Symfony

The Symfony CMF Book

Version: master

generated on January 22, 2019

Contents at a Glance

The Big Picture 4
The Model 9
The Router 13
The Third Party Bundles 16
Creating a Basic CMS using the RoutingAutoBundle 21
Getting Started 23
Routing and Automatic Routing 29
The Backend - Sonata Admin 33
Controllers and Templates 38
Creating a Menu 41
The Site Document and the Homepage 45
Conclusion 52

Chapter 1
The Big Picture

Start using the Symfony CMF in 10 minutes! This quick tour 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 Symfony CMF is created to solve this framework versus CMS dilemma. It provides Symfony bundles to easily add CMS features to your project. Yet, as you're still using the Symfony framework, you can build any custom functionality you can think of. This flexibility 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).

Downloading the Symfony CMF Sandbox

To explore the CMF, it is best to download the Symfony CMF Sandbox. The sandbox contains demonstrations for many of the CMF features and is a good playground to familiarize yourself with the CMF.

1. https://symfony.com/doc/current/quick_tour/the_big_picture.html
2. http://decoupledcms.org

When you want to start an actual project with the CMF, best 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.

The best way to download the Symfony CMF Sandbox is using *Composer*[4]:

Listing 1-1
```
$ composer create-project symfony-cmf/sandbox 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 PHP extension. After that, run these commands:

Listing 1-2
```
$ cd cmf-sandbox
$ cp app/config/phpcr_doctrine_dbal.yml.dist app/config/phpcr.yml
# Or when you're on a Windows PC:
# $ copy app\config\phpcr_doctrine_dbal.yml.dist app\config\phpcr.yml

$ 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 -n
```

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

Running the Symfony Application

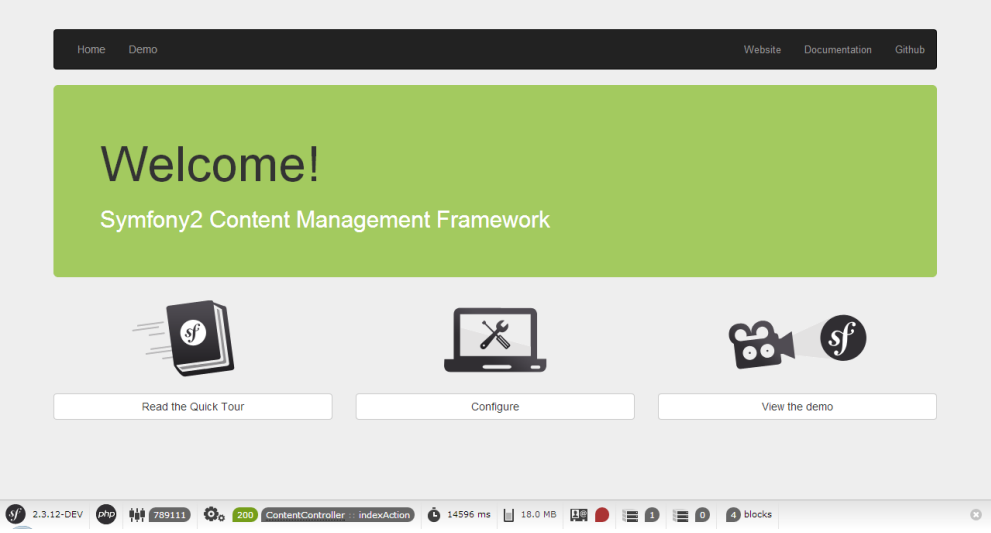
Use the `server:run` command to run a local server for the demo.

Running a CMF application is the same as running any Symfony application, see *Configuring a Web Server*[5] in the Symfony documentation.

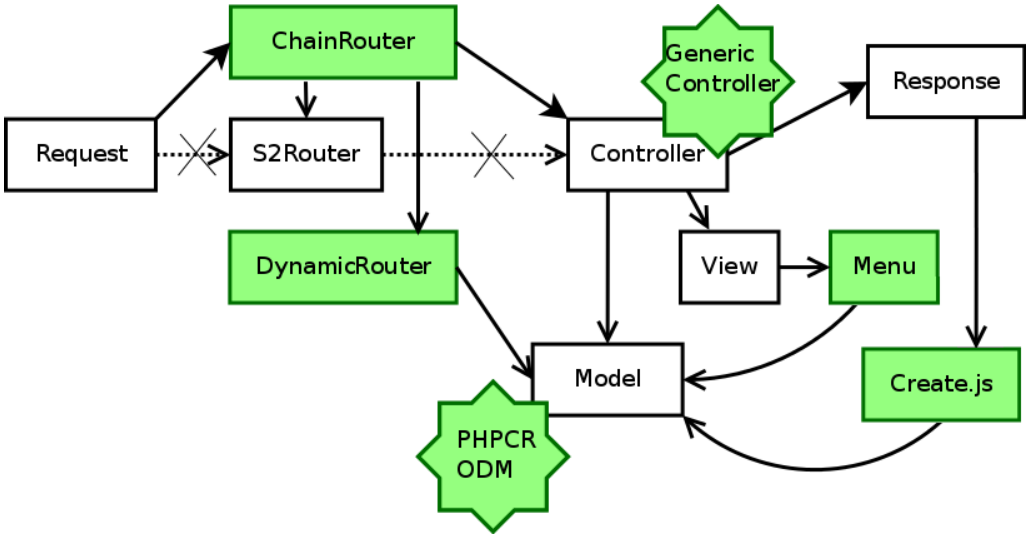
The Request Flow

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

3. https://github.com/symfony/symfony-standard
4. https://getcomposer.org/
5. http://symfony.com/doc/current/setup/web_server_configuration.html



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*[6], a Content Repository for Java. Together with some other developers they created *PHPCR*[7], 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.

6. https://en.wikipedia.org/wiki/Content_repository_API_for_Java
7. 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*[8], and it can have integration with Create.js, for live editing.

The Fixtures

Now you know the request flow, you can start editing content. While the normal usage will be to edit content through a web interface, the CMF sandbox also supports loading content from static files. This is mainly useful for testing purposes.

The fixtures are loaded with the `doctrine:phpcr:fixtures:load` command. To edit the home page, edit the first entry in `src/AppBundle/Resources/data/page.yml` to say something different. Then, run the `doctrine:phpcr:fixtures:load` command to get the changes into the content repository. After refreshing the browser, you can see your modifications!

Don't worry, editing fixture files is only done for developing and testing. The CMF comes with a Sonata admin integration for convenient online editing, or you can build your own editing systems.

8. `http://knpbundles.com/KnpLabs/KnpMenuBundle`

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 written in a storage agnostic way, 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 CMF Sandbox 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:
        menu:
            main:
                admin-item:
                projects-item:
                    cmf-item:
                company-item:
                    team-item:
            ...
        content:
            home:
                phpcr_locale:en:
                phpcr_locale:fr:
```

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

```
                phpcr_locale:de:
                seoMetadata:
                additionalInfoBlock:
                    child1:
            ...
        routes:
            en:
                company:
                    team:
                    more:
                about:
            ...
```

Each data is called a *node* in PHPCR. Everything is attached under the ROOT node (created by PHPCR itself).

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 `home` node. You can view these properties by adding the `--props` switch to the dump command:

Listing 2-3
```
$ php bin/console doctrine:phpcr:node:dump --props /cms/content/home
```

Previously, the PHPCR tree was compared with a Filesystem. While this gives you a good image of what happens, it's not the only truth. You can 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 provided by default in the Symfony Standard Edition, but for PHPCR instead of SQL databases.

Creating Content from Code

Now that 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 edited a page by using a Yaml file which was parsed by the fixture loader of the sandbox. This time, you'll create a page with PHP code.

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

Listing 2-4
```
// src/AppBundle/DataFixtures/PHPCR/LoadQuickTourData.php
namespace AppBundle\DataFixtures\PHPCR;

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

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

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

```
        return 100;
    }

    public function load(ObjectManager $documentManager)
    {
        // you will add code to this method in the next steps
    }
}
```

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

Listing 2-5
```
use Doctrine\ODM\PHPCR\DocumentManager;
use Symfony\Cmf\Bundle\ContentBundle\Doctrine\Phpcr\StaticContent;

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

    $content = new StaticContent();
    $content->setName('quick-tour'); // the name of the node
    $content->setTitle('Quick tour new Page');
    $content->setBody('I have added this page myself!');
}
```

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

Listing 2-6
```
public function load(ObjectManager $documentManager)
{
    // ...
    // get the root document
    $contentRoot = $documentManager->find(null, '/cms/content');
    $content->setParentDocument($contentRoot); // set the parent to the root
}
```

And finally, we have to tell the Document Manager to persist our content document using the Doctrine API:

Listing 2-7
```
public function load(ObjectManager $documentManager)
{
    // ...
    $documentManager->persist($content); // tell the document manager to track the content
    $documentManager->flush(); // doctrine is like a toilet: never forget to flush
}
```

Now you need to execute the `doctrine:phpcr:fixtures:load` command again. When dumping the nodes again, your new page should show up under `/cms/content/quick-tour`!

See "DoctrinePHPCRBundle" 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. Well done, making it 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

The router is central to the CMF. To understand this, let us look at 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 by this pattern.

But if you look at the base of a CMS, it only needs to handle one 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 - may need a specific template and controller.

Other parts of the CMF are also related to the Router. Two 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 straightforward: The router takes the request path and looks for a document with that path. Some examples:

Listing 3-1

```
/cms
    /routes
        /en            # /en Route
            /company   # /en/company Route
                /team  # /en/company/team Route
            /about     # /en/about Route
        /de            # /de Route
            /ueber     # /de/ueber 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 RoutingBundle, all routes are prefixed with `/cms/routes`.

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

Creating a new Route

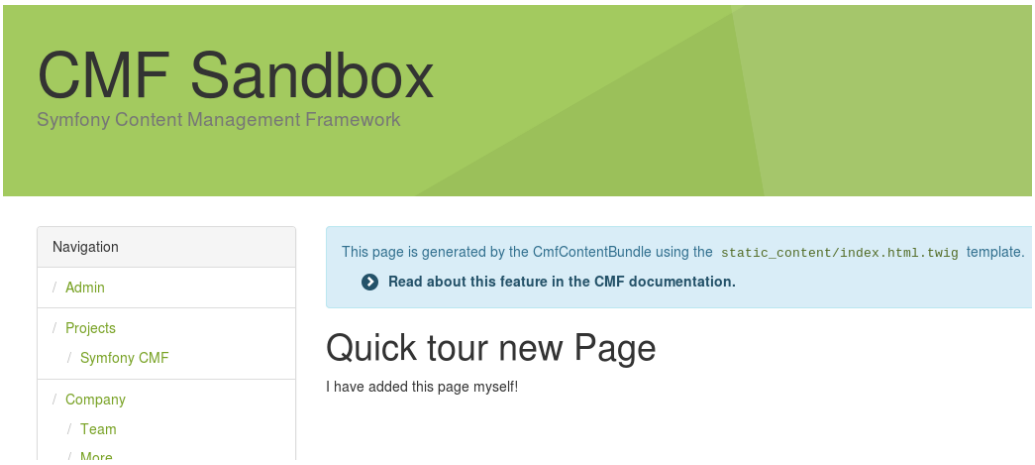
Now you know the basics of routing, you can add a new route to the tree using Doctrine:

Listing 3-2

```php
// src/AppBundle/DataFixtures/PHPCR/LoadQuickTourData.php
namespace AppBundle\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 LoadQuickTourData implements FixtureInterface, OrderedFixtureInterface
{
    public function load(ObjectManager $documentManager)
    {
        // static content from model chapter, resulting in $content being defined
        // ...

        $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');

        $route->setContent($content);

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

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`.



Chaining multiple Routers

Usually, you want to use both the `DynamicRouter` for the editable 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 tries each registered router and stops on the first router that returns a match.

By default, the `ChainRouter` overrides the Symfony router and only has the core and dynamic router in its chain. You can add more routers to the chain in the configuration or by tagging the router services with `cmf_routing.router`.

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 Drupal 8. 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 want 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 integrations of great existing bundles like the *KnpMenuBundle*[1] or *SonataAdminBundle*[2].

Initial Language Choice: Lunetics LocaleBundle

The CMF recommends to rely on the *LuneticsLocaleBundle*[3] to handle requests to `/` on your website. This bundle provides the tools to select the best locale for the user based on various criteria.

When you configure `lunetics_locale`, it is recommended to use a parameter for the locales, as you need to configure the locales for other bundles (e.g. the CoreBundle) too.

Listing 4-1
```
lunetics_locale:
    allowed_locales: "%locales%"
```

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 base view:

Listing 4-2
```
<!-- app/Resources/views/base.html.twig -->

<!-- ... -->
<div class="panel  panel-default  panel-nav">
    <div class="panel-heading">Navigation</div>

    {{ knp_menu_render('main', { template: 'includes/main_menu.html.twig' }) }}
</div>
```

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

As you can see, the menu is rendered by the `knp_menu_render` function. This seems a bit a strange, we are talking about the CmfMenuBundle and not the KnpMenuBundle, aren't we? That's correct, but 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 name. In this case, the menu contains all items implementing the `NodeInterface` inside the `/cms/menu/main` path (since the basepath in the CMF Sandbox is `/cms/menu`).

Creating a new Menu Entry

To add our quick tour page to the menu, we need to add a menu entry. The menu object references the content, which in turn is referenced by the route so that the URL can be created:

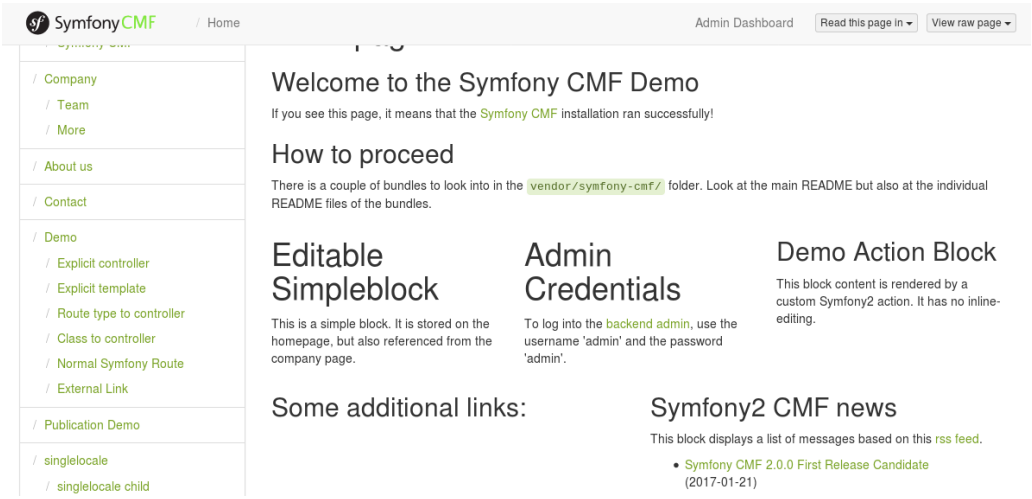
Listing 4-3

```php
// src/AppBundle/DataFixtures/PHPCR/LoadQuickTourData.php
namespace AppBundle\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 LoadQuickTourData implements FixtureInterface, OrderedFixtureInterface
{
    public function load(ObjectManager $documentManager)
    {
        // static content from model chapter, resulting in $content being defined
        // ...

        $menuMain = $documentManager->find(null, '/cms/menu/main');
        $menu = new MenuNode();
        $menu->setParentDocument($menuMain);
        $menu->setName('quick-tour');
        $menu->setLabel('Quick Tour');
        $menu->setContent($content);

        $documentManager->persist($menu);
        $documentManager->flush();
    }
}
```

Re-run the fixtures loading command and then refresh the web site. The menu entry is added at the bottom of the menu!

The BlockBundle

If you visit the homepage of the Sandbox, you'll see five 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*[4]. It provides the ability to store the blocks using PHPCR and it adds some commonly used blocks.

The SeoBundle

There is also a SeoBundle. This bundle is build on top of the *SonataSeoBundle*[5]. 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 Sandbox, 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 4-4
```php
// app/AppKernel.php

// ...
public function registerBundles()
{
    $bundles = [
        // ...
        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 4-5
```yaml
# app/config/config.yml
cmf_seo:
    title: "%%content_title%% | CMF Sandbox"
```

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/AppBundle/Resources/views/layout.html.twig` template with this:

Listing 4-6
```twig
{% block title %}{{ sonata_seo_title() }}{% endblock %}
```

4. https://sonata-project.org/bundles/block/master/doc/index.html
5. https://sonata-project.org/bundles/seo/master/doc/index.html

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 4-7
```yaml
# app/config/config.yml
sonata_seo:
    page:
        title: CMF Sandbox
```

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.

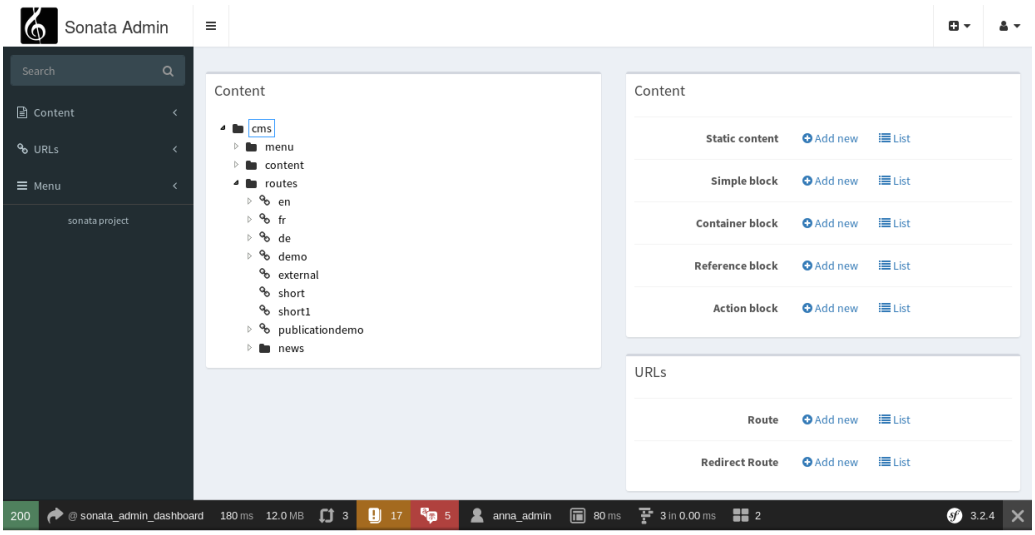
Sonata Admin

We have explained you that the CMF is based on a database, in order to make it editable by editor users without changing the code. But we haven't told you yet how an editor is able to maintain the website. Now it's time to reveal how to do that: Using the *SonataAdminBundle*[6]. The CmfSonataPhpcrAdminIntegrationBundle provides admin classes for all documents provided by the core CMF bundles.

By default, the Admin classes are all deactivated. Activate them for the bundles that you need admins for. For instance, to enable the MenuBundle Admin classes, you would do:

Listing 4-8
```yaml
# app/config/config.yml
cmf_sonata_phpcr_admin_integration:
    bundles:
        menu:
            enabled: true
```

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:



6. `https://sonata-project.org/bundles/admin/master/doc/index.html`

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

See the *Sonata Admin Integration Documentation* to learn about the configuration options for each admin.

Final Thoughts

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

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

Chapter 5
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:

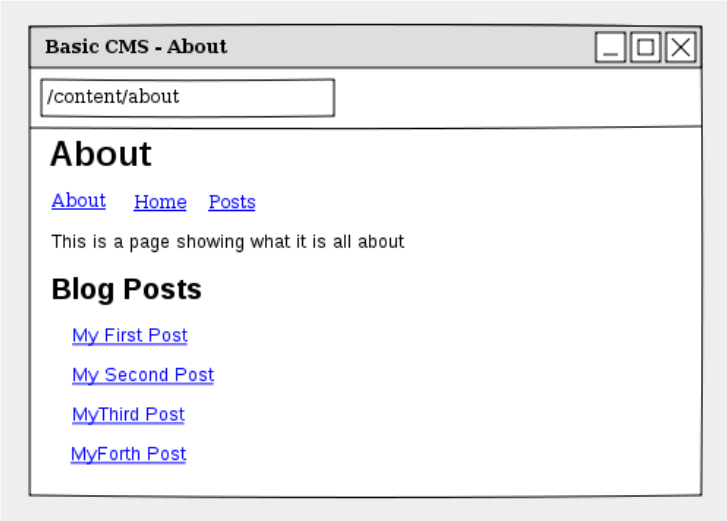
- A working knowledge of the Symfony framework;
- Basic knowledge of PHPCR-ODM.

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 6
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 6-1
```
$ 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 6-2
```
$ 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 6-3
```
$ php bin/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 6-4
```
$ php bin/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 6-5
```
// 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 6-6

```
// 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 6-7

```
// 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 6-8

```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 6-9
```
$ php bin/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 6-10
```
$ php bin/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 6-11
```
{
    ...
    require: {
        ...
        "doctrine/data-fixtures": "1.0.*"
    },
    ...
}
```

Create a page for your CMS:

Listing 6-12
```
// 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 6-13
```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 6-14
```
$ php bin/console doctrine:phpcr:fixtures:load
```

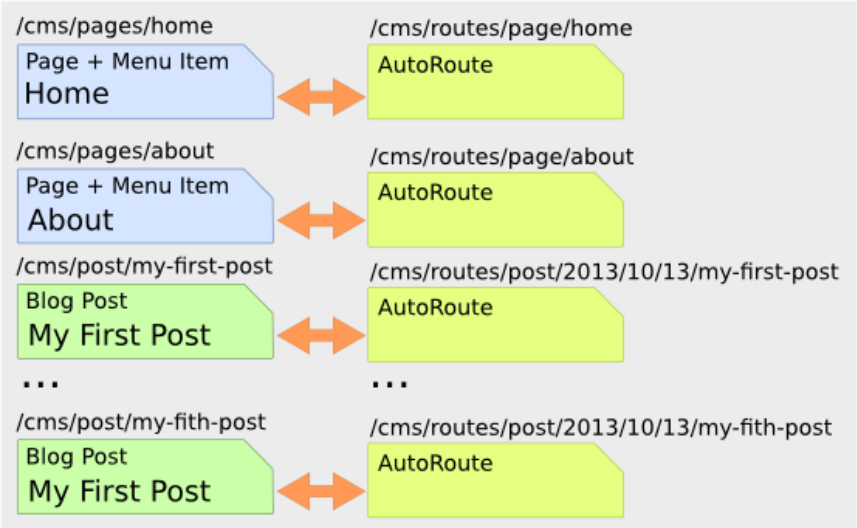
You should now have some data in your content repository.

Chapter 7
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 7-1
```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).

Add the following to your application configuration:

Listing 7-2
```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 7-3
```
# 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 7-4
```
# src/AppBundle/Resources/config/cmf_routing_auto.yml
AppBundle\Document\Page:
    definitions:
        main:
            uri_schema: /page/{title}
    token_providers:
        title: [content_method, { method: getTitle }]

AppBundle\Document\Post:
    definitions:
        main:
            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 7-5
```
$ php bin/console doctrine:phpcr:fixtures:load
```

Have a look at what you have:

Listing 7-6
```
$ 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:
```

```
16                my-fourth-post:
17      posts:
18        My First Post:
19        My Second Post:
20        My Third Post:
21        My Fourth Post:
```

The routes have been automatically created!

Chapter 8
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 8-1
```
$ 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 8-2
```
# app/config/config.yml

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

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:

Listing 8-3
```
# app/config/config.yml
cmf_routing:
    # ...
    dynamic:
        # ...
```

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

```
        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 8-4

```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 8-5

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

```

```php
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).

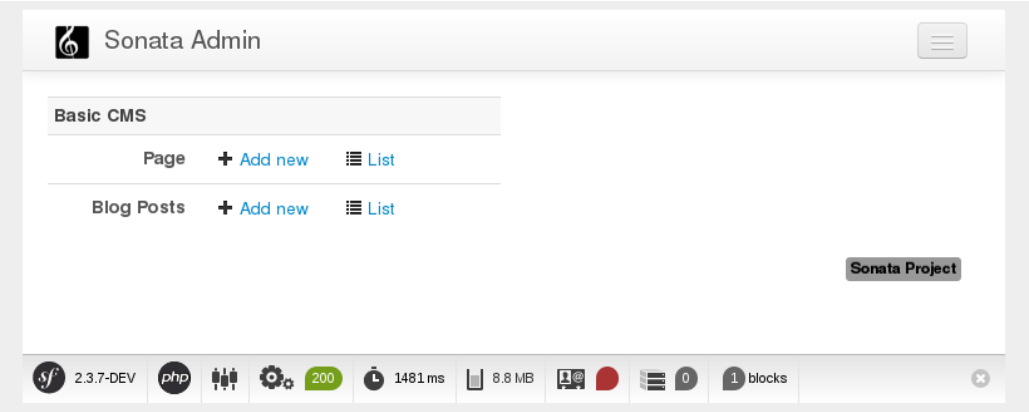
Now you just need to register these classes in the dependency injection container configuration:

Listing 8-6

```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 8-7
```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 8-8
```
$ 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 8-9
```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 8-10
```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 8-11

```
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 9
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 9-1
```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 9-2
```
$ 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 9-3
```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 9-4
```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 9-5
```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 10
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 10-1
```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 10-2
```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 10-3
```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 10-4

```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 10-5

```
$ 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 10-6
```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 }
```

and enable the Twig rendering functionality of the KnpMenuBundle:

Listing 10-7
```yaml
# app/config/config.yml
knp_menu:
    twig: true
```

and finally you can render the menu!

Listing 10-8
```twig
{# 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 11
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 11-1
```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;

```

```
21      public function getHomepage()
22      {
23          return $this->homepage;
24      }
25
26      public function setHomepage($homepage)
27      {
28          $this->homepage = $homepage;
29      }
30
31      public function setId($id)
32      {
33          $this->id = $id;
34      }
35  }
```

Initializing the Site Document

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

Listing 11-2

```
1  ROOT/
2      cms/
3         pages/
4         routes/
5         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 11-3

```
1  // src/AppBundle/Initializer/SiteInitializer.php
2  namespace AppBundle\Initializer;
3
4  use Doctrine\Bundle\PHPCRBundle\Initializer\InitializerInterface;
5  use PHPCR\Util\NodeHelper;
6  use Doctrine\Bundle\PHPCRBundle\ManagerRegistry;
7  use AppBundle\Document\Site;
8
9  class SiteInitializer implements InitializerInterface
10 {
11     private $basePath;
12
13     public function __construct($basePath = '/cms')
14     {
15         $this->basePath = $basePath;
16     }
17
18     public function init(ManagerRegistry $registry)
19     {
20         $dm = $registry->getManager();
21         if ($dm->find(null, $this->basePath)) {
22             return;
23         }
24
25         $site = new Site();
26         $site->setId($this->basePath);
27         $dm->persist($site);
28         $dm->flush();
29
```

```
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  }
```

Now:

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

Listing 11-4
```
# src/AppBundle/Resources/config/services.yml
services:
    # ...
    app.phpcr.initializer.site:
        class: AppBundle\Initializer\SiteInitializer
        tags:
            - { 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 11-5
```
$ php bin/console doctrine:phpcr:node:remove /cms
$ php bin/console doctrine:phpcr:repository:init
$ 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 11-6
```
$ php bin/console doctrine:phpcr:node:dump --props
ROOT:
  cms:
    - jcr:primaryType = nt:unstructured
    - phpcr:class = AppBundle\Document\Site
    ...
```

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 11-7
```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 11-8
```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 11-9
```php
// src/AppBundle/Controller/DefaultController.php

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

// ...
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:

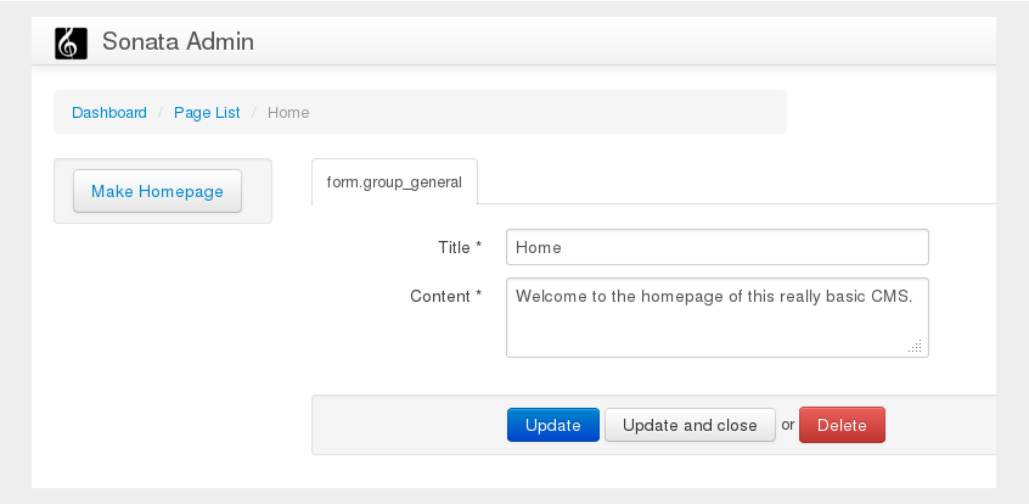
Listing 11-10

```
// 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 11-11

```
// 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
        ));
    }
}
```

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 12
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!