanism to have custom code decide what menu item is the current item. `Knpmenu` already includes a specific factory targeted at `Symfony2`'s Routing component, which this bundle extends, to add support for:

- `Route` instances stored in a database (refer to `RoutingBundle`'s `RouteProvider` for more details on this)
- `Route` instances with associated content (more on this on respective `RoutingBundle`'s section)

As mentioned before, `ContentAwareFactory` is responsible for loading all the menu nodes from the provided root element. The actual loaded nodes can be of any class, even if it's different from the root's, but all must implement `NodeInterface` in order to be included in the generated menu.

The Menu Nodes

Also included in the `MenuBundle` are two menu node content types: `MenuNode` and `MultilangMenuNode`. If you have read the documentation page regarding *Static Content*, you'll find this implementation somewhat familiar.

`MenuNode` implements the above mentioned `NodeInterface`, and holds the information regarding a single menu entry: a `label` and a `uri`, a `children` list, plus some `attributes` for the node and its children that

6. <https://github.com/knplabs/KnpMenuBundle>

will allow the rendering process to be customized. It also includes a `Route` field and two references to `Contents`. These are used to store an associated `Route` object, plus one (not two, despite the fact that two fields exist) `Content` element. The `MenuNode` can have a strong (integrity ensured) or weak (integrity not ensured) reference to the actual `Content` element it points to; it's up to you to choose which best fits your scenario. You can find more information on references on the *Doctrine PHPCR documentation page*⁷.

`MultilangMenuNode` extends `MenuNode` with multilanguage support. It adds a `locale` field to identify which translation set it belongs to, plus `label` and `uri` fields marked as `translated=true`. This means they will differ between translations, unlike the other fields.

`MultilangMenuNode` also specifies the strategy used to persist multiple translations:

```
Listing 10-3 1 /**
              2  * @PHPCRODM\Document(translator="attribute")
              3  */
```

For information on the available translation strategies, refer to the Doctrine page regarding *Multi language support in PHPCR-ODM*⁸

Admin Support

The `MenuBundle` also includes the administration panels and respective services needed for integration with the backend admin tool `SonataDoctrinePHPCRAdminBundle`⁹.

The included administration panels are automatically available but need to be explicitly put on the dashboard if you want to use them. See *Creating a CMS using CMF and Sonata* for instructions on how to install `SonataDoctrinePHPCRAdminBundle`.

Configuration

This bundle is configurable using a set of parameters, but all of them are optional. You can go to the *MenuBundle* reference page for the full configuration options list and additional information.

7. <http://docs.doctrine-project.org/projects/doctrine-phpcr-odm/en/latest/reference/association-mapping.html#references>

8. <http://docs.doctrine-project.org/projects/doctrine-phpcr-odm/en/latest/reference/multilang.html>

9. <http://sonata-project.org/bundles/doctrine-phpcr-admin/master/doc/index.html>

