Symfony

The Symfony CMF Book

Version: master

generated on July 21, 2017

Contents at a Glance

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. https://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
```
$ composer create-project symfony-cmf/standard-edition cmf '1.2.1'
```

The *AcmeDemoBundle* that is used in this tour was removed in version 1.3 of the Symfony CMF Standard Edition. Since then it has become the skeleton for a simple CMS application. This is why we install version 1.2.1. If you insist on checking out the most recent versions of the CMF, check out *symfony-cmf/cmf-sandbox*.

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 the Symfony CMF needs a database in order to make a lot of things configurable using an admin interface.

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

Listing 1-2
```
$ php bin/console doctrine:database:create
$ php bin/console doctrine:phpcr:init:dbal
$ php bin/console doctrine:phpcr:repository:init
$ php bin/console doctrine:phpcr:fixtures:load
```

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

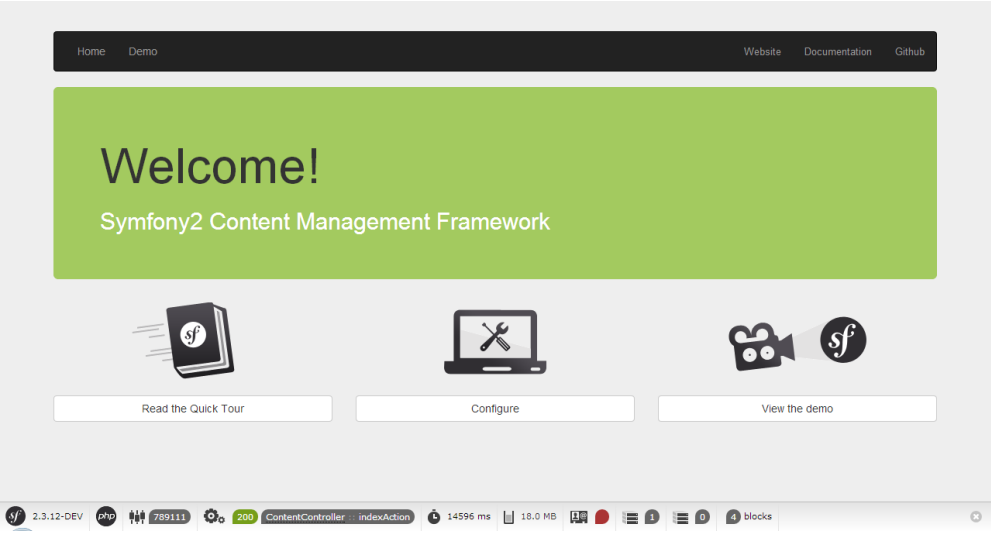
For a complete installation guide, see the "Installing the Standard Edition" chapter of the Book.

The Request Flow

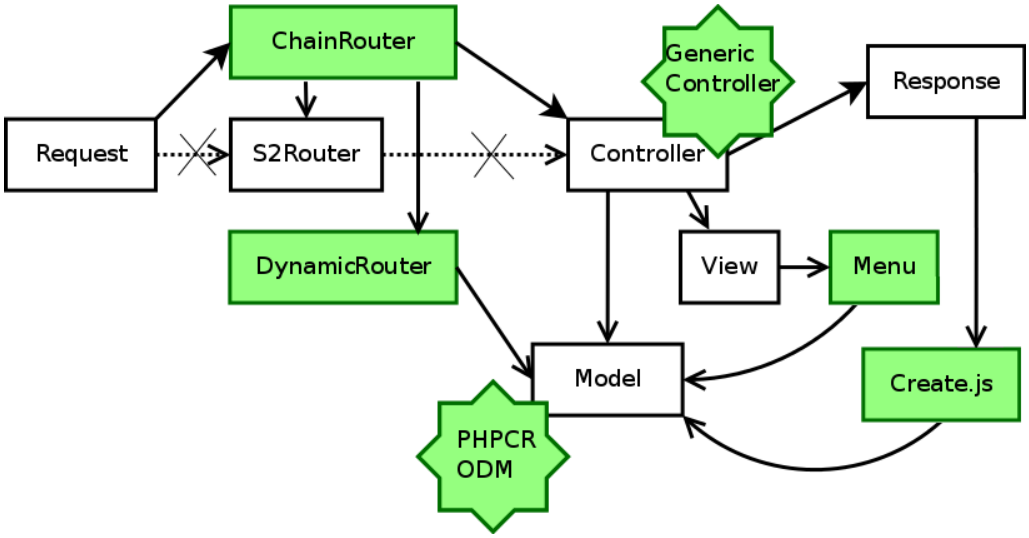
When you have at least PHP 5.4, use the `server:run` command to run a local server for the demo. Otherwise, use a `localhost` and prefix the URLs in this document with `/path-to-project/web/app_dev.php/`.

Now, the Standard Edition is ready to use. Navigate to the homepage (`http://localhost:8000/`) to see the demo:

4. https://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.

5. https://en.wikipedia.org/wiki/Content_repository_API_for_Java
6. http://phpcr.github.io/

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.

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 *further in 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 `src/Acme/DemoBundle/Resources/data` directory:

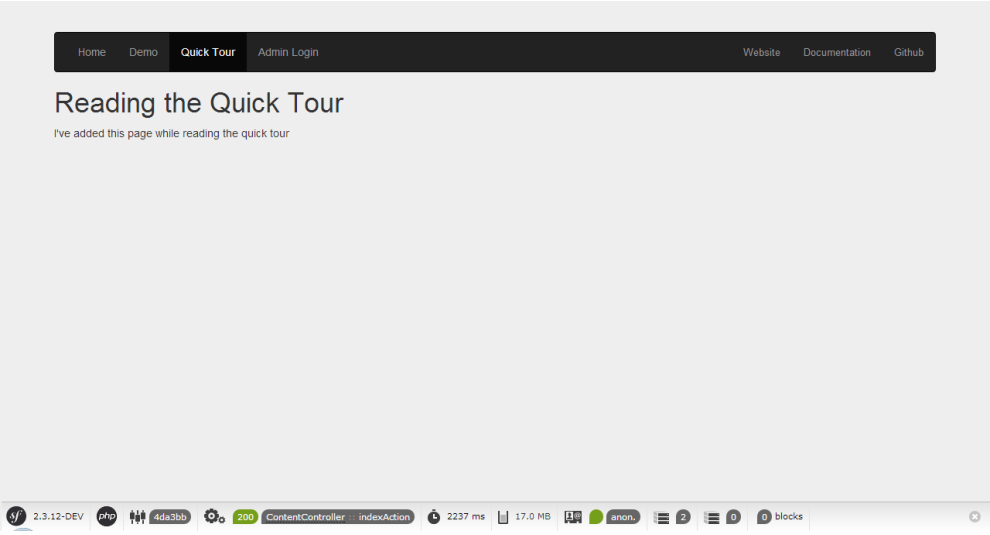
Listing 1-3

```
# src/Acme/DemoBundle/Resources/data/pages.yml
Symfony\Cmf\Bundle\SimpleCmsBundle\Doctrine\Phpcr\Page:
    # ...

    quick_tour:
        id: /cms/simple/quick_tour
        label: "Quick Tour"
```

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

```
8           title: "Reading the Quick Tour"
9           body: "I've added this page while reading the quick tour"
```

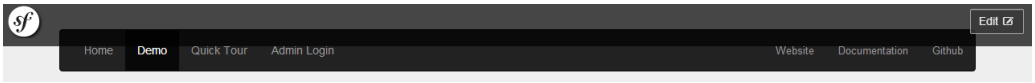
After this, you need to run the `doctrine:phpcr:fixtures:load` to reflect the changes on the database and after refreshing, you can see your new page!



Live Editing

Now is the time you become an admin of this site and editing the content using the Web Interface. To do this click on "Admin Login" and use the provided credentials.

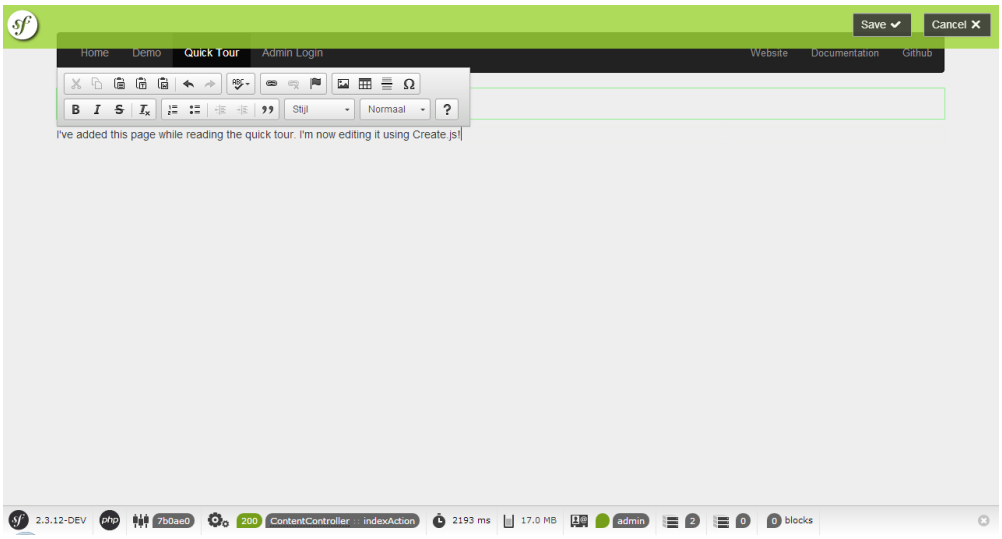
You'll see that you have a new bar at the top of the page:



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
```
$ php bin/console doctrine:phpcr:node:dump
```

The result will be the PHPCR tree:

Listing 2-2
```
ROOT:
  cms:
    simple:
      about:
      contact:
        map:
        team:
      quick_tour:
      dynamic:
      docs:
      demo:
      demo_redirect:
      hardcoded_dynamic:
      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 it's 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 yaml 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 AcmeDemoBundle:

Listing 2-3
```php
// src/Acme/DemoBundle/DataFixtures/PHPCR/LoadPageData.php
namespace Acme\DemoBundle\DataFixtures\PHPCR;

use Doctrine\Common\Persistence\ObjectManager;
use Doctrine\Common\DataFixtures\FixtureInterface;
use Doctrine\Common\DataFixtures\OrderedFixtureInterface;

class LoadPageData implements FixtureInterface, OrderedFixtureInterface
{
    public function getOrder()
    {
        // refers to the order in which the class' load function is called
        // (lower return values are called first)
        return 10;
    }

    public function load(ObjectManager $documentManager)
    {
    }
}
```

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
```php
use Doctrine\ODM\PHPCR\DocumentManager;
use Symfony\Cmf\Bundle\SimpleCmsBundle\Doctrine\Phpcr\Page;

// ...
```

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

```
5  public function load(ObjectManager $documentManager)
6  {
7      if (!$documentManager instanceof DocumentManager) {
8          $class = get_class($documentManager);
9          throw new \RuntimeException("Fixture requires a PHPCR ODM DocumentManager instance, instance of
10 '$class' given.");
11     }
12
13     $page = new Page(); // create a new Page object (document)
14     $page->setName('new_page'); // the name of the node
15     $page->setLabel('Another new Page');
16     $page->setTitle('Another new Page');
17     $page->setBody('I have added this page myself!');
   }
```

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

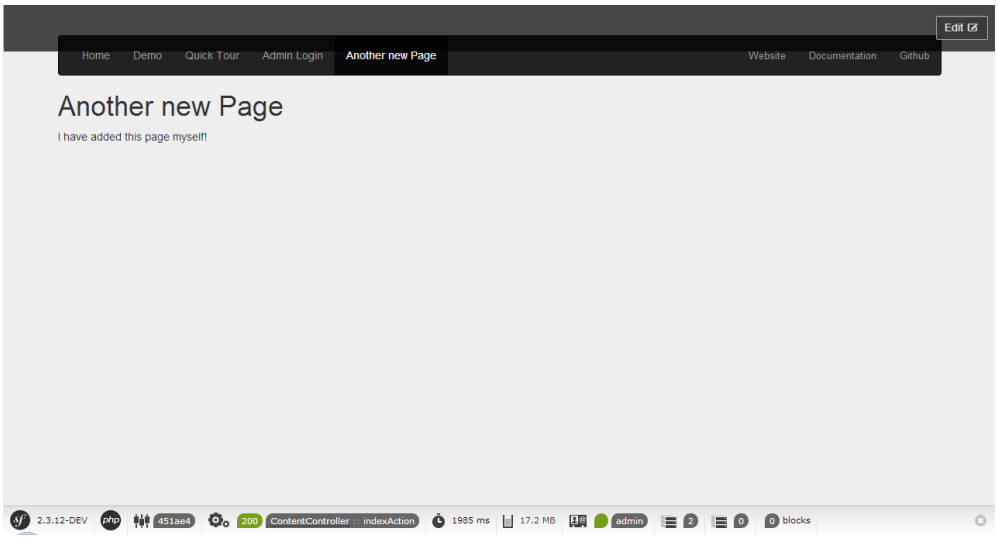
```
1  // ...
2  public function load(ObjectManager $documentManager)
3  {
4      if (!$documentManager instanceof DocumentManager) {
5          $class = get_class($documentManager);
6          throw new \RuntimeException("Fixture requires a PHPCR ODM DocumentManager instance, instance of
7  '$class' given.");
8      }
9
10     // ...
11
12     // get root document (/cms/simple)
13     $simpleCmsRoot = $documentManager->find(null, '/cms/simple');
14
15     $page->setParentDocument($simpleCmsRoot); // set the parent to the root
   }
```

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 $documentManager)
3  {
4      if (!$documentManager instanceof DocumentManager) {
5          $class = get_class($documentManager);
6          throw new \RuntimeException("Fixture requires a PHPCR ODM DocumentManager instance, instance of
7  '$class' given.");
8      }
9
10     // ...
11     $documentManager->persist($page); // add the Page in the queue
12     $documentManager->flush(); // add the Page to PHPCR
   }
```

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 powerful 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 at 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

```
/cms
    /simple
        /about       # /about Route
        /contact     # /contact Route
            /team    # /contact/team Route
            /docs    # /contact/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

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

# ...
cmf_routing:
    chain:
        routers_by_id:
            # the standard DynamicRouter
            cmf_routing.dynamic_router: 200

            # the core symfony router
            router.default: 100
    dynamic:
        persistence:
            phpcr:
                route_basepaths:
                    - /cms/routes
            # /cms/routes is the default base path, the above code is
            # equivalent to:
            # phpcr: true
```

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

Listing 3-3

```php
// src/Acme/DemoBundle/DataFixtures/PHPCR/LoadRoutingData.php
namespace Acme\DemoBundle\DataFixtures\PHPCR;

use Doctrine\Common\Persistence\ObjectManager;
use Doctrine\Common\DataFixtures\FixtureInterface;
use Doctrine\Common\DataFixtures\OrderedFixtureInterface;
use Doctrine\ODM\PHPCR\DocumentManager;

use PHPCR\Util\NodeHelper;

use Symfony\Cmf\Bundle\RoutingBundle\Doctrine\Phpcr\Route;

class LoadRoutingData implements FixtureInterface, OrderedFixtureInterface
{
    public function getOrder()
```

```
    {
        return 20;
    }

    public function load(ObjectManager $documentManager)
    {
        if (!$documentManager instanceof DocumentManager) {
            $class = get_class($documentManager);
            throw new \RuntimeException("Fixture requires a PHPCR ODM DocumentManager instance, instance of
'$class' given.");
        }

        $session = $documentManager->getPhpcrSession();
        NodeHelper::createPath($session, '/cms/routes');

        $routesRoot = $documentManager->find(null, '/cms/routes');

        $route = new Route();
        // set $routesRoot as the parent and 'new-route' as the node name,
        // this is equal to:
        // $route->setName('new-route');
        // $route->setParentDocument($routesRoot);
        $route->setPosition($routesRoot, 'new-route');

        $page = $documentManager->find(null, '/cms/simple/quick_tour');
        $route->setContent($page);

        $documentManager->persist($route); // put $route in the queue
        $documentManager->flush(); // save it
    }
}
```

Above we implemented the `OrderedFixtureInterface` so that our routes were loaded in the correct sequence relative to other fixtures.

Now execute the `doctrine:phpcr:fixtures:load` command again.

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 *KnpMenuBundle*[1] or *SonataAdminBundle*[2].

The MenuBundle

Let's start with the MenuBundle. If you visit the page, you can see a nice menu. You can find the rendering of this menu in the layout view in the AcmeDemoBundle:

Listing 4-1
```
<!-- src/Acme/DemoBundle/Resources/views/layout.html.twig -->

<!-- ... -->
<nav class="navbar  navbar-inverse  page__nav" role="navigation">
    <div class="container-fluid">
        {{ knp_menu_render('simple', {'template': 'AcmeDemoBundle:Menu:bootstrap.html.twig', 'currentClass':
'active'}) }}

        <!-- ... -->
    </div>
</nav>
```

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

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`).

1. https://github.com/KnpLabs/KnpMenuBundle
2. https://sonata-project.org/bundles/admin/master/doc/index.html

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].

The Create.js library works using a REST layer. All elements on a page get *RDFa Mappings*[6], which tells Create.js how to map the element to the document. When you save the page, the new content is passed to the REST api and saved in the database.

Rendering content with RDFa mappings can be very easy, as you can see in the Standard Edition:

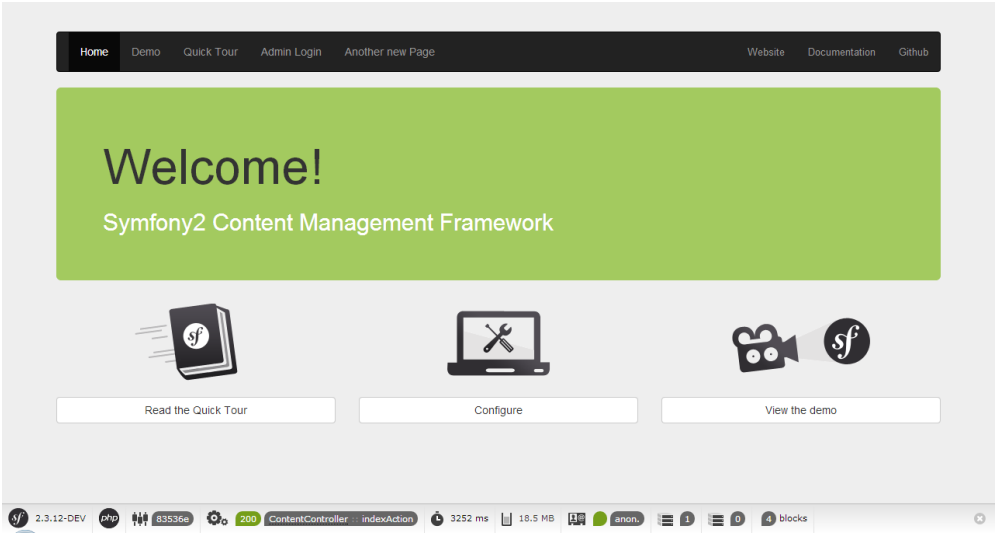
Listing 4-2

```
{% block main %}
{% createphp cmfMainContent as="rdf" %}
{{ rdf|raw }}
{% endcreatephp %}
{% endblock %}
```

This will output the content object using <*div*> elements. You can also customize this completely by using the `createphp_*` functions.

The BlockBundle

If you visit the homepage of the Standard Edition, you'll see three blocks:



These blocks can be edited and used on their own. These blocks are provided by the BlockBundle, which is a tiny layer on top of the *SonataBlockBundle*[7]. It provides the ability to store the blocks using PHPCR and it adds some commonly used blocks.

3. http://demo.createphp.org/
4. http://createjs.org/
5. https://github.com/friendsofsymfony/FOSRestBundle
6. https://en.wikipedia.org/wiki/RDFa
7. https://sonata-project.org/bundles/block/master/doc/index.html

The three blocks in the Standard Edition are custom blocks. A block is handled by a block service. You can find this service in the `Acme\DemoBundle\Block\UnitBlockService` class. Since the blocks are persisted using PHPCR, it also needs a block document, which is located in `Acme\DemoBundle\Document\UnitBlock`.

The SeoBundle

There is also a SeoBundle. This bundle is build on top of the *SonataSeoBundle*[8]. It provides a way to extract SEO information from a document and to make SEO information editable using an admin.

To integrate the SeoBundle into the Standard Edition, you need to include it in your project with `composer require symfony-cmf/seo-bundle` and then register both the CMF and the Sonata bundle in the `AppKernel`:

Listing 4-3
```php
// app/AppKernel.php

// ...
public function registerBundles()
{
    $bundles = array(
        // ...
        new Sonata\SeoBundle\SonataSeoBundle(),
        new Symfony\Cmf\Bundle\SeoBundle\CmfSeoBundle(),
    );
    // ...
}
```

Now, you can configure a standard title. This is the title that is used when the CmfSeoBundle can extract the title from a content object:

Listing 4-4
```yaml
# app/config/config.yml
cmf_seo:
    title: "%%content_title%% | Standard Edition"
```

The `%%content_title%%` will be replaced by the title extracted from the content object. The last thing you need to do is using this title as the title element. To do this, replace the `<title>` tag line in the `src/Acme/DemoBundle/Resources/views/layout.html.twig` template with this:

Listing 4-5
```twig
{% block title %}{{ sonata_seo_title() }}{% endblock %}
```

When you visit the new website, you can see nice titles for each page!

Some pages, like the login page, don't use content objects. In these cases, you can configure a default title:

Listing 4-6
```yaml
# app/config/config.yml
sonata_seo:
    page:
        title: Standard Edition
```

The *default title* is configured under the `sonata_seo` extension, while the *standard title* is configured under the `cmf_seo` extension.

The title is just one feature of the SeoBundle, it can extract and process a lot more SEO information.

8. https://sonata-project.org/bundles/seo/master/doc/index.html

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*[9]. 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*[10] is installed. You can switch off the Admin class in the configuration. For instance, to disable the MenuBundle Admin classes, you would do:

Listing 4-7

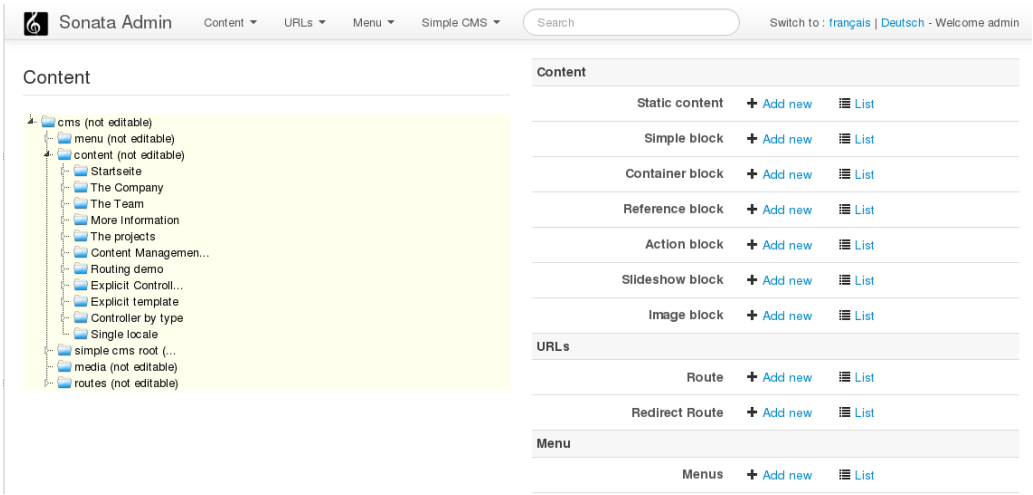
```yaml
# app/config/config.yml
cmf_menu:
    persistence:
        phpcr:
            use_sonata_admin: false
```

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

Listing 4-8

```yaml
# app/config/config.yml
cmf_core:
    persistence:
        phpcr:
            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:



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;

9. https://sonata-project.org/bundles/admin/master/doc/index.html
10. https://sonata-project.org/bundles/doctrine-phpcr-admin/master/doc/index.html

- 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 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.

This tutorial is aimed at people who want to get started with a project based on the Symfony CMF. There are two alternate articles in the cookbook you might want to look at, depending on what you need:

- *Installing and Configuring the CMF Core Bundles* - a guide for adding the CMF to a standard Symfony project.
- *Installing the CMF sandbox* for instructions on how to install a demonstration sandbox. The sandbox contains many examples of what you can do with the CMF.

Preconditions

As Symfony CMF is based on Symfony2, you should make sure you meet the *Requirements for running Symfony2*[1]. Additionally, you need to have *SQLite*[2] 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 *DoctrinePHPCRBundle*.

1. https://symfony.com/doc/current/reference/requirements.html
2. http://www.sqlite.org/

Installation

You can install the Standard Edition in 2 ways:

1) Composer

The easiest way to install Symfony CMF is is using *Composer*[3]. Get it using

Listing 5-1
```
$ curl -sS https://getcomposer.org/installer | php
$ 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
```
$ composer create-project symfony-cmf/standard-edition <path-to-install> "~1.2"
$ cd <path-to-install>
```

The path `<path-to-install>` should either be inside your web server doc root or you need to 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 `var/cache` and `var/logs` directory. In case the final commands end up giving permissions errors, please follow the *guidelines in the Symfony Book*[4] to configure the permissions and then run the `install` command:

Listing 5-3
```
$ composer install
```

2) GIT

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

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

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

Listing 5-5
```
$ composer install
```

To try out things, you can accept the default values for all questions you are asked about the parameters.yml. Revisit that file later when you know more about Jackalope.

Setup

You are almost there. A few more steps need to be done to be ready.

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:

3. https://getcomposer.org/
4. https://symfony.com/doc/current/book/installation.html#book-installation-permissions

Listing 5-6
```
$ php bin/console doctrine:database:create
$ php bin/console doctrine:phpcr:init:dbal --force
$ php bin/console doctrine:phpcr:repository:init
$ php bin/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.

Preparing Assetic

To use the frontend editing in `prod` environment, you need to tell Assetic to dump the assets to the filesystem:

Listing 5-7
```
$ php bin/console --env=prod assetic:dump
```

Configure a Webserver

The project is now ready to be served by your web server. If you have PHP 5.4 installed you can alternatively use the PHP internal web server:

Listing 5-8
```
$ php bin/console server:run
```

And then access the CMF via:

Listing 5-9
```
http://localhost:8000
```

If you run an Apache installation as described in the *Symfony cookbook article on setup*[5], your URL will look like this:

Listing 5-10
```
http://localhost/app_dev.php
```

Adding the `app_dev.php` to the url in your browser is important to actually see the test page. Because the AcmeDemoBundle is only configured to work with the Development Environment. (If you have a look at `AppKernel.php` you can easily spot why)

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`. You also need to uncomment lines in `app/config/config.yml` in section `doctrine.dbal`.

5. https://symfony.com/doc/current/cookbook/configuration/web_server_configuration.html

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 AcmeDemoBundle to help you get started, similar to the AcmeDemoBundle provided by Symfony2 SE. This gives you some demo pages viewable in your browser.

Where are the Controllers?

AcmeDemoBundle 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 biggest chunk of code is the `UnitBlock` that provides a document for an example block.

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.

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 5-11

```
$ php bin/console doctrine:phpcr:node:dump
$ php bin/console doctrine:phpcr:node:dump --props
$ php bin/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 5-12

6. https://symfony.com/doc/current/book/index.html
7. https://symfony.com/doc/current/bundles/DoctrineFixturesBundle/index.html

```
$ php bin/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 one solution can be found in *The Backend - Sonata Admin*. However, if all you want is a simple way to add new pages that you can then edit via the in-line editing, then you can use the SimpleCmsBundle `page` migrator. For example, to add a page called "Testing", creating a file called `app/Resources/data/pages/test.yml` with the following contents:

Listing 5-13
```
label: "Testing"
title: "Testing"
body: |
    <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit.</p>
    <p>Morbi eleifend, ipsum eget facilisis lacinia, lorem dui venenatis quam, at vulputate purus erat sit amet elit.</p>
```

The contents of this file can be loaded into the PHPCR database by calling:

Listing 5-14
```
$ php bin/console doctrine:phpcr:migrator:migrate 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 5-15
```
$ php bin/console doctrine:phpcr:migrator:migrate 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 the 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 differently 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.

The mapping of the `Page` to a template and controller works as explained in the *next chapter*.

Creating a new Page

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

Listing 6-1
```php
// // src/Acme/MainBundle/DataFixtures/PHPCR/LoadSimpleCms.php
namespace Acme\DemoBundle\DataFixtures\PHPCR;

use Doctrine\Common\DataFixtures\FixtureInterface;
use Doctrine\Common\Persistence\ObjectManager;
use Doctrine\ODM\PHPCR\DocumentManager;
use Symfony\Cmf\Bundle\SimpleCmsBundle\Doctrine\Phpcr\Page;

class LoadSimpleCms implements FixtureInterface
{
    /**
     * @param DocumentManager $dm
     */
    public function load(ObjectManager $dm)
    {
        if (!$dm instanceof DocumentManager) {
            $class = get_class($dm);
            throw new \RuntimeException("Fixture requires a PHPCR ODM DocumentManager instance, instance of
'$class' given.");
        }

        $parent = $dm->find(null, '/cms/simple');
        $page = new Page();
        $page->setTitle('About Symfony CMF');
        $page->setLabel('About');
        $page->setBody(...);

        // the tree position defines the URL
        $page->setPosition($parent, 'about');

        $dm->persist($page);
        $dm->flush();
    }
}
```

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
```
/cms/simple/
    about/
    blog/
        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
```php
// // src/Acme/MainBundle/DataFixtures/PHPCR/LoadSimpleCms.php
namespace Acme\DemoBundle\DataFixtures\PHPCR;

use Doctrine\Common\DataFixtures\FixtureInterface;
use Doctrine\Common\Persistence\ObjectManager;
use Doctrine\ODM\PHPCR\DocumentManager;
use Symfony\Cmf\Bundle\SimpleCmsBundle\Doctrine\Phpcr\Page;

class LoadSimpleCms implements FixtureInterface
{
    /**
```

```
12          * @param DocumentManager $dm
13          */
14         public function load(ObjectManager $dm)
15         {
16             if (!$dm instanceof DocumentManager) {
17                 $class = get_class($dm);
18                 throw new \RuntimeException("Fixture requires a PHPCR ODM DocumentManager instance, instance of
19     '$class' given.");
20             }
21
22             $root = $dm->find(null, '/cms/simple');
23
24             $about = new Page();
25             // ... set up about
26             $about->setPosition($root, 'about');
27
28             $dm->persist($about);
29
30             $blog = new Page();
31             // ... set up blog
32             $blog->setPosition($root, 'blog');
33
34             $dm->persist($blog);
35
36             $blogPost = new Page();
37             // ... set up blog post
38             $blogPost->setPosition($blog, 'symfony-cmf-is-great');
39
40             $dm->persist($blogPost);
41
42             $dm->flush();
43         }
    }
```

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 (`/cms/simple` in this case).

The Page at `/cms/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

```yaml
# app/config/config.yml
cmf_routing:
    chain:
        routers_by_id:
            # enable the DynamicRouter with a low priority
            # this way the non dynamic routes take precedence
            # to prevent needless database look ups
            cmf_routing.dynamic_router: 20

            # enable the symfony default router with a higher priority
            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

```yaml
services:
    my_namespace.my_router:
        class: "%my_namespace.my_router_class%"
        tags:
            - { 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. http://api.symfony.com/master/Symfony/Component/Routing/RouterInterface.html

```
# app/config/config.yml
cmf_routing:
    dynamic:
        enabled: true
```

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')`.

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

- 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
```
# app/config/config.yml
cmf_routing:
    dynamic:
        generic_controller: cmf_content.controller:indexAction
        controllers_by_type:
            editable_static: sandbox_main.controller:indexAction
        controllers_by_class:
            Symfony\Cmf\Bundle\ContentBundle\Document\StaticContent: cmf_content.controller::indexAction
        templates_by_class:
            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.

This example uses a controller which is defined as a service. You can also configure a controller by using a fully qualified class name: `CmfContentBundle:Content:index`.

For more information on using controllers as a service read cook book section *How to Define Controllers as Services*[5]

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 provide their own name with the `getRouteKey` method. For normal Symfony routes, the name is only known from their key in the `RouteCollection` collection hashmap. In the CMF, it is possible to use route documents outside of collections, and thus useful to have routes provide their name. The PHPCR routes for example return the repository path when this method is called.

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

5. `https://symfony.com/doc/current/cookbook/controller/service.html`

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
```
# app/config/config.yml
cmf_routing:
    dynamic:
        controllers_by_class:
            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`).

PHPCR-ODM documents are stored in a tree, and their ID is the path in that tree. To match routes, a part of the repository path is used as URL. To avoid mixing routes and other documents, routes are placed under a common root path and that path is removed from the ID to build the URL. The common root path is called "route basepath". The default base path is `/cms/routes`. A new route can be created in PHP code as follows:

Listing 7-6
```
// src/Acme/MainBundle/DataFixtures/PHPCR/LoadRoutingData.php
namespace Acme\DemoBundle\DataFixtures\PHPCR;

use Doctrine\ODM\PHPCR\DocumentManager;
use Doctrine\Common\DataFixtures\FixtureInterface;
use Doctrine\Common\Persistence\ObjectManager;
use Symfony\Cmf\Bundle\RoutingBundle\Doctrine\Phpcr\Route;
use Symfony\Cmf\Bundle\ContentBundle\Doctrine\Phpcr\StaticContent;
use PHPCR\Util\NodeHelper;

class LoadRoutingData implements FixtureInterface
{
    /**
     * @param DocumentManager $dm
     */
    public function load(ObjectManager $dm)
    {
        if (!$dm instanceof DocumentManager) {
            $class = get_class($dm);
            throw new \RuntimeException("Fixture requires a PHPCR ODM DocumentManager instance, instance of
```

```
21  '$class' given.");
22          }
23
24          $session = $dm->getPhpcrSession();
25          NodeHelper::createPath($session, '/cms/routes');
26
27          $route = new Route();
28          $route->setParentDocument($dm->find(null, '/cms/routes'));
29          $route->setName('my-page');
30
31          // link a content to the route
32          $content = new StaticContent();
33          $content->setParentDocument($dm->find(null, '/cms/content'));
34          $content->setName('my-content');
35          $content->setTitle('My Content');
36          $content->setBody('Some Content');
37          $dm->persist($content);
38          $route->setContent($content);
39
40          $dm->persist($route);
41          $dm->flush();
42      }
    }
```

Now the CMF will be able to handle requests for the URL `/my-content`.

As you can see, the code explicitly creates the `/cms/routes` path. The RoutingBundle only creates this path automatically if the Sonata Admin was enabled in the routing configuration using an initializer. Otherwise, it'll assume you do something yourself to create the path (by configuring an initializer or doing it in a fixture like this).

Because you called `setContent` on the route, the controller can expect the `$contentDocument` parameter. You can now configure which controller should handle `StaticContent` as explained above.

The PHPCR-ODM routes support more things, for example route parameters, requirements and defaults. This is explained in the route document section in the RoutingBundle documentation. You can also find route entity documentation and Doctrine ORM integration.

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*[6] component page
- *Handling Multi-Language Documents* for some notes on multilingual routing

6. https://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*

This chapter assumes you are using a Symfony setup with PHPCR-ODM already set up, like the *CMF Standard Edition* or the *CMF sandbox*. See *DoctrinePHPCRBundle* for how to set up PHPCR-ODM in your applications.

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: multi-language support.

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

PHPCR Implementations

In order to let the Doctrine PHPCR-ODM communicate with the PHPCR, a PHPCR implementation is needed. See "*Choosing a PHPCR Implementation*" for an overview of the available implementations.

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

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

class Task
{
    protected $description;

    protected $done = false;
}
```

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

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

use Doctrine\ODM\PHPCR\Mapping\Annotations as PHPCR;

/**
 * @PHPCR\Document()
 */
class Task
{
    /**
     * @PHPCR\Id()
```

```
     */
    protected $id;

    /**
     * @PHPCR\Field(type="string")
     */
    protected $description;

    /**
     * @PHPCR\Field(type="boolean")
     */
    protected $done = false;

    /**
     * @PHPCR\ParentDocument()
     */
    protected $parentDocument;
}
```

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 slugified 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 path + 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*[5]

You may want to implement `Doctrine\ODM\PHPCR\HierarchyInterface` which makes it for example possible to leverage the Sonata Admin Child Extension.

You can also check out Doctrine's Basic Mapping Documentation[6] *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 Doctrine\ODM\PHPCR\Mapping\Annotations as PHPCR;` statement to import 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

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

// ...
use Acme\TaskBundle\Document\Task;
use Symfony\Component\HttpFoundation\Response;

// ...
public function createAction()
{
    $documentManager = $this->get('doctrine_phpcr')->getManager();
```

5. http://docs.doctrine-project.org/projects/doctrine-phpcr-odm/en/latest/reference/basic-mapping.html#basicmapping-identifier-generation-strategies
6. *http://docs.doctrine-project.org/projects/doctrine-phpcr-odm/en/latest/reference/basic-mapping.html*

```
11
12      $rootTask = $documentManager->find(null, '/tasks');
13
14      $task = new Task();
15      $task->setDescription('Finish CMF project');
16      $task->setParentDocument($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:

- **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 = $repository->find('/tasks/'.$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 be name and done
```

```php
$task = $repository->findOneBy(array('name' => 'foo', 'done' => false));

// query for all tasks matching the name, ordered by done
$tasks = $repository->findBy(
    array('name' => 'foo'),
    array('done' => 'ASC')
);
```

If you use the repository class, you can also create a custom repository for a specific document. This helps with "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*[7]" in the core documentation.

You can also query objects by using the Query Builder provided by Doctrine PHPCR-ODM. For more information, read *the QueryBuilder documentation*[8].

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

```php
public function updateAction($name)
{
    $documentManager = $this->get('doctrine_phpcr')->getManager();
    $repository = $documentManager->getRepository('AcmeTaskBundle:Task');
    $task = $repository->find('/tasks/'.$name);

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

    if (!$task->isDone()) {
        $task->setDone(true);
    }

    $documentManager->flush();

    return new Response('[x] '.$task->getDescription());
}
```

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

```php
$documentManager->remove($task);
$documentManager->flush();
```

7. https://symfony.com/doc/current/book/doctrine.html#custom-repository-classes
8. http://docs.doctrine-project.org/projects/doctrine-phpcr-odm/en/latest/reference/query-builder.html

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*[9] and *subscribe to events* that allow you to take different actions as objects go through their persistence lifecycle.

9. `http://docs.doctrine-project.org/projects/doctrine-phpcr-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).

This document class implements three interfaces that enable additional functionality:

- `RouteReferrersInterface` means that the content has associated Routes.
- `PublishTimePeriodInterface` **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`

- `PublishableInterface` **means that the content has a boolean flag,**

  which will be handled by Symfony CMF's core to determine whether or not to display the content from `StaticContent`.

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 page.

Admin Support

The last component needed to handle the included content types is an administration panel. Symfony CMF can optionally support *SonataDoctrinePHPCRAdminBundle*[1], 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 *The Backend - Sonata Admin* 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. `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 *SonataDoctrinePHPCRAdminBundle*[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. https://sonata-project.org/bundles/doctrine-phpcr-admin/master/doc/index.html
3. https://github.com/knplabs/KnpMenuBundle
4. https://symfony.com/doc/master/bundles/KnpMenuBundle/index.html
5. https://symfony.com/doc/master/bundles/KnpMenuBundle/index.html#rendering-menus

```
{{ 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
```
cmf_menu:
    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

Menu factories generate the full `MenuItem` hierarchy from the provided menu node. 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*[6]'s page.

`KnpMenu` already includes a specific factory targeted at the Symfony Routing component 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, the factory 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 is the `MenuNode` document. 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 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

6. `https://github.com/knplabs/KnpMenuBundle`

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].

Admin Support

The MenuBundle also includes the administration panels and respective services needed for integration with the backend admin tool *SonataDoctrinePHPCRAdminBundle*[8].

The included administration panels are automatically available but need to be explicitly put on the dashboard if you want to use them. See *The Backend - Sonata Admin* 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. https://sonata-project.org/bundles/doctrine-phpcr-admin/master/doc/index.html

Chapter 11
Creating a Basic CMS using the RoutingAutoBundle

This series of articles will show you how to create a basic CMS from scratch using the following bundles:

- *RoutingAutoBundle*;
- *DoctrinePHPCRBundle*;
- *MenuBundle*;
- *SonataDoctrinePHPCRAdminBundle*[1].

It is assumed that you have:

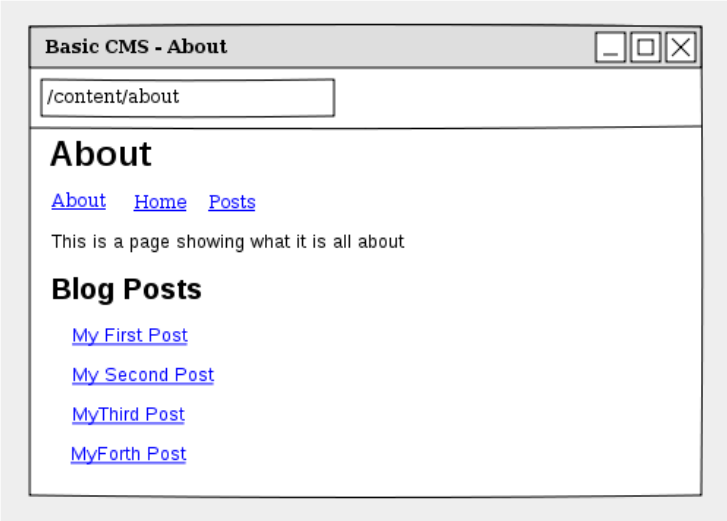
- A working knowledge of the Symfony 2 framework;
- Basic knowledge of PHPCR-ODM.

The CMS will have two types of content:

- **Pages**: HTML content accessed at, for example `/page/home`, `/page/about`, etc.
- **Posts**: Blog posts accessed as `/blog/2012/10/23/my-blog-post`.

The auto routing integration will automatically create and update the routes (effectively the URLs with which you can access the content) for the page and post content documents. In addition each page content document will double up as a menu item.

1. `https://github.com/sonata-project/SonataDoctrinePhpcrAdminBundle`



There exists a bundle called *SimpleCmsBundle* which provides a similar solution to the one proposed in this tutorial. It combines the route, menu and content into a single document and uses a custom router. The approach taken in this tutorial will combine only the menu and content into a single document, the routes will be managed automatically and the native CMF `DynamicRouter` will be used.

Chapter 12
Getting Started

Initializing the Project

First, follow the generic steps in *Create a New Project with PHPCR-ODM* to create a new project using the PHPCR-ODM.

Install Additional Bundles

The complete tutorial requires the following packages:

- *symfony-cmf/routing-auto-bundle*[1];
- *sonata-project/doctrine-phpcr-admin-bundle*[2];
- *doctrine/data-fixtures*[3];
- *symfony-cmf/menu-bundle*[4].

Each part of the tutorial will detail the packages that it requires (if any) in a section titled "installation".

If you intend to complete the entire tutorial you can save some time by adding all of the required packages now:

Listing 12-1
```
$ composer require symfony-cmf/routing-auto-bundle \
    symfony-cmf/menu-bundle \
    sonata-project/doctrine-phpcr-admin-bundle \
    symfony-cmf/tree-browser-bundle \
    doctrine/data-fixtures \
    symfony-cmf/routing-bundle
```

1. https://packagist.org/packages/symfony-cmf/routing-auto-bundle
2. https://packagist.org/packages/sonata-project/doctrine-phpcr-admin-bundle
3. https://packagist.org/packages/doctrine/data-fixtures
4. https://packagist.org/packages/symfony-cmf/menu-bundle

Initialize the Database

If you have followed the main instructions in *DoctrinePHPCRBundle* then you are using the *Doctrine DBAL Jackalope*[5] PHPCR backend with MySQL and you will need to create the database:

Listing 12-2
```
$ php bin/console doctrine:database:create
```

This will create a new database according to the configuration file `parameters.yml`.

The Doctrine DBAL backend needs to be initialized, the following command will create the MySQL schema required to store the hierarchical node content of the PHPCR content repository:

Listing 12-3
```
$ php app/console doctrine:phpcr:init:dbal
```

The *Apache Jackrabbit*[6] implementation is the reference java based backend and does not require such initialization. It does however require the use of Java.

Generate the Bundle

Now you can generate the bundle in which you will write most of your code:

Listing 12-4
```
$ php app/console generate:bundle --namespace=AppBundle --dir=src --format=yml --no-interaction
```

The Documents

You will create two document classes, one for the pages and one for the posts. These two documents share much of the same logic, so you create a `trait` to reduce code duplication:

Listing 12-5
```
// src/AppBundle/Document/ContentTrait.php
namespace AppBundle\Document;

use Doctrine\ODM\PHPCR\Mapping\Annotations as PHPCR;

trait ContentTrait
{
    /**
     * @PHPCR\Id()
     */
    protected $id;

    /**
     * @PHPCR\ParentDocument()
     */
    protected $parent;

    /**
     * @PHPCR\Nodename()
     */
    protected $title;

    /**
     * @PHPCR\Field(type="string", nullable=true)
     */
    protected $content;

    protected $routes;

    public function getId()
```

5. https://github.com/jackalope/jackalope-doctrine-dbal
6. https://jackrabbit.apache.org/jcr/index.html

```
    {
        return $this->id;
    }

    public function getParentDocument()
    {
        return $this->parent;
    }

    public function setParentDocument($parent)
    {
        $this->parent = $parent;
    }

    public function getTitle()
    {
        return $this->title;
    }

    public function setTitle($title)
    {
        $this->title = $title;
    }

    public function getContent()
    {
        return $this->content;
    }

    public function setContent($content)
    {
        $this->content = $content;
    }

    public function getRoutes()
    {
        return $this->routes;
    }
}
```

The `Page` class is therefore nice and simple:

Listing 12-6

```
// src/AppBundle/Document/Page.php
namespace AppBundle\Document;

use Symfony\Cmf\Component\Routing\RouteReferrersReadInterface;

use Doctrine\ODM\PHPCR\Mapping\Annotations as PHPCR;

/**
 * @PHPCR\Document(referenceable=true)
 */
class Page implements RouteReferrersReadInterface
{
    use ContentTrait;
}
```

Note that the page document should be `referenceable`. This will enable other documents to hold a reference to the page. The `Post` class will also be referenceable and in addition will automatically set the date using the *pre persist lifecycle event*[7] if it has not been explicitly set previously:

Listing 12-7

```
// src/AppBundle/Document/Post.php
namespace AppBundle\Document;

use Doctrine\ODM\PHPCR\Mapping\Annotations as PHPCR;
```

7. http://docs.doctrine-project.org/projects/doctrine-phpcr-odm/en/latest/reference/events.html#lifecycle-callbacks

```php
use Symfony\Cmf\Component\Routing\RouteReferrersReadInterface;

/**
 * @PHPCR\Document(referenceable=true)
 */
class Post implements RouteReferrersReadInterface
{
    use ContentTrait;

    /**
     * @PHPCR\Date()
     */
    protected $date;

    /**
     * @PHPCR\PrePersist()
     */
    public function updateDate()
    {
        if (!$this->date) {
            $this->date = new \DateTime();
        }
    }

    public function getDate()
    {
        return $this->date;
    }

    public function setDate(\DateTime $date)
    {
        $this->date = $date;
    }
}
```

Both the `Post` and `Page` classes implement the `RouteReferrersReadInterface`. This interface enables the DynamicRouter to generate URLs from instances of these classes. (for example with `{{ path(content) }}` in Twig).

Repository Initializer

Repository initializers enable you to establish and maintain PHPCR nodes required by your application, for example you will need the paths `/cms/pages`, `/cms/posts` and `/cms/routes`. The `GenericInitializer` class can be used easily initialize a list of paths. Add the following to your service container configuration:

Listing 12-8

```yaml
# src/AppBundle/Resources/config/services.yml
services:
    app.phpcr.initializer:
        class: Doctrine\Bundle\PHPCRBundle\Initializer\GenericInitializer
        arguments:
            - My custom initializer
            - ["/cms/pages", "/cms/posts", "/cms/routes"]
        tags:
            - { name: doctrine_phpcr.initializer }
```

The initializers operate at the PHPCR level, not the PHPCR-ODM level - this means that you are dealing with nodes and not documents. You do not have to understand these details right now. To learn more about PHPCR read *Choosing a Storage Layer*.

The initializers will be executed automatically when you load your data fixtures (as detailed in the next section) or alternatively you can execute them manually using the following command:

Listing 12-9
```
$ php app/console doctrine:phpcr:repository:init
```

This command is *idempotent*[8], which means that it is safe to run it multiple times, even when you have data in your repository. Note however that it is the responsibility of the initializer to respect idempotency!

You can check to see that the repository has been initialized by dumping the content repository:

Listing 12-10
```
$ php app/console doctrine:phpcr:node:dump
```

Create Data Fixtures

You can use the doctrine data fixtures library to define some initial data for your CMS.

Ensure that you have the following package installed:

Listing 12-11
```
{
    ...
    require: {
        ...
        "doctrine/data-fixtures": "1.0.*"
    },
    ...
}
```

Create a page for your CMS:

Listing 12-12
```
// src/AppBundle/DataFixtures/PHPCR/LoadPageData.php
namespace AppBundle\DataFixtures\PHPCR;

use AppBundle\Document\Page;
use Doctrine\Common\DataFixtures\FixtureInterface;
use Doctrine\Common\Persistence\ObjectManager;
use Doctrine\ODM\PHPCR\DocumentManager;

class LoadPageData implements FixtureInterface
{
    public function load(ObjectManager $dm)
    {
        if (!$dm instanceof DocumentManager) {
            $class = get_class($dm);
            throw new \RuntimeException("Fixture requires a PHPCR ODM DocumentManager instance, instance of
'$class' given.");
        }

        $parent = $dm->find(null, '/cms/pages');

        $page = new Page();
        $page->setTitle('Home');
        $page->setParentDocument($parent);
        $page->setContent(<<<HERE
Welcome to the homepage of this really basic CMS.
HERE
        );

        $dm->persist($page);
        $dm->flush();
    }
}
```

and add some posts:

8. https://en.wiktionary.org/wiki/idempotent

Listing 12-13

```php
// src/AppBundle/DataFixtures/PHPCR/LoadPostData.php
namespace AppBundle\DataFixtures\PHPCR;

use Doctrine\Common\DataFixtures\FixtureInterface;
use Doctrine\Common\Persistence\ObjectManager;
use Doctrine\ODM\PHPCR\DocumentManager;
use AppBundle\Document\Post;

class LoadPostData implements FixtureInterface
{
    public function load(ObjectManager $dm)
    {
        if (!$dm instanceof DocumentManager) {
            $class = get_class($dm);
            throw new \RuntimeException("Fixture requires a PHPCR ODM DocumentManager instance, instance of
'$class' given.");
        }

        $parent = $dm->find(null, '/cms/posts');

        foreach (array('First', 'Second', 'Third', 'Fourth') as $title) {
            $post = new Post();
            $post->setTitle(sprintf('My %s Post', $title));
            $post->setParentDocument($parent);
            $post->setContent(<<<HERE
This is the content of my post.
HERE
            );

            $dm->persist($post);
        }

        $dm->flush();
    }
}
```

Then load the fixtures:

Listing 12-14

```
$ php app/console doctrine:phpcr:fixtures:load
```

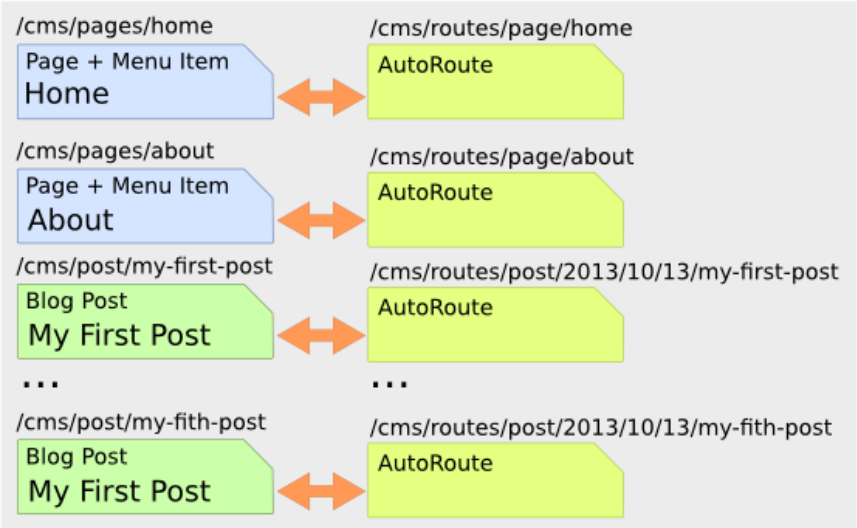
You should now have some data in your content repository.

Chapter 13
Routing and Automatic Routing

The routes (URIs) to your content will be automatically created and updated using the RoutingAutoBundle. This bundle uses a configuration language to specify automatic creation of routes, which can be a bit hard to grasp the first time you see it.

For a full explanation refer to the *RoutingAutoBundle*.

In summary, you will configure the auto routing system to create a new auto routing document in the routing tree for every post or content created. The new route will be linked back to the target content:



The paths above represent the path in the PHPCR-ODM document tree. In the next section you will define `/cms/routes` as the base path for routes, and subsequently the contents will be available at the following URIs:

- **Home**: `http://localhost:8000/page/home`
- **About**: `http://localhost:8000/page/about`
- etc.

Installation

Ensure that you installed the RoutingAutoBundle package as detailed in the Install Additional Bundles section.

Enable the routing bundles to your kernel:

Listing 13-1
```php
class AppKernel extends Kernel
{
    public function registerBundles()
    {
        $bundles = array(
            // ...
            new Symfony\Cmf\Bundle\RoutingBundle\CmfRoutingBundle(),
            new Symfony\Cmf\Bundle\RoutingAutoBundle\CmfRoutingAutoBundle(),
        );

        // ...
    }
}
```

The *symfony-cmf/routing-bundle* package is installed automatically as *symfony-cmf/routing-auto-bundle* depends on it.

Enable the Dynamic Router

The RoutingAutoBundle uses the CMF *RoutingBundle*[1] which enables routes to be provided from a database (in addition to being provided from the routing configuration files as in core Symfony 2).

Add the following to your application configuration:

Listing 13-2
```yaml
# /app/config/config.yml
cmf_routing:
    chain:
        routers_by_id:
            cmf_routing.dynamic_router: 200
            router.default: 100
    dynamic:
        enabled: true
        persistence:
            phpcr: true
```

This will:

1. Cause the default Symfony router to be replaced by the chain router. The chain router enables you to have multiple routers in your application. You add the dynamic router (which can retrieve routes from the database) and the default Symfony router (which retrieves routes from configuration files). The number indicates the order of precedence - the router with the highest number will be called first;
2. Configure the **dynamic** router which you have added to the router chain. You specify that it should use the PHPCR backend and that the *root* route can be found at `/cms/routes`.

Auto Routing Configuration

First you need to configure the auto routing bundle:

1. https://symfony.com/doc/master/cmf/bundles/routing/index.html

Listing 13-3
```yaml
# app/config/config.yml
cmf_routing_auto:
    persistence:
        phpcr:
            enabled: true
```

The above configures the RoutingAutoBundle to work with PHPCR-ODM.

You can now proceed to mapping your documents, create the following in your *bundles* configuration directory:

Listing 13-4
```yaml
# src/AppBundle/Resources/config/cmf_routing_auto.yml
AppBundle\Document\Page:
    uri_schema: /page/{title}
    token_providers:
        title: [content_method, { method: getTitle }]

AppBundle\Document\Post:
    uri_schema: /post/{date}/{title}
    token_providers:
        date: [content_datetime, { method: getDate }]
        title: [content_method, { method: getTitle }]
```

RoutingAutoBundle mapping bundles are registered automatically when they are named as above, you may alternatively explicitly declare from where the mappings should be loaded, see the *RoutingAutoBundle* documentation for more information.

This will configure the routing auto system to automatically create and update route documents for both the `Page` and `Post` documents.

In summary, for each class:

- We defined a `uri_schema` which defines the form of the URI which will be generated. * Within the schema you place `{tokens}` - placeholders for values provided by...
- Token providers provide values which will be substituted into the URI. Here you use two different providers - `content_datetime` and `content_method`. Both will return dynamic values from the subject object itself.

Now reload the fixtures:

Listing 13-5
```
$ php bin/console doctrine:phpcr:fixtures:load
```

Have a look at what you have:

Listing 13-6
```
$ php bin/console doctrine:phpcr:node:dump
ROOT:
  cms:
    pages:
      Home:
    routes:
      page:
        home:
      post:
        2013:
          10:
            12:
              my-first-post:
              my-second-post:
              my-third-post:
              my-fourth-post:
    posts:
      My First Post:
      My Second Post:
```

```
20         My Third Post:
21         My Fourth Post:
```

The routes have been automatically created!

Chapter 14
The Backend - Sonata Admin

In this chapter you will build an administration interface with the help of the *SonataDoctrinePHPCRAdminBundle*[1].

First, follow the *Sonata installation guide*[2], and then the *instructions to set up the SonataPhpcrAdminIntegrationBundle*.

Configuration

Now start a local webserver:

Listing 14-1
```
$ php bin/console server:run
```

That works? Great, now have a look at *http://127.0.0.1:8000/admin/dashboard*

No translations? Uncomment the translator in the configuration file:

Listing 14-2
```
# app/config/config.yml

# ...
framework:
    # ...
    translator:      { fallback: "%locale%" }
```

See Sonata PHPCR-ODM Admin for more information on Sonata Admin and multi-language.

When looking at the admin dashboard, you will notice that there is an entry to administrate Routes. The administration class of the RoutingBundle has been automatically registered. However, you do not need this in your application as the routes are managed by the RoutingAutoBundle and not the administrator. You can disable the RoutingBundle admin:

1. https://sonata-project.org/bundles/doctrine-phpcr-admin/master/doc/index.html
2. https://sonata-project.org/bundles/doctrine-phpcr-admin/1-x/doc/reference/installation.html

Listing 14-3
```yaml
# app/config/config.yml
cmf_routing:
    # ...
    dynamic:
        # ...
        persistence:
            phpcr:
                # ...
                use_sonata_admin: false
```

All Sonata Admin aware CMF bundles have such a configuration option and it prevents the admin class (or classes) from being registered.

Creating the Admin Classes

Create the following admin classes, first for the `Page` document:

Listing 14-4
```php
// src/AppBundle/Admin/PageAdmin.php
namespace AppBundle\Admin;

use Sonata\DoctrinePHPCRAdminBundle\Admin\Admin;
use Sonata\AdminBundle\Datagrid\DatagridMapper;
use Sonata\AdminBundle\Datagrid\ListMapper;
use Sonata\AdminBundle\Form\FormMapper;

class PageAdmin extends Admin
{
    protected function configureListFields(ListMapper $listMapper)
    {
        $listMapper
            ->addIdentifier('title', 'text')
        ;
    }

    protected function configureFormFields(FormMapper $formMapper)
    {
        $formMapper
            ->with('form.group_general')
                ->add('title', 'text')
                ->add('content', 'textarea')
            ->end()
        ;
    }

    public function prePersist($document)
    {
        $parent = $this->getModelManager()->find(null, '/cms/pages');
        $document->setParentDocument($parent);
    }

    protected function configureDatagridFilters(DatagridMapper $datagridMapper)
    {
        $datagridMapper->add('title', 'doctrine_phpcr_string');
    }

    public function getExportFormats()
    {
        return array();
    }
}
```

and then for the `Post` document - as you have already seen this document is almost identical to the `Page` document, so extend the `PageAdmin` class to avoid code duplication:

Listing 14-5

```php
// src/AppBundle/Admin/PostAdmin.php
namespace AppBundle\Admin;

use Sonata\DoctrinePHPCRAdminBundle\Admin\Admin;
use Sonata\AdminBundle\Datagrid\DatagridMapper;
use Sonata\AdminBundle\Datagrid\ListMapper;
use Sonata\AdminBundle\Form\FormMapper;

class PostAdmin extends PageAdmin
{
    protected function configureFormFields(FormMapper $formMapper)
    {
        parent::configureFormFields($formMapper);

        $formMapper
            ->with('form.group_general')
                ->add('date', 'date')
            ->end()
        ;
    }
}
```

In the `prePersist` method of the `PageAdmin` you hard-code the parent path. You may want to modify this behavior to enable pages to be structured (for example to have nested menus).

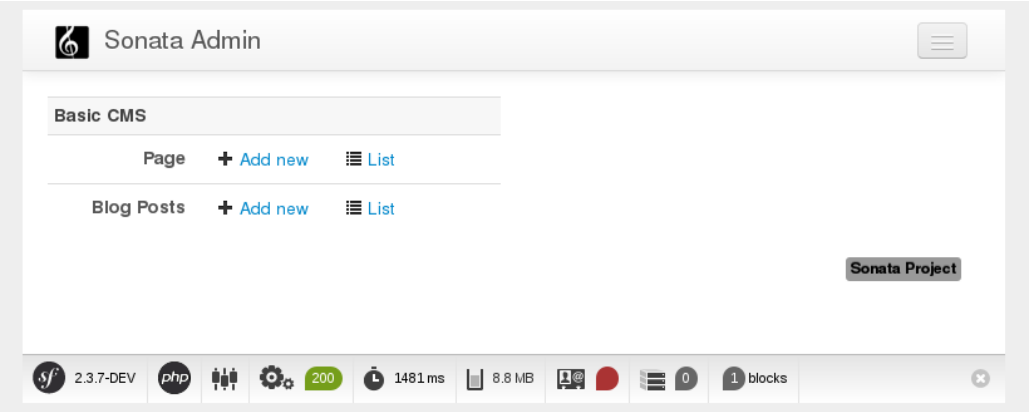
Now you just need to register these classes in the dependency injection container configuration:

Listing 14-6

```yaml
# src/AppBundle/Resources/config/services.yml
services:
    app.admin.page:
        class: AppBundle\Admin\PageAdmin
        arguments:
            - ''
            - AppBundle\Document\Page
            - 'SonataAdminBundle:CRUD'
        tags:
            - { name: sonata.admin, manager_type: doctrine_phpcr, group: 'Basic CMS', label: Page }
        calls:
            - [setRouteBuilder, ['@sonata.admin.route.path_info_slashes']]
    app.admin.post:
        class: AppBundle\Admin\PostAdmin
        arguments:
            - ''
            - AppBundle\Document\Post
            - 'SonataAdminBundle:CRUD'
        tags:
            - { name: sonata.admin, manager_type: doctrine_phpcr, group: 'Basic CMS', label: 'Blog Posts' }
        calls:
            - [setRouteBuilder, ['@sonata.admin.route.path_info_slashes']]
```

In the XML version of the above configuration you specify `manager_type` (with an underscore). This should be `manager-type` (with a hyphen) and is fixed in Symfony version 2.4.

Check it out at *http://localhost:8000/admin/dashboard*



Configure the Admin Tree on the Dashboard

Sonata admin provides a useful tree view of your whole content. You can click items on the tree to edit them, right-click to delete them or add children and drag and drop to reorganize your content.

Enable the CmfTreeBundle and the FOSJsRoutingBundle in your kernel:

Listing 14-7
```php
// app/AppKernel.php
class AppKernel extends Kernel
{
    // ...

    public function registerBundles()
    {
        $bundles = array(
            // ...
            new FOS\JsRoutingBundle\FOSJsRoutingBundle(),
            new Symfony\Cmf\Bundle\TreeBrowserBundle\CmfTreeBrowserBundle(),
        );

        // ...
    }
}
```

Now publish your assets again:

Listing 14-8
```
$ php bin/console assets:install --symlink web/
```

Routes used by the tree in the frontend are handled by the FOSJsRoutingBundle. The relevant routes are tagged with the `expose` flag, they are available automatically. However, you need to load the routes of the TreeBundle and the FOSJsRoutingBundle:

Listing 14-9
```yaml
# app/config/routing.yml
cmf_tree:
    resource: .
    type: 'cmf_tree'

fos_js_routing:
    resource: "@FOSJsRoutingBundle/Resources/config/routing/routing.xml"
```

Add the tree block to the `sonata_block` configuration and tell sonata admin to display the block (be careful to *add* to the existing configuration and not to create another section!):

Listing 14-10
```yaml
# app/config/config.yml

# ...
sonata_block:
```

```
    blocks:
        # ...
        sonata_admin_doctrine_phpcr.tree_block:
            settings:
                id: '/cms'
            contexts: [admin]

sonata_admin:
    dashboard:
        blocks:
            - { position: left, type: sonata_admin_doctrine_phpcr.tree_block }
            - { position: right, type: sonata.admin.block.admin_list }
```

To see your documents on the tree in the admin dashboard tree, you need to tell sonata about them:

Listing 14-11

```
sonata_doctrine_phpcr_admin:
    document_tree_defaults: [locale]
    document_tree:
        Doctrine\ODM\PHPCR\Document\Generic:
            valid_children:
                - all
        AppBundle\Document\Page:
            valid_children:
                - AppBundle\Document\Post
        AppBundle\Document\Post:
            valid_children: []
        # ...
```

To have a document show up in the tree, it needs its own entry. You can allow all document types underneath it by having the `all` child. But if you explicitly list allowed children, the right click context menu will propose only those documents. This makes it easier for your users to not make mistakes.

Chapter 15
Controllers and Templates

Make your content route aware

In the *Getting Started* section, you defined your *Post* and *Page* documents as implementing the `RoutesReferrersReadInterface`. This interface enables the routing system to retrieve routes which refer to the object implementing this interface, and this enables the system to generate a URL (for example when you use `{{ path(mydocument) }}` in Twig).

Earlier you did not have the RoutingBundle installed, so you could not add the mapping.

Map the `$routes` property to contain a collection of all the routes which refer to this document:

Listing 15-1
```php
// src/AppBundle/Document/ContentTrait.php

// ...
trait ContentTrait
{
    // ...

    /**
     * @PHPCR\Referrers(
     *     referringDocument="Symfony\Cmf\Bundle\RoutingBundle\Doctrine\Phpcr\Route",
     *     referencedBy="content"
     * )
     */
    protected $routes;

    // ...
}
```

And clear your cache:

Listing 15-2
```
$ php bin/console cache:clear
```

Now you can call the method `getRoutes` on either `Page` or `Post` and retrieve all the routes which refer to that document ... pretty cool!

Route Requests to a Controller

Go to the URL *http://127.0.0.1:8000/page/home* in your browser - this should be your page, but it says that it cannot find a controller. In other words it has found the *route referencing the page* for your page but Symfony does not know what to do with it.

You can map a default controller for all instances of `Page`:

Listing 15-3
```yaml
# app/config/config.yml
cmf_routing:
    dynamic:
        # ...
        controllers_by_class:
            AppBundle\Document\Page: AppBundle\Controller\DefaultController::pageAction
```

This will cause requests to be forwarded to this controller when the route which matches the incoming request is provided by the dynamic router **and** the content document that that route references is of class `AppBundle\Document\Page`.

Now create the action in the default controller - you can pass the `Page` object and all the `Posts` to the view:

Listing 15-4
```php
// src/AppBundle/Controller/DefaultController.php

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

class DefaultController extends Controller
{
    // ...

    /**
     * @Template()
     */
    public function pageAction($contentDocument)
    {
        $dm = $this->get('doctrine_phpcr')->getManager();
        $posts = $dm->getRepository('AppBundle:Post')->findAll();

        return array(
            'page'  => $contentDocument,
            'posts' => $posts,
        );
    }
}
```

The `Page` object is passed automatically as `$contentDocument`.

Add a corresponding template (note that this works because you use the `@Template` annotation):

Listing 15-5
```twig
{# src/AppBundle/Resources/views/Default/page.html.twig #}
<h1>{{ page.title }}</h1>
<p>{{ page.content|raw }}</p>
<h2>Our Blog Posts</h2>
<ul>
    {% for post in posts %}
        <li><a href="{{ path(post) }}">{{ post.title }}</a></li>
    {% endfor %}
</ul>
```

Now have another look at: *http://localhost:8000/page/home*

Notice what is happening with the post object and the `path` function - you pass the `Post` object and the `path` function will pass the object to the router and because it implements the `RouteReferrersReadInterface` the `DynamicRouter` will be able to generate the URL for the post.

Click on a `Post` and you will have the same error that you had before when viewing the page at `/home` and you can resolve it in the same way.

If you have different content classes with different templates, but you don't need specific controller logic, you can configure `templates_by_class` instead of `controllers_by_class` to let the default controller render a specific template. See Configuring the Controller for a Route for more information on this.

Chapter 16
Creating a Menu

In this section you will modify your application so that `Page` documents act as menu nodes. The root page document can then be rendered using the Twig helper of the *KnpMenuBundle*[1].

Installation

Ensure that you installed the `symfony-cmf/menu-bundle` package as detailed in the Install Additional Bundles section.

Add the CMF *MenuBundle* and its dependency, *CoreBundle*, to your kernel:

Listing 16-1
```php
// app/AppKernel.php
class AppKernel extends Kernel
{
    public function registerBundles()
    {
        $bundles = array(
            // ...
            new Symfony\Cmf\Bundle\CoreBundle\CmfCoreBundle(),
            new Symfony\Cmf\Bundle\MenuBundle\CmfMenuBundle(),
        );

        // ...
    }
}
```

The KnpMenuBundle is also required but was already included in the *The Backend - Sonata Admin* chapter. If you skipped that chapter be sure to add this bundle now.

1. `https://github.com/KnpLabs/KnpMenuBundle`

Modify the Page Document

The menu document has to implement the `Knp\Menu\NodeInterface` provided by the KnpMenuBundle. Modify the Page document so that it implements this interface:

Listing 16-2
```php
// src/AppBundle/Document/Page.php
namespace AppBundle\Document;

// ...
use Knp\Menu\NodeInterface;

use Doctrine\ODM\PHPCR\Mapping\Annotations as PHPCR;

class Page implements RouteReferrersReadInterface, NodeInterface
```

Now add the following to the document to fulfill the contract:

Listing 16-3
```php
// src/AppBundle/Document/Page.php

// ...
class Page implements RouteReferrersReadInterface, NodeInterface
{
    // ...

    /**
     * @PHPCR\Children()
     */
    protected $children;

    public function getName()
    {
        return $this->title;
    }

    public function getChildren()
    {
        return $this->children;
    }

    public function getOptions()
    {
        return array(
            'label' => $this->title,
            'content' => $this,

            'attributes'         => array(),
            'childrenAttributes' => array(),
            'displayChildren'    => true,
            'linkAttributes'     => array(),
            'labelAttributes'    => array(),
        );
    }
}
```

In a typical CMF application, there are two `NodeInterface` which have nothing to do with each other. The interface we use here is from KnpMenuBundle and describes menu tree nodes. The other interface is from the PHP content repository and describes content repository tree nodes.

Menus are hierarchical, PHPCR-ODM is also hierarchical and so lends itself well to this use case.

Here you add an additional mapping, `@Children`, which will cause PHPCR-ODM to populate the annotated property instance `$children` with the child documents of this document.

The options are the options used by KnpMenu system when rendering the menu. The menu URL is inferred from the `content` option (note that you added the `RouteReferrersReadInterface` to `Page` earlier).

The attributes apply to the HTML elements. See the *KnpMenu*[2] documentation for more information.

Modify the Data Fixtures

The menu system expects to be able to find a root item which contains the first level of child items. Modify your fixtures to declare a root element to which you will add the existing `Home` page and an additional `About` page:

Listing 16-4

```php
// src/AppBundle/DataFixtures/PHPCR/LoadPageData.php
namespace AppBundle\DataFixtures\PHPCR;

use Doctrine\Common\DataFixtures\FixtureInterface;
use Doctrine\Common\Persistence\ObjectManager;
use Doctrine\ODM\PHPCR\DocumentManager;
use AppBundle\Document\Page;

class LoadPageData implements FixtureInterface
{
    public function load(ObjectManager $dm)
    {
        if (!$dm instanceof DocumentManager) {
            $class = get_class($dm);
            throw new \RuntimeException("Fixture requires a PHPCR ODM DocumentManager instance, instance of
'$class' given.");
        }

        $parent = $dm->find(null, '/cms/pages');

        $rootPage = new Page();
        $rootPage->setTitle('main');
        $rootPage->setParentDocument($parent);
        $dm->persist($rootPage);

        $page = new Page();
        $page->setTitle('Home');
        $page->setParentDocument($rootPage);
        $page->setContent(<<<HERE
Welcome to the homepage of this really basic CMS.
HERE
        );
        $dm->persist($page);

        $page = new Page();
        $page->setTitle('About');
        $page->setParentDocument($rootPage);
        $page->setContent(<<<HERE
This page explains what its all about.
HERE
        );
        $dm->persist($page);

        $dm->flush();
    }
}
```

Load the fixtures again:

Listing 16-5

```
$ php bin/console doctrine:phpcr:fixtures:load
```

2. https://github.com/KnpLabs/KnpMenu

Register the Menu Provider

Now you can register the `PhpcrMenuProvider` from the menu bundle in the service container configuration:

Listing 16-6
```yaml
# src/AppBundle/Resources/config/services.yml
services:
    app.menu_provider:
        class: Symfony\Cmf\Bundle\MenuBundle\Provider\PhpcrMenuProvider
        arguments:
            - '@cmf_menu.loader.node'
            - '@doctrine_phpcr'
            - /cms/pages
        tags:
            - { name: knp_menu.provider }
```

New in version 2.0: The first argument of the `PhpcrMenuProvider` class was changed in CmfMenuBundle 2.0. You had to inject the `cmf_menu.factory` service prior to version 2.0.

and enable the Twig rendering functionality of the KnpMenuBundle:

Listing 16-7
```yaml
# app/config/config.yml
knp_menu:
    twig: true
```

and finally you can render the menu!

Listing 16-8
```
{# src/AppBundle/Resources/views/Default/page.html.twig #}

{# ... #}
{{ knp_menu_render('main') }}
```

Note that `main` refers to the name of the root page you added in the data fixtures.

Chapter 17
The Site Document and the Homepage

All of your content should now be available at various URLs but your homepage (*http://localhost:8000*) still shows the default Symfony Standard Edition index page.

In this section you will add a side menu to Sonata Admin which allows the user to mark a `Page` to act as the homepage of your CMS.

This is just one of many strategies for routing the homepage. For example, another option would be put a `RedirectRoute` document at `/cms/routes`.

Storing the Data

You need a document which can store data about your CMS - this will be known as the site document and it will contain a reference to the `Page` document which will act as the homepage.

Create the site document:

Listing 17-1
```php
// src/AppBundle/Document/Site.php
namespace AppBundle\Document;

use Doctrine\ODM\PHPCR\Mapping\Annotations as PHPCR;

/**
 * @PHPCR\Document()
 */
class Site
{
    /**
     * @PHPCR\Id()
     */
    protected $id;

    /**
     * @PHPCR\ReferenceOne(targetDocument="AppBundle\Document\Page")
     */
    protected $homepage;

```

```
    public function getHomepage()
    {
        return $this->homepage;
    }

    public function setHomepage($homepage)
    {
        $this->homepage = $homepage;
    }

    public function setId($id)
    {
        $this->id = $id;
    }
}
```

Initializing the Site Document

Where does the `Site` document belong? The document hierarchy currently looks like this:

Listing 17-2

```
ROOT/
    cms/
       pages/
       routes/
       posts/
```

There is one `cms` node, and this node contains all the children nodes of our site. This node is therefore the logical position of your `Site` document.

Earlier, you used the `GenericInitializer` to initialize the base paths of our project, including the `cms` node. The nodes created by the `GenericInitializer` have no PHPCR-ODM mapping however.

You can *replace* the `GenericInitializer` with a custom initializer which will create the necessary paths **and** assign a document class to the `cms` node:

Listing 17-3

```
// src/AppBundle/Initializer/SiteInitializer.php
namespace AppBundle\Initializer;

use Doctrine\Bundle\PHPCRBundle\Initializer\InitializerInterface;
use PHPCR\Util\NodeHelper;
use Doctrine\Bundle\PHPCRBundle\ManagerRegistry;
use AppBundle\Document\Site;

class SiteInitializer implements InitializerInterface
{
    private $basePath;

    public function __construct($basePath = '/cms')
    {
        $this->basePath = $basePath;
    }

    public function init(ManagerRegistry $registry)
    {
        $dm = $registry->getManager();
        if ($dm->find(null, $this->basePath)) {
            return;
        }

        $site = new Site();
        $site->setId($this->basePath);
        $dm->persist($site);
        $dm->flush();

```

```
30          $session = $registry->getConnection();
31
32          // create the 'cms', 'pages', and 'posts' nodes
33          NodeHelper::createPath($session, $this->basePath . '/pages');
34          NodeHelper::createPath($session, $this->basePath . '/posts');
35          NodeHelper::createPath($session, $this->basePath . '/routes');
36
37          $session->save();
38      }
39
40      public function getName()
41      {
42          return 'My site initializer';
43      }
44  }
```

New in version 1.1: Since version 1.1, the `init` method receives the `ManagerRegistry` rather than the PHPCR `SessionInterface`. This allows the creation of documents in initializers. With 1.0, you would need to manually set the `phpcr:class` property to the right value.

Now:

1. *Remove* the initializer service that you created in the *Getting Started* chapter (`app.phpcr.initializer`).
2. Register your new site initializer:

Listing 17-4
```
1  # src/AppBundle/Resources/config/services.yml
2  services:
3      # ...
4      app.phpcr.initializer.site:
5          class: AppBundle\Initializer\SiteInitializer
6          tags:
7              - { name: doctrine_phpcr.initializer, priority: 50 }
```

You may have noticed that you have set the priority of the initializer. Initializers with high priorities will be called before initializers with lower priorities. Here it is necessary to increase the priority of your listener to prevent other initializers creating the *cms* node first.

Now empty your repository, reinitialize it and reload your fixtures:

Listing 17-5
```
1  $ php bin/console doctrine:phpcr:node:remove /cms
2  $ php bin/console doctrine:phpcr:repository:init
3  $ php bin/console doctrine:phpcr:fixtures:load
```

and verify that the `cms` node has been created correctly, using the `doctrine:phpcr:node:dump` command with the `props` flag:

Listing 17-6
```
1  $ php bin/console doctrine:phpcr:node:dump --props
2  ROOT:
3    cms:
4      - jcr:primaryType = nt:unstructured
5      - phpcr:class = AppBundle\Document\Site
6      ...
```

Why use an initializer instead of a data fixture? In this instance, the site object is a constant for your application. There is only one site object, new sites will not be created and the existing site document will not be removed. DataFixtures are intended to provide sample data, not data which is integral to the functioning of your site.

Instead of *replacing* the `GenericInitializer` you could simply add another initializer which is run first and create the `/cms` document with the right class. The drawback then is that there are two places where initialization choices take place - do whatever you prefer.

Reconfigure the Admin Tree

If you look at your admin interface now, you will notice that the tree has gone!

You need to tell the admin tree about the new `Site` document which is now the root of your websites content tree:

Listing 17-7
```yaml
sonata_doctrine_phpcr_admin:
    # ...
    document_tree:
        # ...
        AppBundle\Document\Site:
            valid_children:
                - all
```

If you check your admin interface you will see that the `Site` document is now being displayed, however it has no children. You need to map the children on the `Site` document, modify it as follows:

Listing 17-8
```php
// src/AppBundle/Document/Site.php

// ...

/**
 * @PHPCR\Document()
 */
class Site
{
    /**
     * @PHPCR\Children()
     */
    protected $children;

    // ...

    public function getChildren()
    {
        return $this->children;
    }
}
```

The tree should now again show your website structure.

Create the Make Homepage Button

You will need a way to allow the administrator of your site to select which page should act as the homepage. You will modify the `PageAdmin` class so that a "Make Homepage" button will appear when editing a page. You will achieve this by adding a "side menu".

Firstly though you will need to create an action which will do the work of making a given page the homepage. Add the following to the existing `DefaultController`:

Listing 17-9
```php
// src/AppBundle/Controller/DefaultController.php

use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Method;
```

```php

// ...
class DefaultController extends Controller
{
    // ...

    /**
     * @Route(
     *   name="make_homepage",
     *   pattern="/admin/make_homepage/{id}",
     *   requirements={"id": ".+"}
     * )
     * @Method({"GET"})
     */
    public function makeHomepageAction($id)
    {
        $dm = $this->get('doctrine_phpcr')->getManager();

        $site = $dm->find(null, '/cms');
        if (!$site) {
            throw $this->createNotFoundException('Could not find /cms document!');
        }

        $page = $dm->find(null, $id);

        $site->setHomepage($page);
        $dm->persist($page);
        $dm->flush();

        return $this->redirect($this->generateUrl('admin_app_page_edit', array(
            'id' => $page->getId()
        )));
    }
}
```

You have specified a special requirement for the `id` parameter of the route, this is because by default routes will not allow forward slashes "/" in route parameters and our "id" is a path.

Now modify the `PageAdmin` class to add the button in a side-menu:

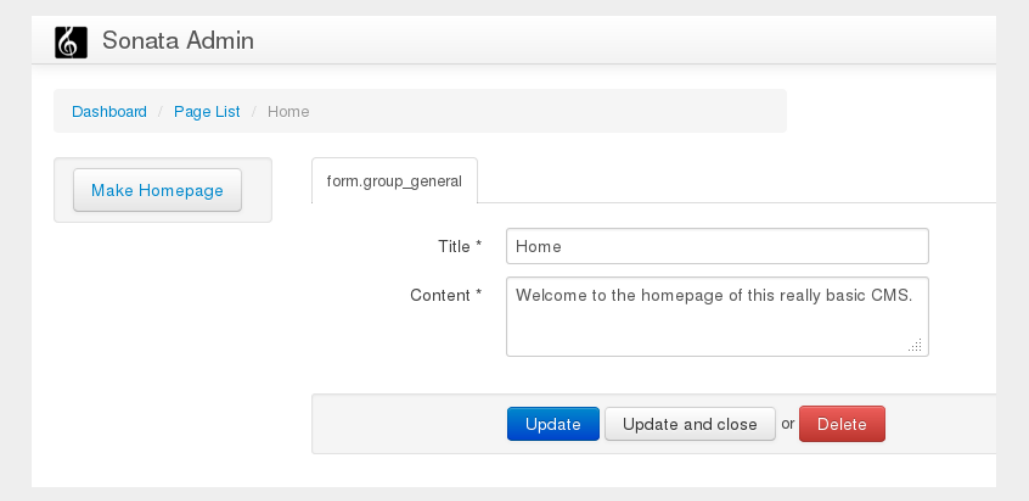
Listing 17-10

```php
// src/AppBundle/Admin/PageAdmin

// ...
use Knp\Menu\ItemInterface;
use Sonata\AdminBundle\Admin\AdminInterface;

class PageAdmin extends Admin
{
    // ...
    protected function configureSideMenu(ItemInterface $menu, $action, AdminInterface $childAdmin = null)
    {
        if ('edit' !== $action) {
            return;
        }

        $page = $this->getSubject();

        $menu->addChild('make-homepage', array(
            'label' => 'Make Homepage',
            'attributes' => array('class' => 'btn'),
            'route' => 'make_homepage',
            'routeParameters' => array(
                'id' => $page->getId(),
            ),
        ));
    }
}
```

The two arguments which concern you here are:

- `$menu`: This will be a root menu item to which you can add new menu items (this is the same menu API you worked with earlier);
- `$action`: Indicates which kind of page is being configured;

If the action is not `edit` it returns early and no side-menu is created. Now that it knows the edit page is requested, it retrieves the *subject* from the admin class which is the `Page` currently being edited, it then adds a menu item to the menu.



Routing the Homepage

Now that you have enabled the administrator to designate a page to be used as a homepage you need to actually make the CMS use this information to render the designated page.

This is easily accomplished by modifying the `indexAction` action of the `DefaultController` to forward requests matching the route pattern `/` to the page action:

Listing 17-11

```
// src/AppBundle/Controller/DefaultController.php

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

class DefaultController extends Controller
{
    // ...

    /**
     * Load the site definition and redirect to the default page.
     *
     * @Route("/")
     */
    public function indexAction()
    {
        $dm = $this->get('doctrine_phpcr')->getManager();
        $site = $dm->find('AppBundle\Document\Site', '/cms');
        $homepage = $site->getHomepage();

        if (!$homepage) {
            throw $this->createNotFoundException('No homepage configured');
        }

        return $this->forward('AppBundle:Default:page', array(
            'contentDocument' => $homepage
```

```
27             ));
28         }
29     }
```

In contrast to previous examples you specify a class when calling `find` - this is because you need to be *sure* that the returned document is of class `Site`.

Now test it out, visit: *http://localhost:8000*

Chapter 18
Conclusion

And that's it! Well done. You have created a very minimum but functional CMS which can act as a good foundation for larger projects!