Symfony

The Symfony CMF Book

for Symfony master

generated on February 4, 2013

Contents at a Glance

Chapter 1
Installing the Symfony CMF Standard Edition

The goal of this tutorial is to install all the CMF components with the minimum necessary configuration and some very simple examples into a working Symfony2 application. This can be used to familiarize yourself with the CMF or to be used as a starting point for a new custom application.

If this is your first encounter with the Symfony CMF it would be a good idea to first take a look at:

- *The Big Picture*[1]
- The online sandbox demo at *cmf.liip.ch*[2]

For other Symfony CMF installation guides, please read: - The cookbook entry on *Installing the CMF sandbox* for instructions on how to install a more complete demo instance of Symfony CMF. - *Installing and configuring the CMF core* for step-by-step installation and configuration details of just the core components into an existing Symfony application.

Preconditions

As Symfony CMF is based on Symfony2, you should make sure you meet the *Requirements for running Symfony2*[3]. Additionally, you need to have *SQLite*[4] PDO extension (pdo_sqlite) installed, since it is used as the default storage medium.

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

Git[5] and *Curl*[6] are also needed to follow the installation steps listed below.

1. http://slides.liip.ch/static/2012-01-17_symfony_cmf_big_picture.html#1
2. http://cmf.liip.ch
3. http://symfony.com/doc/current/reference/requirements.html
4. http://www.sqlite.org/

Installation

The easiest way to install Symfony CMF is is using *Composer*[7]. Get it using

Listing 1-1
```
curl -s http://getcomposer.org/installer | php --
```

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

Listing 1-2
```
php composer.phar create-project symfony-cmf/standard-edition <path-to-install>
--stability=dev
mv composer.phar <path-to-install>/.
cd <path-to-install>
```

It is actually recommended to move `composer.phar` into the bin directory of your filesystem, so that you can access the command from any directory.

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

This will clone the standard edition and install all the dependencies and run some initial commands. These commands require write permissions to the `app/cache` and `app/logs` directory. In case the final commands end up giving permissions errors, please follow the guidelines in the official documentation for configuring the permissions and then run the `composer.phar install` command mentioned below.

If you prefer you can also just clone the project:

Listing 1-3
```
git clone git://github.com/symfony-cmf/symfony-cmf-standard.git <dir-name>
cd <dir-name>
```

If there were problems during the `create-project` command, or if you used `git clone` or if you updated the checkout later, always run the following command to update the dependencies:

Listing 1-4
```
php composer.phar install
```

The next step is to setup the database, if you want to use SQLite as your database backend just go ahead and run the following:

Listing 1-5
```
app/console doctrine:database:create
app/console doctrine:phpcr:init:dbal
app/console doctrine:phpcr:register-system-node-types
app/console doctrine:phpcr:fixtures:load
```

This will create a file called app.sqlite inside your app folder, containing the database content.

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

Listing 1-6
```
app/console server:run
```

And then access the CMF via:

5. http://git-scm.com/
6. http://curl.haxx.se/
7. http://getcomposer.org/

Listing 1-7
```
1 http://localhost:8000
```

If you prefer to use another database backend, for example MySQL, run the configurator (point your browser to `/web/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 should set it to `null`.

Overview

This guide 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.

For other Symfony CMF installation guides, please read: - The cookbook entry on *Installing the CMF sandbox* for instructions on how to install a more complete demo instance of Symfony CMF. - *Installing and configuring the CMF core* for step-by-step installation and configuration details of just the core components into an existing Symfony application.

AcmeMainBundle and SimpleCMSBundle

Symfony CMF SE comes with a default AcmeMainBundle to help you get started, in a similar way that Symfony2 has AcmeDemoBundle, providing you some demo pages visible on your browser. However, AcmeMainBundle doesn't include controllers or configuration files, like you probably would expect. It contains little more than a twig file and *Fixtures*[8] data, that was loaded into your database during installation.

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 with a specific ContentBundle's Content stored in the database.
- The retrieved content's information is used to determine which controller to pass it on to, and which template to use.
- As configured, the retrieved content is passed to ContentBundle's ContentController, which will handle it and render AcmeMainBundle's layout.html.twig.

  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, a careful look into each component is needed.

8. `http://symfony.com/doc/current/bundles/DoctrineFixturesBundle/index.html`

Chapter 2
Routing

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

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, as it's not suited to handle dynamic user defined routes, nor it scales well to a large number of routes.

The solution

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

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

With these requirements in mind, the Symfony CMF Routing component was developed.

The ChainRouter

At the core of Symfony CMF's Routing component sits the `ChainRouter`. It's used as a replacement for Symfony2's default routing system and, like it, 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 2-1
```yaml
# app/config/config.yml
symfony_cmf_routing_extra:
    chain:
        routers_by_id:
            # enable the DynamicRouter with high priority to allow overwriting configured
routes with content
            symfony_cmf_routing_extra.dynamic_router: 200

            # enable the symfony default router with a lower 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 2-2
```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 default 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 given provider. It then uses a matching process to the incoming request to a specific Route, which in turn is used to determine which Controller to forward the request to.

The bundle's default configuration states that `DynamicRouter` is disabled by default. To activate it, just add the following to your configuration file:

Listing 2-3

1. `http://api.symfony.com/master/Symfony/Component/Routing/RouterInterface.html`

```yaml
# app/config/config.yml
symfony_cmf_routing_extra:
    dynamic:
        enabled: true
```

This is the minimum configuration required to load the `DynamicRouter` as a service, thus making it capable of performing any routing. Actually, when you browse the default pages that come with the Symfony CMF SE, it's the `DynamicRouter` that's matching 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, and must implement the `RouteProviderInterface`. As part of this bundle, an implementation for *PHPCR-ODM*[2] is provided, but you can easily create your own, as the Router itself is storage agnostic. The default 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 *RoutingExtraBundle*.

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 *Routing*.

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 stored Route document itself can explicitly declare the target**

  Controller by specifying the '_controller' value in `getRouteDefaults()`.

- **By alias: the Route returns a 'type' value in `getRouteDefaults()`,**

  which is then matched against the provided configuration from config.yml

- **By class: requires the Route instance to implement `RouteObjectInterface`**

  and return an object for `getRouteContent()`. 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 in `getRouteDefaults()`.

2. `https://github.com/doctrine/phpcr-odm`

- **By class: requires the Route instance to implement `RouteObjectInterface`**

  and return an object for `getRouteContent()`. The returned class type is then matched against the provided configuration from config.yml.

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

Listing 2-4
```yaml
# app/config/config.yml
symfony_cmf_routing_extra:
    dynamic:
        generic_controller: symfony_cmf_content.controller:indexAction
        controllers_by_type:
            editablestatic: sandbox_main.controller:indexAction
        controllers_by_class:
            Symfony\Cmf\Bundle\ContentBundle\Document\StaticContent:
symfony_cmf_content.controller::indexAction
        templates_by_class:
            Symfony\Cmf\Bundle\ContentBundle\Document\StaticContent:
SymfonyCmfContentBundle:StaticContent:index.html.twig
```

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

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

Linking a Route with a Model instance

Depending on you 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` variable, if declared as parameter.

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

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

Redirections

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

Listing 2-5
```yaml
# app/config/config.yml
symfony_cmf_routing_extra:
    controllers_by_class:
```

```
3           Symfony\Cmf\Component\Routing\RedirectRouteInterface:
4   symfony_cmf_routing_extra.redirect_controller:redirectAction
```

The actual configuration for this association exists as a service, not as part of a config.yml file. Like 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 either an implementation of `RouteObjectInterface` or a `RouteAwareInterface` as `name` parameter
- Or supply an implementation of `ContentRepositoryInterface` and the id of the model instance as parameter `content_id`

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.

All routes are located under a configured root path, for example '/cms/routes'. A new route can be created in PHP code as follows:

Listing 2-6

```php
use Symfony\Cmf\Bundle\RoutingExtraBundle\Document\Route;
$route = new Route;
$route->setParent($dm->find(null, '/routes'));
$route->setName('projects');

// link a content to the route
$content = new Content('my content');
$route->setRouteContent($content);

// now configure some parameter, do not forget leading slash if you want /projects/{id}
and not /projects{id}
$route->setVariablePattern('/{id}');
$route->setRequirement('id', '\d+');
$route->setDefault('id', 1);
```

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

Your controller can expect the $id parameter as well as the $contentDocument as we set a content on the route. The content could be used to define an intro section that is the same for each project or other shared data. If you don't need content, you can just not set it in the document.

For more details, see the *route document section in the RoutingExtraBundle documentation*.

Integrating with SonataAdmin

If `sonata-project/doctrine-phpcr-admin-bundle` is added to the composer.json require section, the route documents are exposed in the SonataDoctrinePhpcrAdminBundle. For instructions on how to configure this Bundle see *SonataDoctrinePhpcrAdminBundle*.

By default, `use_sonata_admin` is automatically set based on whether `SonataDoctrinePhpcrAdminBundle` is available but you can explicitly disable it to not have it even if sonata is enabled, or explicitly enable to get an error if Sonata becomes unavailable.

You have a couple of configuration options for the admin. The `content_basepath` points to the root of your content documents.

Listing 2-7
```
# app/config/config.yml
symfony_cmf_routing_extra:
    use_sonata_admin: auto # use true/false to force using / not using sonata admin
    content_basepath: ~ # used with sonata admin to manage content, defaults to
symfony_cmf_core.content_basepath
```

Terms Form Type

The bundle defines a form type that can be used for classical "accept terms" checkboxes where you place urls in the label. Simply specify `symfony_cmf_routing_extra_terms_form_type` as the form type name and specify a label and an array with `content_ids` in the options

Listing 2-8
```
$form->add('terms', 'symfony_cmf_routing_extra_terms_form_type', array(
    'label' => 'I have seen the <a href="%team%">Team</a> and <a href="%more%">More</a>
pages ...',
    'content_ids' => array('%team%' => '/cms/content/static/team', '%more%' => '/cms/
content/static/more')
));
```

The form type automatically generates the routes for the specified content and passes the routes to the trans twig helper for replacement in the label.

Further notes

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

- *Routing* for most of the actual functionality implementation
- *RoutingExtraBundle* for Symfony2 integration bundle for Routing Bundle
- Symfony2's *Routing*[3] component page

3. `http://symfony.com/doc/current/components/routing/introduction.html`

Chapter 3
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.

Symfony CMF SE comes with the `ContentBundle`: a basic implementation of a content structure, including support for multiple languages and database storage of Routes.

Static Content

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

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

- `RouteAwareInterface` means that the content has associated Routes.
- `PublishWorkflowInterface` **means that the content has publishing and** unpublishing dates, which will be handled by Symfony CMF's core to determine access.

Multilang Static Content

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

It also specifies the translation strategy:

Listing 3-1

```
/**
 * @PHPCRODM\Document(translator="child", referenceable=true)
 */
```

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

Content Controller

To handle both content types, a `Controller` is also included. Its inner workings are pretty straightforward: it accepts a content instance and optionally a template to render it. If none is provided, it uses a pre-configured default. It also takes into account the document's publishing status and multi language. Both the content instance and the optional template are provided to the Controller by the `DynamicRouter` of the `RoutingExtraBundle`. 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*[2] , a back office generation tool. For more information about it, please refer to the bundle's *documentation section*[3].

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 `SonataDoctrinePHPCRAdminBundle` (refer to *Creating a CMS using CMF and Sonata* for instructions on that).

Configuration

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

Final thoughts

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

1. http://docs.doctrine-project.org/projects/doctrine-phpcr-odm/en/latest/reference/multilang.html
2. https://github.com/sonata-project/SonataDoctrinePhpcrAdminBundle
3. https://github.com/sonata-project/SonataDoctrinePhpcrAdminBundle/tree/master/Resources/doc

Chapter 4
Menu

Concept

No CMS system is complete without a menu system that allows users to navigate between content pages and perform certain actions. While it does usually map the actual content tree structure, menus often have a logic of their own, include options not mapped by contents 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 *KnpMenuBundle*[1], with a set of hierarchical, multi language menu elements, along with the tools to load and store them from/to a database. It also includes the administration panel definitions and related services needed for integration with *SonataDoctrinePHPCRAdminBundle*[2]

The `MenuBundle` extends and greatly relies on *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

`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 4-1

1. https://github.com/knplabs/KnpMenuBundle
2. https://github.com/sonata-project/SonataDoctrinePhpcrAdminBundle
3. https://github.com/knplabs/KnpMenuBundle
4. https://github.com/KnpLabs/KnpMenuBundle/blob/master/Resources/doc/index.md
5. https://github.com/KnpLabs/KnpMenuBundle/blob/master/Resources/doc/index.md#rendering-menus

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

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

The Provider

The core of `MenuBundle` is `PHPCRMenuProvider`, a `MenuProviderInterface` implementation that's responsible for dynamically loading menus from a PHPCR database. It does so based on the menu root's `path` value, by combining a preconfigured `basepath` value with a `name` given by the developer when instantiating the menu rendering call. This allows the `PHPCRMenuProvider` to handle several menu hierarchies using a single storage mechanism.

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

The Factory

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

The included implementation focuses on generating `MenuNode` instances from `NodeInterface` instances, as it is the best approach to handle tree-like structures like the ones typically used by CMS. Other approaches are implemented in the extended classes, and their respective documentation pages can be found in *KnpMenuBundle*[6]'s page.

`ContentAwareFactory` is responsible for getting the full menu hierarchy and rendering the respective `MenuNode` instances from the root node it receives from the `MenuProviderInterface` implementation. It is also responsible for determining which (if any) menu item is currently being viewed by the user. `KnpMenu` already includes a specific factory targeted at Symfony2's Routing component, which this bundle extends, to add support for:

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

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

The Menu Nodes

Also included in `MenuBundle` come two menu node content types: `MenuNode` and `MultilangMenuNode`. If you have read the documentation page regarding *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, like you would expect, plus some `attributes` for himself and its children, that will allow the actual rendering proccess to be customized. It also includes a `Route` field and two references to Contents. These are used to store an associated `Route` object, plus one (not two, despite the fact that two fields exist) Content element. The `MenuNode` can have a strong (integrity ensured) or weak (integrity not ensured) reference to the actual Content element it points to, it's up to you to choose which best fits your scenario. You can find more information on references on the *Doctrine PHPCR documentation page*[7].

6. https://github.com/knplabs/KnpMenuBundle
7. http://docs.doctrine-project.org/projects/doctrine-phpcr-odm/en/latest/reference/association-mapping.html#references

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

It also specifies the strategy used to store the multiple translations to database:

Listing 4-2
```
/**
 * @PHPCRODM\Document(translator="attribute")
 */
```

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

The `MenuNode` and `MultilangMenuNode` content types exist to preserve backwards compatibility with previous versions of the bundle, but they simply extend their Node counterparts. These classes are deprecated, and will be removed in a later version.

Admin support

`MenuBundle` also includes the administration panels and respective services needed for integration with *SonataDoctrinePHPCRAdminBundle*[9], a back office generation tool that can be installed with Symfony CMF. For more information about it, please refer to the bundle's *documentation section*[10].

The included administration panels will automatically be loaded if you install `SonataDoctrinePHPCRAdminBundle` (refer to *Creating a CMS using CMF and Sonata* for instructions on how to do so).

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.

Further notes

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

- *MenuBundle* for advanced details and configuration reference
- *KnpMenuBundle*[11] page for information on the bundle on which `MenuBundle` relies
- *KnpMenu*[12] page for information on the underlying library used by `KnpMenuBundle`

8. http://docs.doctrine-project.org/projects/doctrine-phpcr-odm/en/latest/reference/multilang.html
9. https://github.com/sonata-project/SonataDoctrinePhpcrAdminBundle
10. https://github.com/sonata-project/SonataDoctrinePhpcrAdminBundle/tree/master/Resources/doc
11. https://github.com/knplabs/KnpMenuBundle
12. https://github.com/knplabs/KnpMenu

Chapter 5
SimpleCMS

Concept

In the previous documentation pages all the basic components of Symfony CMF have been analysed: the *Routing* that allows you to associate URLs with your *Content*, which users can browse using a *Menu*.

These three components complement each other but are independent: they work without each other, allowing you to choose which ones you want to use, extend or ignore. In some cases, however, you might just want a simple implementation that gathers all those functionalities in a ready-to-go package. For that purpose, the `SimpleCMSBundle` was created.

SimpleCMSBundle

`SimpleCMSBundle` is implemented on top of most of the other Symfony CMF Bundles, combining them into a functional CMS. It's a simple solution, but you will find it very useful when you start implementing your own CMS using Symfony CMF. Whether you decide to extend or replace it, it's up to you, but in both cases, it's a good place to start developing your first CMS.

Page

`SimpleCMSBundle` basic content type is `Page`. Its class declaration points out many of the features available:

- **It extends `Route`, meaning it's not only a `Content` instance, but**

  also a `Route`. In this case, as declared in `getRouteContent()`, the `Route` as an associated content, itself.

- **It implements `RouteAwareInterface`, which means it has associated `Route`**

  instances. As expected, and as seen in `getRoutes()`, it has only one `Route` associated: itself.

- **It implements `NodeInterface`, which means it can be used by `MenuBundle`**

  to generate a menu structure.

The class itself is similar to the `StaticContent` already described in the documentation page regarding *Content*, although some key atributes, like `parent` or `path` come from the `Route` class it extends.

Three-in-one

Like explained before, this bundle gathers functionality from three distinct bundles: *Content*, *Routing* and *Menu*. The routing component receives a request, that it matches to a `Route` instance loaded from database. That `Route` points to a `Content` instance: itself. A controller is also determined by the routing component, that renders the `Content` using a template which, in turn, presents the user with a HTML visualization of the stored information tree structure, rendered using `MenuItem` obtained from equivalent `NodeInterface` instances.

`SimpleCMSBundle` simplifies this process: `Content`, `Route` and `NodeInterface` are gathered in one class: `Page`. This three-in-one approach is the key concept behind this bundle.

MultilangPage

Like you would expect, a multilanguage version of `Page` is also included. `MultilangPage` defines a `locale` variable and which fields will be translated (`title`, `label` and `body`). It also includes `getStaticPrefix()` to handle the path prefix of the `Page`. This is part of the route handling mechanism, and will be analysed bellow.

> The `MultilangPage` uses the `attribute` strategy for translation: the several translations coexist in the same database entry, and the several translated versions of each field are stored as different attributes in that same entry.
>
> As the routing is not separated from the content, it is not possible to create different routes for different languages. This is one of the biggest disadvantages of the `SimpleCmsBundle`.

Configuring the content class

By default, `SimpleCMSBundle` will use `Symfony\Cmf\Bundle\SimpleCmsBundle\Document\Page` as the content class if multilanguage is not enabled (default). If no other class is chosen, and multilanguage support is enabled, it will automatically switch to `Symfony\Cmf\Bundle\SimpleCmsBundle\Document\MultilangPage`. You can explicitly specify your content class and/or enable multilanguage support using the configuration parameters

Listing 5-1
```yaml
# app/config/config.yml
symfony_cmf_simple_cms:
    document_class: ~  # Symfony\Cmf\Bundle\SimpleCmsBundle\Document\Page
    multilang:
        locales: ~  # defaults to [], declare your locales here to enable multilanguage
```

SimpleCMSBundle in detail

Now that you understand what `SimpleCMSBundle` does, we'll detail how it does it. Several other components are part of this bundle, that change the default behaviour of its dependencies.

The routing

`SimpleCMSBundle` doesn't add much functionality to the routing part of Symfony CMF. Instead, it greatly relies on `RoutingExtraBundle` and its set of configurable functionalities to meet its requirements. It declares an independent `DynamicRouter`, with it's own specific `RouteProvider`, `NestedMatcher`, Enhancers set and other useful services, all of them instances of the classes bundled with `RoutingBundle` and `RoutingExtraBudle`. This service declaration duplication allows you to reuse the original `RoutingExtraBundle` configuration options to declare another Router, if you wish to do so.

The only exception to this is `RouteProvider`: the `SimpleCMSBundle` has its own strategy to retrieve `Route` instances from database. This is related with the way `Route` instances are stored in database by `RoutingExtraBundle`. By default, the `path` parameter will hold the prefixed full URI, including the locale identifier. This would mean an independent `Route` instance should exist for each translation of the same `Content`. However, as we've seen, `MultilangPage```stores all translations in the same entry. So, to avoid duplication, the locale prefix is stripped from the URI prior to persistance, and ``SimpleCMSBundle` includes `MultilangRouteProvider`, which is responsible for fetching `Route` instances taking that into account.

When rendering the actual URL from `Route`, the locale prefix needs to be put back, otherwise the resulting addresses wouldn't specify the locale they refer to. To do so, `MultilangPage` uses the already mentioned `getStaticPrefix()` implementation.

Exemplifying: An incoming request for `contact` would be prefixed with `/cms/simple` basepath, and the storage would be queried for `/cms/simple/contact/`. However, in a multilanguage setup, the locale is prefixed to the URI, resulting in a query either for `/cms/simple/en/contact/` or `/cms/simple/de/contact/`, which would require two independent entries to exist for the same actual content. With the above mentioned approach, the `locale` is stripped from the URI prior to `basepath` prepending, resulting in a query for `/cms/simple/contact/` in both cases.

Routes and Redirections

`SimpleCMSBundle` includes `MultilangRoute` and `MultilangRedirectRoute`, extensions to the `Route` and `RedirectRoute` found in `RoutingExtraBudle`, but with the necessary changes to handle the prefix strategy discussed earlier.

Content handling

`Route` instances are responsible for determining which `Controller` will handle the current request. *Getting the Controller and Template* shows how Symfony CMF SE can determine which `Controller` to use when rendering a certain content, and `SimpleCMSBundle` uses these mechanisms to do so.

Listing 5-2
```yaml
# app/config/config.yml
symfony_cmf_simple_cms:
    generic_controller: ~  # symfony_cmf_content.controller:indexAction
```

By default, it uses the above mentioned service, which instanciates `ContentController` from `ContentBundle`. The default configuration associates all `document_class` instances with this `Controller`, and specifies no default template. However, you can configure several `controllers_by_class` and `templates_by_class` rules, which will associate, respectively, `Controller` and templates to a specific Content type. Symfony CMF SE includes an example of both in its default configuration.

Listing 5-3
```yaml
# app/config/config.yml
symfony_cmf_simple_cms:
    routing:
        templates_by_class:
```

```
            Symfony\Cmf\Bundle\SimpleCmsBundle\Document\Page:
SymfonyCmfSimpleCmsBundle:Page:index.html.twig
        controllers_by_class:
            Symfony\Cmf\Bundle\RoutingExtraBundle\Document\RedirectRoute:
symfony_cmf_routing_extra.redirect_controller:redirectAction
```

These configuration parameters will be used to instantiate *Route Enhancers*. More information about them can be found in the *Routing* component documentation page.

These specific example determines that content instances of class `Page` will be rendered using the above mentioned template, if no other is explicitly provided by the associated `Route` (which, in this case, is `Page` itself). It also states that all contents that instantiate `RedirectRoute` will be rendered using the mentioned `Controller` instead of the default. Again, the actual `Route` can provided a controller, in will take priority over this one. Both the template and the controller are part of `SimpleCMSBundle`.

Menu generation

Like mentioned before, `Page` implements `NodeInterface`, which means it can be used to generate `MenuItem` that will, in turn, be rendered into HTML menus presented to the user.

To do so, the default `MenuBundle` mechanisms are used, only a custom `basepath` is provided to the `PHPCRMenuProvider` instance. This is defined in `SimpleCMSBundle` configuration options, and used when handling content storage, to support functionality as described in *Menu* documentation. This parameter is optional, can be configured like so:

Listing 5-4
```yaml
# app/config/config.yml
symfony_cmf_simple_cms:
    use_menu: ~  # defaults to auto , true/false can be used to force providing / not
providing a menu
    basepath: ~  # /cms/simple
```

Admin support

`SimpleCMSBundle` also includes the administration panel and respective service needed for integration with *SonataDoctrinePHPCRAdminBundle*[1], a backoffice generation tool that can be installed with Symfony CMF. For more information about it, please refer to the bundle's *documentation section*[2].

The included administration panels will automatically be loaded if you install `SonataDoctrinePHPCRAdminBundle` (refer to *Creating a CMS using CMF and Sonata* for instructions on how to do so). You can change this behaviour with the following configuration option:

Listing 5-5
```yaml
# app/config/config.yml
symfony_cmf_simple_cms:
    use_sonata_admin: ~  # defaults to auto , true/false can be used to using / not using
SonataAdmin
```

1. https://github.com/sonata-project/SonataDoctrinePhpcrAdminBundle
2. https://github.com/sonata-project/SonataDoctrinePhpcrAdminBundle/tree/master/Resources/doc

Fixtures

`SimpleCMSBundle` includes a support class for integration with *DoctrineFixturesBundle*[3], aimed at making loading initial data easier. A working example is provided in Symfony CMF SE, that illustrates how you can easily generate `MultilangPage` and `MultilangMenuNode` instances from yml files.

Configuration

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

Further notes

For more information on the SimpleCMSBundle, please refer to:

- ***SimpleCmsBundle* for configuration reference and advanced details**
  about the bundle.
- ***Routing* for information about the routing component**
  in which `SimpleCMSBundle` is based on.
- ***Content* for information about the base content**
  bundle that `SimpleCMSBundle` depends on.
- ***Menu* for information about the menu system used**
  by `SimpleCMSBundle`.

3. `http://symfony.com/doc/master/bundles/DoctrineFixturesBundle/index.html`

Chapter 6
Choosing a storage layer

When building a CMS no doubt the choice of storage layer is one of the key decisions to take. Many factors must be considered, the good news is that with all the components and Bundles in the CMF we take extra care to provide the necessary extension points to ensure the **CMF remains storage layer agnostic**.

The goal of this tutorial is to explain the considerations and why we suggest *PHPCR*[1] and *PHPCR-ODM*[2] as the ideal basis for a CMS. However all components and Bundles can be integrated with other solutions with a fairly small amount of work.

Requirements for a CMS storage layer

At the most fundamental level a CMS is about storing, so the first requirement is that *a CMS must provide means to store content with different properties*.

A CMS has very different storage needs than for example a system for processing orders. Do note however that it is entirely possible and very intended of the CMF initiative to enable developers to combine the CMF with a system for processing orders. So for example one could create a shopping solution using the CMF for storing the product catalog, while using another system for maintaining the inventory, customer data and orders. This leads to the second requirement, *a CMS must provide means to reference content*, both content stored inside the CMS, but also in other systems.

The actual content in a CMS tends to be organized in a tree like structure, mimicking a file system. Note that content authors might want to use different structures for how to organize the content and how to organize other aspects like the menu and the routing. This leads to the third requirement, *a CMS must provide means to represent the content as a tree structure*. Furthermore a fourth requirement is that *a CMS should allow maintaining several independent tree structures*.

In general data inside a CMS tends to be unstructured. So while several pages inside the CMS might be very similar, there is a good chance that there will be many permutations needing different extra fields, therefore *a CMS must not enforce a singular schema for content*. That being said, in order to better maintain the content structure and enabling UI layers from generically displaying content elements it is important to optionally be able to express rules that must be followed and that can also help attach

1. http://phpcr.github.com
2. http://www.doctrine-project.org/projects/phpcr-odm.html

additional semantic meaning. So *a CMS must provide means to optionally define a schema for content elements*.

This requirement actually also relates to another need, in that a CMS must make it easy for content authors to prepare a series of changes in a staging environment that then needs to go online in a single step. This means another requirement is that it is necessary that *a CMS should support moving and exporting content between independent tree structures*. Note that exporting can be useful also for backups.

When making changes it would however also be useful to be able to version the change sets, so that they remain available for historical purposes, but also to be able to revert whenever needed. Therefore the next requirement is that *a CMS should provide the ability to version content*.

As we live in a globalized world, websites need to provide content in multiple languages addressing different regions. However not all pieces of content need to be translated and others might only be eventually translated but until then the user should be presented the content in one of the available languages, so *a CMS should provide the ability to store content in different languages, with optional fallback rules*.

As a CMS usually tends to store an increasing amount of content it will become necessary to provide some way for users to search the content even when the user has only a very fuzzy idea about the content they are looking for, leading to the requirement that *a CMS must provide full text search capabilities*, ideally leveraging both the contents tree structure and the data schema.

Another popular need is limiting read and/or write access of content to specific users or groups. Ideally this solution would also integrate with the tree structure. So it would be useful if *a CMS provides capabilities to define access controls* that leverage the tree structure to quickly manage access for entire subtrees.

Finally not all steps in the content authoring process will be done by the same person. As a matter of fact there might be multiple steps all of which might not even be done by a person. Instead some of the steps might even be executed by a machine. So for example a photographer might upload a new image, a content author might attach the photo to some text, then the system automatically generates thumbnails and web optimized renditions and finally an editor decides on the final publication. Therefore *a CMS should provide capabilities to assist in the management of workflows*.

Summary

Here is a summary of the above requirements. Note some of the requirements have a *must*, while others only have a *should*. Obviously depending on your use case you might prioritize features differently:

- a CMS must provide means to store content with different properties
- a CMS must provide means to reference content
- a CMS must provide means to represent the content as a tree structure
- a CMS must provide full text search capabilities
- a CMS must not enforce a singular schema for content
- a CMS must provide means to optionally define a schema for content elements
- a CMS should allow maintaining several independent tree structures
- a CMS should support moving and exporting content between independent tree structures
- a CMS should provide the ability to version content
- a CMS should provide the ability to store content in different languages, with optional fallback rules
- a CMS should provides capabilities to define access controls
- a CMS should provide capabilities to assist in the management of workflows

RDBMS

Looking at the above requirements it becomes apparent that out the box an RDBMS is ill-suited to address the needs of a CMS. RDBMS were never intended to store tree structures of unstructured content. Really the only requirement RDBMS cover from the above list is the ability to store content, some way to reference content, keep multiple separate content structures and a basic level of access controls and triggers.

This is not a failing of RDBMS in the sense that they were simply designed for a different use case: the ability to store, manipulate and aggregate structured data. This makes them ideal for storing inventory and orders.

That is not to say that it is impossible to build a system on top of an RDBMS that addresses more or even all of the above topics. Some RDBMS natively support recursive queries, which can be useful for retrieving tree structures. Even if such native support is missing, there are algorithms like materialized path and nested sets that can enable efficient storage and retrieval of tree structures for different use cases.

The point is however that these all require algorithms and code on top of an RDBMS which also tightly bind your business logic to a particular RDBMS and/or algorithm even if some of them can be abstracted. So again using an ORM one could create a pluggable system for managing tree structures with different algorithms which prevent binding the business logic of the CMS to a particular algorithm.

However it should be said once more, that all Bundles and Components in the CMF are developed to enable any persistent storage API and we welcome contributions for adding implementations for other storage systems. So for example RoutingExtraBundle currently only provides Document classes for PHPCR ODM, but the interfaces defined in the Routing component are storage agnostic and we would accept a contribution to add Doctrine ORM support.

PHPCR

PHPCR essentially is a set of interfaces addressing most of the requirements from the above list. This means that PHPCR is totally storage agnostic in the sense that it is possible to really put any persistence solution behind PHPCR. So in the same way as an ORM can support different tree storage algorithms via some plugin, PHPCR aims to provide an API for the entire breath of CMS needs, therefore cleanly separating the entire business logic of your CMS from the persistence choice. As a matter of fact the only feature above not natively supported by PHPCR is support for translations.

Thanks to the availability of several PHPCR implementations supporting various kinds of persistence choices, creating a CMS on top of PHPCR means that end users are enabled to pick and choose what works best for them, their available resources, their expertise and their scalability requirements.

So for the simplest use cases there is for example a Doctrine DBAL based solution provided by the *Jackalope*[3] PHPCR implementation that can use the SQLite RDBMS shipped with PHP itself. At the other end of the spectrum Jackalope also supports *Jackrabbit*[4] which supports clustering and can efficiently handle data into the hundreds of gigabytes. By default Jackrabbit simply uses the file system for persistence, but it can also use an RDBMS. However future versions will support MongoDB and support for other NoSQL solutions like CouchDB or Cassandra is entirely possible. Again, switching the persistence solution would require no code changes as the business logic is only bound to the PHPCR interfaces.

Please see *Installing and configuring Doctrine PHPCR-ODM* for more details on the available PHPCR implementations and their requirements and how to setup Symfony2 with one of them.

3. `https://github.com/jackalope/jackalope`
4. `http://jackrabbit.apache.org`

PHPCR ODM

As mentioned above using PHPCR does not mean giving up on RDBMS. In many ways, PHPCR can be considered a specialized ORM solution for CMS. However while PHPCR works with so called *nodes*, in an ORM people expect to be able to map class instances to a persistence layer. This is exactly what PHPCR ODM provides. It follows the same interface classes as Doctrine ORM while also exposing all the additional capabilities of PHPCR, like trees and versioning. Furthermore, it also provides native support for translations, covering the only omission of PHPCR for the above mentioned requirements list of a CMS storage solution.

Chapter 7
Installing and configuring the CMF core

The goal of this tutorial is to install the minimal CMF components ("core") with the minimum necessary configuration. From there, you can begin incorporating CMF functionality into your application as needed.

This is aimed at experienced user who want to know all about the Symfony CMF details. If this is your first encounter with the Symfony CMF it would be a good idea to start with:

- *Installing the Symfony CMF Standard Edition* page for instructions on how to quickly install the CMF (recommended for development)
- *Installing the CMF sandbox* for instructions on how to install a demonstration sandbox.

Preconditions

- *Installation of Symfony2*[1] (2.1.x)
- *Installing and configuring Doctrine PHPCR-ODM*

Installation

Download the bundles

Add the following to your `composer.json` file

Listing 7-1
```
"minimum-stability": "dev",
"require": {
    ...
    "symfony-cmf/symfony-cmf": "1.0.*"
}
```

And then run

1. `http://symfony.com/doc/2.1/book/installation.html`

Listing 7-2
```
1 php composer.phar update
```

Initialize bundles

Next, initialize the bundles in `AppKernel.php` by adding them to the `registerBundles` method

Listing 7-3
```
1  // app/AppKernel.php
2
3  public function registerBundles()
4  {
5      $bundles = array(
6          // ...
7
8          new Symfony\Cmf\Bundle\RoutingExtraBundle\SymfonyCmfRoutingExtraBundle(),
9          new Symfony\Cmf\Bundle\CoreBundle\SymfonyCmfCoreBundle(),
10         new Symfony\Cmf\Bundle\MenuBundle\SymfonyCmfMenuBundle(),
11         new Symfony\Cmf\Bundle\ContentBundle\SymfonyCmfContentBundle(),
12         new Symfony\Cmf\Bundle\BlockBundle\SymfonyCmfBlockBundle(),
13
14         // Dependencies of the SymfonyCmfMenuBundle
15         new Knp\Bundle\MenuBundle\KnpMenuBundle(),
16
17         // Dependencies of the SymfonyCmfBlockBundle
18         new Sonata\BlockBundle\SonataBlockBundle(),
19     );
20
21     // ...
22 }
```

This also enables the PHPCR-ODM and related dependencies; setup instructions can be found in the dedicated documentation.

Configuration

To get your application running, very little configuration is needed.

Minimum configuration

These steps are needed to ensure your AppKernel still runs.

If you haven't done so already, make sure you have followed these steps from *Installing and configuring Doctrine PHPCR-ODM*:

- Initialize `DoctrinePHPCRBundle` in `app/AppKernel.php`
- Ensure there is a `doctrine_phpcr:` section in `app/config/config.yml`
- Add the `AnnotationRegistry::registerFile` line to `app/autoload.php`

Configure the BlockBundle in your `config.yml`:

Listing 7-4

```yaml
# app/config/config.yml
sonata_block:
    default_contexts: [cms]
```

Additional configuration

Because most CMF components use the DynamicRouter from the RoutingExtraBundle, which by default is not loaded, you will need to enable it as follows:

Listing 7-5

```yaml
# app/config/config.yml
symfony_cmf_routing_extra:
    chain:
        routers_by_id:
            symfony_cmf_routing_extra.dynamic_router: 200
            router.default: 100
    dynamic:
        enabled: true
```

You might want to configure more on the dynamic router, i.e. to automatically choose controllers based on content. See *RoutingExtraBundle* for details.

For now this is the only configuration we need. Mastering the configuration of the different bundles will be handled in further tutorials. If you're looking for the configuration of a specific bundle take a look at the corresponding *bundles entry*.

Chapter 8
Installing and configuring Doctrine PHPCR-ODM

The Symfony2 CMF needs somewhere to store the content. Many of the bundles provide documents and mappings for the PHP Content Repository - Object Document Mapper (PHPCR-ODM), and the documentation is currently based around using this. However, it should also be possible to use any other form of content storage, such as another ORM/ODM or MongoDB.

The goal of this tutorial is to install and configure Doctrine PHPCR-ODM, ready for you to get started with the CMF.

For more details see the *full PHPCR-ODM documentation*[1]. Some additional information can be found in the *DoctrinePHPCRBundle* reference, which for the most part mimics the standard *DoctrineBundle*[2].

Finally for information about PHPCR see the *official PHPCR website*[3].

If you just want to use PHPCR but not the PHPCR-ODM, you can skip the step about registering annotations and the part of the configuration section starting with *odm*.

Preconditions

- php >= 5.3
- libxml version >= 2.7.0 (due to a bug in libxml *http://bugs.php.net/bug.php?id=36501*[4])
- phpunit >= 3.6 (if you want to run the tests)
- Symfony2 (version 2.1.x)

1. http://www.doctrine-project.org/projects/phpcr-odm.html
2. https://github.com/doctrine/DoctrineBundle
3. http://phpcr.github.com
4. http://bugs.php.net/bug.php?id=36501

Installation

Choosing a content repository

The first thing to decide is what content repository to use. A content repository is essentially a database that will be responsible for storing all the content you want to persist. It provides an API that is optimized for the needs of a CMS (tree structures, references, versioning, full text search etc.). While every content repository can have very different requirements and performance characteristics, the API is the same for all of them.

Furthermore, since the API defines an export/import format, you can always switch to a different content repository implementation later on.

These are the available choices:

- **Jackalope with Jackrabbit** (Jackrabbit requires Java, it can persist into the file system, a database etc.)
- **Jackalope with Doctrine DBAL** (requires an RDBMS like MySQL, PostgreSQL or SQLite)
- **Midgard** (requires the midgard2 PHP extension and an RDBMS like MySQL, PostgreSQL or SQLite)

The following documentation includes examples for all of the above options.

If you are just getting started with the CMF, it is best to choose a content repository based on a storage engine that you are already familiar with. For example, **Jackalope with Doctrine DBAL** will work with your existing RDBMS and does not require you to install Java or the midgard2 PHP extension. Once you have a working application it should be easy to switch to another option.

Download the bundles

Add the following to your `composer.json` file, depending on your chosen content repository.
Jackalope with Jackrabbit

Listing 8-1

```
"minimum-stability": "dev",
"require": {
    ...
    "jackalope/jackalope-jackrabbit": "1.0.*",
    "doctrine/phpcr-bundle": "1.0.*",
    "doctrine/phpcr-odm": "1.0.*"
}
```

Jackalope with Doctrine DBAL

Listing 8-2

```
"minimum-stability": "dev",
"require": {
    ...
    "jackalope/jackalope-doctrine-dbal": "dev-master",
    "doctrine/phpcr-bundle": "1.0.*",
    "doctrine/phpcr-odm": "1.0.*"
}
```

Midgard

Listing 8-3

```
"minimum-stability": "dev",
"require": {
```

```
    ...
    "midgard/phpcr": "dev-master",
    "doctrine/phpcr-bundle": "1.0.*",
    "doctrine/phpcr-odm": "1.0.*"
}
```

For all of the above, if you are also using Doctrine ORM, make sure to use `"doctrine/orm": "2.3.*"`, otherwise composer can't resolve the dependencies as Doctrine PHPCR-ODM depends on the newer 2.3 Doctrine Commons. (Symfony2.1 standard edition uses "2.2.*".)

To install the above dependencies, run:

Listing 8-4
```
php composer.phar update
```

Register annotations

PHPCR-ODM uses annotations and these need to be registered in your `app/autoload.php` file. Add the following line, immediately after the last `AnnotationRegistry::registerFile` line:

Listing 8-5
```
// app/autoload.php

// ...
AnnotationRegistry::registerFile(__DIR__.'/../vendor/doctrine/phpcr-odm/lib/Doctrine/ODM/
PHPCR/Mapping/Annotations/DoctrineAnnotations.php');
// ...
```

Initialize bundles

Next, initialize the bundles in `app/AppKernel.php` by adding them to the `registerBundle` method:

Listing 8-6
```
// app/AppKernel.php

public function registerBundles()
{
    $bundles = array(
        // ...

        // Doctrine PHPCR
        new Doctrine\Bundle\PHPCRBundle\DoctrinePHPCRBundle(),

    );
    // ...
}
```

Configuration

Next step is to configure the bundles.

PHPCR Session

Basic configuration for each content repository is shown below; add the appropriate lines to your `app/config/config.yml`. More information on configuring this bundle can be found in the reference chapter *DoctrinePHPCRBundle*.

The workspace, username and password parameters are for the PHPCR repository and should not be confused with possible database credentials. They come from your content repository setup. If you want to use a different workspace than *default* you have to create it first in your repository.

If you want to use the PHPCR-ODM as well, please also see the next section.

Jackalope with Jackrabbit

Listing 8-7

```yaml
# app/config/config.yml
doctrine_phpcr:
    session:
        backend:
            type: jackrabbit
            url: http://localhost:8080/server/
        workspace: default
        username: admin
        password: admin
    # odm configuration see below
```

Jackalope with Doctrine DBAL

Listing 8-8

```yaml
# app/config/config.yml
doctrine_phpcr:
    session:
        backend:
            type: doctrinedbal
            connection: doctrine.dbal.default_connection
        workspace: default
        username: admin
        password: admin
    # odm configuration see below
```

Make sure you also configure the main `doctrine:` section for your chosen RDBMS. If you want to use a different than the default connection, configure it in the dbal section and specify it in the connection parameter. A typical example configuration is:

```
doctrine:
    dbal:
        driver: %database_driver% host: %database_host% port: %database_port% dbname: %database_name% user: %database_user% password: %database_password% charset: UTF8
```

See *Databases and Doctrine*[5] for more information.

Midgard

Listing 8-9

5. `http://symfony.com/doc/2.1/book/doctrine.html`

```yaml
# app/config/config.yml
doctrine_phpcr:
    session:
        backend:
            type: midgard2
            db_type: MySQL
            db_name: midgard2_test
            db_host: "0.0.0.0"
            db_port: 3306
            db_username: ""
            db_password: ""
            db_init: true
            blobdir: /tmp/cmf-blobs
        workspace: default
        username: admin
        password: admin
    # odm configuration see below
```

Doctrine PHPCR-ODM

Any of the above configurations will give you a valid PHPCR session. If you want to use the Object-Document manager, you need to configure it as well. The simplest is to set `auto_mapping: true` to make the PHPCR bundle recognize documents in the `<Bundle>/Document` folder and look for mappings in `<Bundle>/Resources/config/doctrine/<Document>.phpcr.xml` resp. `...yml`. Otherwise you need to manually configure the mappings section. See the *PHPCR-ODM configuration reference* for details.

Listing 8-10
```yaml
# app/config/config.yml
doctrine_phpcr:
    session:
        ...
    odm:
        auto_mapping: true
```

Setting up the content repository

Jackalope Jackrabbit

These are the steps necessary to install Apache Jackrabbit:

- Make sure you have Java Virtual Machine installed on your box. If not, you can grab one from here: *http://www.java.com/en/download/manual.jsp*[6]
- Download the latest version from the *Jackrabbit Downloads page*[7]
- Run the server. Go to the folder where you downloaded the .jar file and launch it

Listing 8-11
```
java -jar jackrabbit-standalone-*.jar
```

Going to *http://localhost:8080/*[8] should now display a Apache Jackrabbit page.

More information about *running a Jackrabbit server*[9] can be found on the Jackalope wiki.

6. http://www.java.com/en/download/manual.jsp
7. http://jackrabbit.apache.org/downloads.html
8. http://localhost:8080/
9. https://github.com/jackalope/jackalope/wiki/Running-a-jackrabbit-server

Jackalope Doctrine DBAL

Run the following commands to create the database and set up a default schema:

Listing 8-12
```
app/console doctrine:database:create
app/console doctrine:phpcr:init:dbal
```

For more information on how to configure Doctrine DBAL with Symfony2, see the *Doctrine chapter in the Symfony2 documentation*[10] and the explanations in the *PHPCR reference chapter*.

Midgard

Midgard is a C extension that implements the PHPCR API on top of a standard RDBMS.

See the *official Midgard PHPCR documentation*[11].

Registering system node types

PHPCR-ODM uses a *custom node type*[12] to track meta information without interfering with your content. There is a command that makes it trivial to register this type and the PHPCR namespace:

Listing 8-13
```
php app/console doctrine:phpcr:register-system-node-types
```

Using the ValidPhpcrOdm constraint validator

The bundle provides a `ValidPhpcrOdm` constraint validator you can use to check if your document `Id` or `Nodename` and `Parent` fields are correct :

Listing 8-14
```
<?php

namespace Acme\DemoBundle\Document;

use Doctrine\ODM\PHPCR\Mapping\Annotations as PHPCRODM;
use Doctrine\Bundle\PHPCRBundle\Validator\Constraints as Assert;

/**
 * @PHPCRODM\Document
 * @Assert\ValidPhpcrOdm
 */
class MyDocument
{
    /** @PHPCRODM\Id(strategy="parent") */
    protected $id;

    /** @PHPCRODM\Nodename */
    protected $name;

    /** @PHPCRODM\ParentDocument */
    protected $parent;

    ...
```

10. http://symfony.com/doc/current/book/doctrine.html
11. http://midgard-project.org/phpcr/
12. https://github.com/doctrine/phpcr-odm/wiki/Custom-node-type-phpcr%3Amanaged

Chapter 9
Installing and configuring inline editing

The goal of this tutorial is to install and configure the inline editing support.

This provides a solution to easily integrate with *VIE.js*[1] and *create.js*[2] to provide inline editing based on *RDFa*[3] output.

For more information for now see the documentation of the *CreateBundle*[4]

Installation

Download the bundles

Add the following to your `composer.json` file

Listing 9-1
```
"require": {
    ...
    "symfony-cmf/create-bundle": "1.0.*"
},
"scripts": {
    "post-install-cmd": [
        "Symfony\\Cmf\\Bundle\\CreateBundle\\Composer\\ScriptHandler::initSubmodules",
        ...
    ],
    "post-update-cmd": [
        "Symfony\\Cmf\\Bundle\\CreateBundle\\Composer\\ScriptHandler::initSubmodules",
        ...
    ]
},
```

And then run

1. http://viejs.org
2. http://createjs.org
3. http://rdfa.info
4. https://github.com/symfony-cmf/CreateBundle

Listing 9-2
```
php composer.phar update symfony-cmf/create-bundle
```

Initialize bundles

Next, initialize the bundles in `app/AppKernel.php` by adding them to the `registerBundle` method

Listing 9-3
```
public function registerBundles()
{
    $bundles = array(
        // ...

        new Symfony\Cmf\Bundle\CreateBundle\SymfonyCmfCreateBundle(),
        new FOS\RestBundle\FOSRestBundle(),
        new JMS\SerializerBundle\JMSSerializerBundle($this),
    );
    // ...
}
```

Configuration

Next step is to configure the bundles.

Basic configuration, add to your application configuration:

Listing 9-4
```
# app/config/config.yml
symfony_cmf_create:
    phpcr_odm: true
    map:
        '<http://rdfs.org/sioc/ns#Post>':
'Symfony\Cmf\Bundle\MultilangContentBundle\Document\MultilangStaticContent'
    image:
        model_class: Symfony\Cmf\Bundle\CreateBundle\Document\Image
        controller_class: Symfony\Cmf\Bundle\CreateBundle\Controller\PHPCRImageController
```

If you have your own documents, add them to the mapping and place the RDFa mappings in `Resources/rdf-mappings` either inside the `app` directory or inside any Bundle. The filename is the full class name including namespace with the backslashes `\\` replaced by a dot `.`.

Reference

See *CreateBundle*

Chapter 10
Creating a CMS using CMF and Sonata

The goal of this tutorial is to create a simple content management system using the CMF as well as *SonataAdminBundle*[1] and *SonataDoctrinePhpcrAdminBundle*.

Preconditions

- *Installing and configuring the CMF core*
- *Symfony SecurityBundle*[2] (required by the SonataAdminBundle default templates)

Installation

Download the bundles

Add the following to your `composer.json` file

Listing 10-1
```
"require": {
    ...
    "sonata-project/doctrine-phpcr-admin-bundle": "1.0.*",
}
```

And then run

Listing 10-2
```
php composer.phar update
```

Initialize bundles

Next, initialize the bundles in `app/AppKernel.php` by adding them to the `registerBundle` method

1. https://github.com/sonata-project/SonataAdminBundle
2. http://symfony.com/doc/master/book/security.html

Listing 10-3
```php
public function registerBundles()
{
    $bundles = array(
        // ...

        // support for the admin
        new Symfony\Cmf\Bundle\TreeBrowserBundle\SymfonyCmfTreeBrowserBundle(),
        new Sonata\jQueryBundle\SonatajQueryBundle(),
        new Sonata\BlockBundle\SonataBlockBundle(),
        new Sonata\AdminBundle\SonataAdminBundle(),
        new Sonata\DoctrinePHPCRAdminBundle\SonataDoctrinePHPCRAdminBundle(),
        new FOS\JsRoutingBundle\FOSJsRoutingBundle(),
    );
    // ...
}
```

Configuration

Add the sonata bundles to your application configuration

Listing 10-4
```yaml
# app/config/config.yml
sonata_block:
    default_contexts: [cms]
    blocks:
        sonata.admin.block.admin_list:
            contexts:   [admin]
        sonata_admin_doctrine_phpcr.tree_block:
            settings:
                id: '/cms'
            contexts:   [admin]

sonata_admin:
    templates:
        # default global templates
        ajax:    SonataAdminBundle::ajax_layout.html.twig
    dashboard:
        blocks:
            # display a dashboard block
            - { position: right, type: sonata.admin.block.admin_list }
            - { position: left, type: sonata_admin_doctrine_phpcr.tree_block }

sonata_doctrine_phpcr_admin:
    document_tree:
        Doctrine\PHPCR\Odm\Document\Generic:
            valid_children:
                - all
        Symfony\Cmf\Bundle\SimpleCmsBundle\Document\Page: ~
        Symfony\Cmf\Bundle\RoutingExtraBundle\Document\Route:
            valid_children:
                - Symfony\Cmf\Bundle\RoutingExtraBundle\Document\Route
                - Symfony\Cmf\Bundle\RoutingExtraBundle\Document\RedirectRoute
        Symfony\Cmf\Bundle\RoutingExtraBundle\Document\RedirectRoute:
            valid_children: []
        Symfony\Cmf\Bundle\MenuBundle\Document\MenuNode:
            valid_children:
                - Symfony\Cmf\Bundle\MenuBundle\Document\MenuNode
```

```
                - Symfony\Cmf\Bundle\MenuBundle\Document\MultilangMenuNode
        Symfony\Cmf\Bundle\MenuBundle\Document\MultilangMenuNode:
            valid_children:
                - Symfony\Cmf\Bundle\MenuBundle\Document\MenuNode
                - Symfony\Cmf\Bundle\MenuBundle\Document\MultilangMenuNode

fos_js_routing:
    routes_to_expose:
        - admin_sandbox_main_editablestaticcontent_create
        - admin_sandbox_main_editablestaticcontent_delete
        - admin_sandbox_main_editablestaticcontent_edit
        - admin_bundle_menu_menunode_create
        - admin_bundle_menu_menunode_delete
        - admin_bundle_menu_menunode_edit
        - admin_bundle_menu_multilangmenunode_create
        - admin_bundle_menu_multilangmenunode_delete
        - admin_bundle_menu_multilangmenunode_edit
        - admin_bundle_content_multilangstaticcontent_create
        - admin_bundle_content_multilangstaticcontent_delete
        - admin_bundle_content_multilangstaticcontent_edit
        - admin_bundle_routingextra_route_create
        - admin_bundle_routingextra_route_delete
        - admin_bundle_routingextra_route_edit
        - admin_bundle_simplecms_page_create
        - admin_bundle_simplecms_page_delete
        - admin_bundle_simplecms_page_edit
        - symfony_cmf_tree_browser.phpcr_children
        - symfony_cmf_tree_browser.phpcr_move
        - sonata.admin.doctrine_phpcr.phpcrodm_children
        - sonata.admin.doctrine_phpcr.phpcrodm_move
```

Add route in to your routing configuration

Listing 10-5

```
# app/config/routing.yml
admin:
    resource: '@SonataAdminBundle/Resources/config/routing/sonata_admin.xml'
    prefix: /admin

sonata_admin:
    resource: .
    type: sonata_admin
    prefix: /admin

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

symfony_cmf_tree:
    resource: .
    type: 'symfony_cmf_tree'
```

Sonata Assets

Listing 10-6

```
app/console assets:install --symlink
```

Finally

Now Sonata is configured to work with the PHPCR you can access the dashboard using via /admin/dashboard in your site.

Tree Problems

If you have not yet added anything to the content repository, the tree view will not load as it cannot find a root node. To fix this, load some data as fixtures by following this doc:

- *Using the BlockBundle and ContentBundle with PHPCR*

Chapter 11

Using the BlockBundle and ContentBundle with PHPCR

The goal of this tutorial is to demonstrate how the CMF *BlockBundle* and *ContentBundle* can be used as stand-alone components, and to show how they fit into the PHPCR.

This tutorial demonstrates the simplest possible usage, to get you up and running quickly. Once you are familiar with basic usage, the in-depth documentation of both bundles will help you to adapt these basic examples to serve more advanced use cases.

We will begin with using only BlockBundle, with content blocks linked directly into the PHPCR. Next, we will introduce the ContentBundle to show how it can represent content pages containing blocks.

Although not a requirement for using BlockBundle or ContentBundle, this tutorial will also make use of *DoctrineFixturesBundle*[1]. This is because it provides an easy way to load in some test content.

Preconditions

- *Installation of Symfony2*[2] (2.1.x)
- *Installing and configuring Doctrine PHPCR-ODM*

This tutorial is based on using PHPCR-ODM set up with Jackalope, Doctrine DBAL and a MySQL database. It should be easy to adapt this to work with one of the other PHPCR options documented in *Installing and configuring Doctrine PHPCR-ODM*.

1. http://symfony.com/doc/current/bundles/DoctrineFixturesBundle/index.html
2. http://symfony.com/doc/2.1/book/installation.html

Create and configure the database

You can use an existing database, or create one now to help you follow this tutorial. For a new database, run these commands in MySQL:

Listing 11-1
```
CREATE DATABASE symfony DEFAULT CHARACTER SET utf8 DEFAULT COLLATE utf8_general_ci;
CREATE USER 'symfony'@'localhost' IDENTIFIED BY 'UseABetterPassword';
GRANT ALL ON symfony.* TO 'symfony'@'localhost';
```

Your `parameters.yml` file needs to match the above, for example:

Listing 11-2
```
# app/config/parameters.yml
parameters:
    database_driver:   pdo_mysql
    database_host:     localhost
    database_port:     ~
    database_name:     symfony
    database_user:     symfony
    database_password: UseABetterPassword
```

Configure the Doctrine PHPCR component

If you have followed *Installing and configuring Doctrine PHPCR-ODM*, you can skip this section.

You need to install the PHPCR-ODM components. Add the following to your `composer.json` file:

Listing 11-3
```
"require": {
    ...
    "jackalope/jackalope-jackrabbit": "1.0.*",
    "jackalope/jackalope-doctrine-dbal": "dev-master",
    "doctrine/phpcr-bundle": "1.0.*",
    "doctrine/phpcr-odm": "1.0.*"
}
```

To install the above, run:

Listing 11-4
```
php composer.phar update
```

In your `config.yml` file, add following configuration for *doctrine_phpcr*:

Listing 11-5
```
# app/config/config.yml
doctrine_phpcr:
    session:
        backend:
            type: doctrinedbal
            connection: doctrine.dbal.default_connection
        workspace: default
    odm:
        auto_mapping: true
```

Add the following line to the registerBundles() method in `AppKernel.php`:

Listing 11-6
```php
// app/AppKernel.php

public function registerBundles()
{
    $bundles = array(
        // ...
        new Doctrine\Bundle\PHPCRBundle\DoctrinePHPCRBundle(),
    );

    // ...
}
```

Add the following line to your `autoload.php` file, immediately after the last `AnnotationRegistry::registerFile` line:

Listing 11-7
```php
// app/autoload.php

// ...
AnnotationRegistry::registerFile(__DIR__.'/../vendor/doctrine/phpcr-odm/lib/Doctrine/ODM/
PHPCR/Mapping/Annotations/DoctrineAnnotations.php');
// ...
```

Create the database schema and register the PHPCR node types using the following console commands:

Listing 11-8
```
php app/console doctrine:phpcr:init:dbal
php app/console doctrine:phpcr:register-system-node-types
```

Now you should have a number of tables in your MySQL database with the `phpcr_` prefix.

Install the needed Symfony CMF components

Add the following to `composer.json`:

Listing 11-9
```
"require": {
    ...
    "symfony-cmf/block-bundle": "dev-master"
}
```

To install the above dependencies, run:

Listing 11-10
```
php composer.phar update
```

Add the following lines to AppKernel.php:

Listing 11-11
```php
// app/AppKernel.php

public function registerBundles()
{
    $bundles = array(
        // ...
        new Sonata\BlockBundle\SonataBlockBundle(),
        new Symfony\Cmf\Bundle\BlockBundle\SymfonyCmfBlockBundle(),
```

```
        );

        // ...
    }
```

SonataBlockBundle is a dependency of the CMF BlockBundle and needs to be configured. Add the following to your `config.yml`:

Listing 11-12
```
# app/config/config.yml
sonata_block:
    default_contexts: [cms]
```

Install DoctrineFixturesBundle

As mentioned at the start, this is not a requirement for BlockBundle or ContentBundle; nevertheless it is a good way to manage example or default content.

Add the following to `composer.json`:

Listing 11-13
```
"require": {
    ...
    "doctrine/doctrine-fixtures-bundle": "dev-master"
}
```

To install the above dependencies, run:

Listing 11-14
```
php composer.phar update
```

Add the following line to the registerBundles() method in `AppKernel.php`:

Listing 11-15
```
// app/AppKernel.php

public function registerBundles()
{
    $bundles = array(
        // ...
        new Doctrine\Bundle\FixturesBundle\DoctrineFixturesBundle(),
    );

    // ...
}
```

Loading fixtures

Based on the *DoctrineFixturesBundle documentation*[3], you will need to create a fixtures class.

3. http://symfony.com/doc/current/bundles/DoctrineFixturesBundle/index.html

To start with, create a `DataFixtures` directory inside your own bundle (e.g. "MainBundle"), and inside there, create a directory named `PHPCR`. As you follow the examples further below, the DoctrineFixturesBundle will automatically load the fixtures classes placed here.

Within a fixtures loader, an example of creating a content block might look like this:

Listing 11-16
```php
$myBlock = new SimpleBlock();
$myBlock->setParentDocument($parentPage);
$myBlock->setName('sidebarBlock');
$myBlock->setTitle('My first block');
$myBlock->setContent('Hello block world!');

$documentManager->persist($myBlock);
```

The above on its own will not be enough however, because there is no parent (`$parentPage`) to link the blocks to. There are several possible options that you can use as the parent:

- Link the blocks directly to the root document (not shown)
- Create a document from the PHPCR bundle (shown below using the `Generic` document type)
- Create a document from the CMF ContentBundle (shown below using `StaticContent` document type)

Using the PHPCR

To store a CMF block directly in the PHPCR, create the following class inside your `DataFixtures/PHPCR` directory:

Listing 11-17
```php
<?php
// src/Acme/MainBundle/DataFixtures/PHPCR/LoadBlockWithPhpcrParent.php
namespace Acme\MainBundle\DataFixtures\ORM;

use Doctrine\Common\DataFixtures\AbstractFixture;
use Doctrine\Common\Persistence\ObjectManager;
use Doctrine\ODM\PHPCR\Document\Generic;
use Symfony\Component\DependencyInjection\ContainerAwareInterface;
use Symfony\Component\DependencyInjection\ContainerInterface;
use Symfony\Cmf\Bundle\BlockBundle\Document\SimpleBlock;

class LoadBlockWithPhpcrParent extends AbstractFixture implements ContainerAwareInterface
{
    public function load(ObjectManager $manager)
    {
        // Get the root document from the PHPCR
        $rootDocument = $manager->find(null, '/');

        // Create a generic PHPCR document under the root, to use as a kind of category
for the blocks
        $document = new Generic();
        $document->setParent($rootDocument);
        $document->setNodename('blocks');
        $manager->persist($document);

        // Create a new SimpleBlock (see http://symfony.com/doc/master/cmf/bundles/
block.html#block-types)
        $myBlock = new SimpleBlock();
        $myBlock->setParentDocument($document);
        $myBlock->setName('testBlock');
```

```
        $myBlock->setTitle('CMF BlockBundle only');
        $myBlock->setContent('Block from CMF BlockBundle, parent from the PHPCR (Generic
document).');
        $manager->persist($myBlock);

        // Commit $document and $block to the database
        $manager->flush();
    }

    public function setContainer(ContainerInterface $container = null)
    {
        $this->container = $container;
    }
}
```

This class loads an example content block using the CMF BlockBundle (without needing any other CMF bundle). To ensure the block has a parent in the repository, the loader also creates a `Generic` document named 'blocks' within the PHPCR.

Now load the fixtures using the console:

Listing 11-18
```
php app/console doctrine:phpcr:fixtures:load
```

The content in your database should now look something like this:

Listing 11-19
```
SELECT path, parent, local_name FROM phpcr_nodes;
```

path	parent	local_name
/		
/blocks	/	blocks
/blocks/testBlock	/ blocks	testBlock

Using the CMF ContentBundle

Follow this example to use both the CMF Block and Content components together.

The ContentBundle also requires RoutingExtraBundle, so to save time you can install both together. Add the following to `composer.json`:

Listing 11-20
```
"require": {
    ...
    "symfony-cmf/content-bundle": "dev-master",
    "symfony-cmf/routing-extra-bundle": "dev-master"
}
```

Install as before:

Listing 11-21
```
php composer.phar update
```

Add the following line to AppKernel.php:

Listing 11-22

```php
// app/AppKernel.php

public function registerBundles()
{
    $bundles = array(
        // ...
        new Symfony\Cmf\Bundle\ContentBundle\SymfonyCmfContentBundle(),
    );

    // ...
}
```

Now you should have everything needed to load a sample content page with a sample block, so create the `LoadBlockWithCmfParent.php` class:

Listing 11-23

```php
<?php
// src/Acme/Bundle/MainBundle/DataFixtures/PHPCR/LoadBlockWithCmfParent.php
namespace Acme\MainBundle\DataFixtures\PHPCR;

use Doctrine\Common\DataFixtures\AbstractFixture;
use Doctrine\Common\Persistence\ObjectManager;
use Symfony\Component\DependencyInjection\ContainerAwareInterface;
use Symfony\Component\DependencyInjection\ContainerInterface;
use PHPCR\Util\NodeHelper;
use Symfony\Cmf\Bundle\BlockBundle\Document\SimpleBlock;
use Symfony\Cmf\Bundle\ContentBundle\Document\StaticContent;

class LoadBlockWithCmfParent extends AbstractFixture implements ContainerAwareInterface
{
    public function load(ObjectManager $manager)
    {
        // Get the base path name to use from the configuration
        $session = $manager->getPhpcrSession();
        $basepath = $this->container->getParameter('symfony_cmf_content.static_basepath');

        // Create the path in the repository
        NodeHelper::createPath($session, $basepath);

        // Create a new document using StaticContent from the CMF ContentBundle
        $document = new StaticContent();
        $document->setPath($basepath . '/blocks');
        $manager->persist($document);

        // Create a new SimpleBlock (see http://symfony.com/doc/master/cmf/bundles/
block.html#block-types)
        $myBlock = new SimpleBlock();
        $myBlock->setParentDocument($document);
        $myBlock->setName('testBlock');
        $myBlock->setTitle('CMF BlockBundle and ContentBundle');
        $myBlock->setContent('Block from CMF BlockBundle, parent from CMF ContentBundle
(StaticContent).');
        $manager->persist($myBlock);

        // Commit $document and $block to the database
        $manager->flush();
    }

    public function setContainer(ContainerInterface $container = null)
```

```
44      {
45          $this->container = $container;
        }
    }
```

This class creates an example content page using the CMF ContentBundle. It then loads our example block as before, using the new content page as its parent.

By default, the base path for the content is /cms/content/static. To show how it can be configured to any path, add the following, optional entry to your config.yml:

Listing 11-24
```
# app/config/config.yml
symfony_cmf_content:
    static_basepath: /content
```

Now it should be possible to load in the above fixtures:

Listing 11-25
```
php app/console doctrine:phpcr:fixtures:load
```

All being well, the content in your database should look something like this (if you also followed the `LoadBlockWithPhpcrParent` example, you should still have two `/blocks` entries as well):

Listing 11-26
```
SELECT path, parent, local_name FROM phpcr_nodes;
```

path	parent	local_name
/		
/content	/	content
/content/blocks	/content	blocks
/content/blocks/testBlock	/content/blocks	testBlock

Rendering the blocks

This is handled by the Sonata BlockBundle. `sonata_block_render` is already registered as a Twig extension by including `SonataBlockBundle` in `AppKernel.php`. Therefore, you can render any block within any template by referring to its path.

The following code shows the rendering of both `testBlock` instances from the examples above. If you only followed one of the examples, make sure to only include that block:

Listing 11-27
```
{# src/Acme/Bundle/MainBundle/resources/views/Default/index.html.twig #}

{# include this if you followed the BlockBundle with PHPCR example #}
{{ sonata_block_render({
    'name': '/blocks/testBlock'
}) }}

{# include this if you followed the BlockBundle with ContentBundle example #}
{{ sonata_block_render({
    'name': '/content/blocks/testBlock'
}) }}
```

Now your index page should show the following (assuming you followed both examples):

Listing 11-28
```
CMF BlockBundle only
Block from CMF BlockBundle, parent from the PHPCR (Generic document).

CMF BlockBundle and ContentBundle
Block from CMF BlockBundle, parent from CMF ContentBundle (StaticContent).
```

This happens when a block is rendered, see the .. index:: BlockBundle for more details:

- a document is loaded based on the name
- if caching is configured, the cache is checked and content is returned if found
- each block document also has a block service, the execute method of it is called:
    - you can put here logic like in a controller
    - it calls a template
    - the result is a Response object

A block can also be configured using settings, this allows you to create more advanced blocks and reuse it. The default settings are configured in the block service and can be altered in the twig helper and the block document. An example is an rss reader block, the url and title are stored in the settings of the block document, the maximum amount of items to display is specified when calling `sonata_block_render`.

Next steps

You should now be ready to use the BlockBundle and/or the ContentBundle in your application, or to explore the other available CMF bundles.

- See the *BlockBundle* and *ContentBundle* documentation to learn about more advanced usage of these bundles
- To see a better way of loading fixtures, look at the *fixtures in the CMF Sandbox*[4]
- Take a look at the *PHPCR Tutorial*[5] for a better understanding of the underlying content repository

Troubleshooting

If you run into problems, it might be easiest to start with a fresh Symfony2 installation. You can also try running and modifying the code in the external *CMF Block Sandbox*[6] working example.

Doctrine configuration

If you started with the standard Symfony2 distribution (version 2.1.x), this should already be configured correctly in your `config.yml` file. If not, try using the following section:

Listing 11-29
```
# app/config/config.yml
doctrine:
    dbal:
        driver:   "%database_driver%"
```

4. https://github.com/symfony-cmf/cmf-sandbox/tree/master/src/Sandbox/MainBundle/DataFixtures/PHPCR
5. https://github.com/phpcr/phpcr-docs/blob/master/tutorial/Tutorial.md
6. https://github.com/fazy/cmf-block-sandbox

```
        host:     "%database_host%"
        port:     "%database_port%"
        dbname:   "%database_name%"
        user:     "%database_user%"
        password: "%database_password%"
        charset:  UTF8
    orm:
        auto_generate_proxy_classes: "%kernel.debug%"
        auto_mapping: true
```

"No commands defined" when loading fixtures

Listing 11-30
```
[InvalidArgumentException]
There are no commands defined in the "doctrine:phpcr:fixtures" namespace.
```

Make sure AppKernel.php contains the following lines:

Listing 11-31
```
new Doctrine\Bundle\FixturesBundle\DoctrineFixturesBundle(),
new Doctrine\Bundle\PHPCRBundle\DoctrinePHPCRBundle(),
```

"You did not configure a session"

Listing 11-32
```
[InvalidArgumentException]
You did not configure a session for the document managers
```

Make sure you have the following in your app/config.yml:

Listing 11-33
```
doctrine_phpcr:
    session:
        backend:
            type: doctrinedbal
            connection: doctrine.dbal.default_connection
        workspace: default
    odm:
        auto_mapping: true
```

"Annotation does not exist"

Listing 11-34
```
[Doctrine\Common\Annotations\AnnotationException]
[Semantical Error] The annotation "@Doctrine\ODM\PHPCR\Mapping\Annotations\Document" in
class Doctrine\ODM\PHPCR\Document\Generic does not exist, or could not be auto-loaded.
```

Make sure you add this line to your app/autoload.php (immediately after the AnnotationRegistry::registerLoader line):

Listing 11-35
```
AnnotationRegistry::registerFile(__DIR__.'/../vendor/doctrine/phpcr-odm/lib/Doctrine/ODM/
PHPCR/Mapping/Annotations/DoctrineAnnotations.php');
```

SimpleBlock class not found

Listing 11-36
```
[Doctrine\Common\Persistence\Mapping\MappingException]
The class 'Symfony\Cmf\Bundle\BlockBundle\Document\SimpleBlock' was not found in the chain
configured namespaces Doctrine\ODM\PHPCR\Document, Sonata\UserBundle\Document,
FOS\UserBundle\Document
```

Make sure the CMF BlockBundle is installed and loaded in app/AppKernel.php:

Listing 11-37
```
new Symfony\Cmf\Bundle\BlockBundle\SymfonyCmfBlockBundle(),
```

RouteAwareInterface not found

Listing 11-38
```
Fatal error: Interface 'Symfony\Cmf\Component\Routing\RouteAwareInterface' not found in
/var/www/your-site/vendor/symfony-cmf/content-bundle/Symfony/Cmf/Bundle/ContentBundle/
Document/StaticContent.php on line 15
```

If you are using ContentBundle, make sure you have also installed the RoutingExtraBundle:

Listing 11-39
```
// composer.json
"symfony-cmf/routing-extra-bundle": "dev-master"
```

...and install:

Listing 11-40
```
php composer.phar update
```

Chapter 12
BlockBundle

The *BlockBundle*[1] provides integration with SonataBlockBundle. It assists you in managing fragments of contents, so-called blocks. What the BlockBundle does is similar to what Twig does, but for blocks that are persisted in a DB. Thus, the blocks can be made editable for an editor. Also the BlockBundle provides the logic to determine which block should be rendered on which pages.

The BlockBundle does not provide an editing functionality for blocks itself. However, you can find examples on how making blocks editable in the *Symfony CMF Sandbox*[2].

Dependencies

This bundle is based on the *SonataBlockBundle*[3]

Configuration

The configuration key for this bundle is `symfony_cmf_block`

Listing 12-1
```yaml
# app/config/config.yml
symfony_cmf_block:
    document_manager_name:  default
```

Block Document

Before you can render a block, you need to create a data object representing your block in the repository. You can do so with the following code snippet (Note that `$parentPage` needs to be an instance of a page defined by the *ContentBundle*[4]):

1. https://github.com/symfony-cmf/BlockBundle#readme
2. https://github.com/symfony-cmf/cmf-sandbox
3. https://github.com/sonata-project/SonataBlockBundle
4. https://github.com/symfony-cmf/ContentBundle

Listing 12-2
```php
$myBlock = new SimpleBlock();
$myBlock->setParentDocument($parentPage);
$myBlock->setName('sidebarBlock');
$myBlock->setTitle('My first block');
$myBlock->setContent('Hello block world!');

$documentManager->persist($myBlock);
```

Note the 'sidebarBlock' is the identifier we chose for the block. Together with the parent document of the block, this makes the block unique. The other properties are specific to `Symfony\Cmf\Bundle\BlockBundle\Document\SimpleBlock`.

The simple block is now ready to be rendered, see *Block rendering*.

Always make sure you implement the interface `Sonata\BlockBundle\Model\BlockInterface` or an existing block document like `Symfony\Cmf\Bundle\BlockBundle\Document\BaseBlock`.

Block Service

If you look inside the SimpleBlock class, you will notice the method `getType`. This defines the name of the block service that processes the block when it is rendered.

A block service contains:

- an execute method
- default settings
- form configuration
- cache configuration
- js and css assets to be loaded
- a load method

Take a look at the block services in `Symfony\Cmf\Bundle\BlockBundle\Block` to see some examples.

Always make sure you implement the interface `Sonata\BlockBundle\Block\BlockServiceInterface` or an existing block service like `Sonata\BlockBundle\Block\BaseBlockService`.

execute method

This contains `controller` logic. Merge the default settings in this method if you would like to use them:

Listing 12-3
```php
// ...
if ($block->getEnabled()) {
    // merge settings
    $settings = array_merge($this->getDefaultSettings(), $block->getSettings());

    $feed = false;
```

```
7       if ($settings['url']) {
8           $feed = $this->feedReader->import($block);
9       }
10
11      return $this->renderResponse($this->getTemplate(), array(
12          'feed' => $feed,
13          'block' => $block,
14          'settings' => $settings
15      ), $response);
16  }
17  // ...
```

If you have much logic to be used, you can move that to a specific service and inject it in the block service. Then use this specific service in the execute method.

default settings

The method `getDefaultSettings` contains the default settings of a block. Settings can be altered on multiple places afterward, it cascades like this:

- default settings are stored in the block service
- settings can be altered through template helpers:

Listing 12-4
```
{{ sonata_block_render({
  'type': 'acme_main.block.rss',
  'settings': {
      'title': 'Symfony2 CMF news',
      'url': 'http://cmf.symfony.com/news.rss'
  }
}) }}
```

- and settings can also be altered in a block document, the advantage is that settings are stored in PHPCR and allows to implement a frontend or backend UI to change some or all settings

form configuration

The methods `buildEditForm` and `buildCreateForm` contain form configuration for editing using a frontend or backend UI. The method `validateBlock` contains the validation configuration.

cache configuration

The method `getCacheKeys` contains cache keys to be used for caching the block.

js and css

The methods `getJavascripts` and `getStylesheets` can be used to define js and css assets. Use the twig helpers `sonata_block_include_javascripts` and `sonata_block_include_stylesheets` to render them.

Listing 12-5
```
{{ sonata_block_include_javascripts() }}
{{ sonata_block_include_stylesheets() }}
```

This will output the js and css of all blocks loaded in the service container of your application.

load method

The method `load` can be used to load additional data. It is called each time a block is rendered before the `execute` method is called.

Block rendering

To have the block from the example of the Block Document section actually rendered, you just add the following code to your Twig template:

Listing 12-6
```
1 {{ sonata_block_render({'name': 'sidebarBlock'}) }}
```

This will make the BlockBundle rendering the according block on every page that has a block named 'sidebarBlock'. Of course, the actual page needs to be rendered by the template that contains the snippet above.

This happens when a block is rendered, see the separate sections for more details:

- a document is loaded based on the name
- if caching is configured, the cache is checked and content is returned if found
- each block document also has a block service, the execute method of it is called:
    - you can put here logic like in a controller
    - it calls a template
    - the result is a Response object

Block types

The BlockBundle comes with four general purpose blocks:

- SimpleBlock: A simple block with nothing but a title and a field of hypertext. This would usually be what an editor edits directly, for example contact information
- ContainerBlock: A block that contains 0 to n child blocks
- ReferenceBlock: A block that references a block stored somewhere else in the content tree. For example you might want to refer parts of the contact information from the homepage
- ActionBlock: A block that calls a Symfony2 action. "Why would I use this instead of directly calling the action from my template?", you might wonder. Well imagine the following case: You provide a block that renders teasers of your latest news. However, there is no rule where they should appear. Instead, your customer wants to decide himself on what pages this block is to be displayed. Providing an according ActionBlock, you allow your customer to do so without calling you to change some templates (over and over again!).

Create your own blocks

Follow these steps to create a block:

- create a block document

- create a block service and declare it (optional)
- create a data object representing your block in the repository, see *Block Document*
- render the block, see *Block rendering*

Lets say you are working on a project where you have to integrate data received from several RSS feeds. Of course you could create an ActionBlock for each of these feeds, but wouldn't this be silly? In fact all those actions would look similar: Receive data from a feed, sanitize it and pass the data to a template. So instead you decide to create your own block, the RSSBlock.

Create a block document

The first thing you need is an document that contains the data. It is recommended to extend `Symfony\Cmf\Bundle\BlockBundle\Document\BaseBlock` contained in this bundle (however you are not forced to do so, as long as you implement `Sonata\BlockBundle\Model\BlockInterface`). In your document, you need to define the `getType` method which just returns 'acme_main.block.rss'.

Listing 12-7
```php
namespace Acme\MainBundle\Document;

use Doctrine\ODM\PHPCR\Mapping\Annotations as PHPCRODM;
use Symfony\Cmf\Bundle\BlockBundle\Document\BaseBlock;

/**
 * Rss Block
 *
 * @PHPCRODM\Document(referenceable=true)
 */
class RssBlock extends BaseBlock
{
    public function getType()
    {
        return 'acme_main.block.rss';
    }
}
```

Create a block service

You could choose to use a an already existing block service because the configuration and logic already satisfy your needs. For our rss block we create a service that knows how to handle RSSBlocks:

- the method `getDefaultSettings` configures a title, url and the maximum amount of items

Listing 12-8
```php
// ...
public function getDefaultSettings()
{
    return array(
        'url'      => false,
        'title'    => 'Insert the rss title',
        'maxItems' => 10,
    );
}
// ..
```

- the execute method passes the settings to an rss reader service and forwards the feed items to a template, see *execute method*

Make sure you implement the interface `Sonata\BlockBundle\Block\BlockServiceInterface` or an existing block service like `Sonata\BlockBundle\Block\BaseBlockService`.

Define the service in a config file. It is important to tag your BlockService with 'sonata.block', otherwise it will not be known by the Bundle.

Listing 12-9
```yaml
# services.yml
acme_main.rss_reader:
    class: Acme\MainBundle\Feed\SimpleReader

sandbox_main.block.rss:
    class: Acme\MainBundle\Block\RssBlockService
    arguments:
        - "acme_main.block.rss"
        - "@templating"
        - "@sonata.block.renderer"
        - "@acme_main.rss_reader"
    tags:
        - {name: "sonata.block"}
```

Examples

You can find example usages of this bundle in the *Symfony CMF Sandbox*[5]. Have a look at the BlockBundle in the Sandbox. It also shows you how to make blocks editable using the *CreateBundle*[6].

Relation to Sonata Block Bundle

The BlockBundle is based on the *SonataBlockBundle*[7]. It replaces components of the bundle where needed to be compatible with PHPCR.

The following picture shows where we use our own components (blue):

../../_images/classdiagram.jpg

5. https://github.com/symfony-cmf/cmf-sandbox
6. https://github.com/symfony-cmf/CreateBundle
7. https://github.com/sonata-project/SonataBlockBundle

Chapter 13
ContentBundle

This bundle provides a document for static content and the controller to render it.
For an introduction see the *Content* page in the "Getting started" section.

Configuration

The configuration key for this bundle is `symfony_cmf_content`

Listing 13-1
```yaml
# app/config/config.yml
symfony_cmf_content:
    admin_class:          ~
    document_class:       ~
    default_template:     ~
    content_basepath:     /cms/content
    static_basepath:      /cms/content/static
    use_sonata_admin:     auto
    multilang:                # the whole multilang section is optionnal
        admin_class:          ~
        document_class:       ~
        use_sonata_admin:     auto
        locales:              [] # if you use multilang, you have to define at least one locale
```

Chapter 14
CoreBundle

This is the *CoreBundle*[1] for the Symfony2 content management framework. This bundle provides common functionality, helpers and utilities for the other CMF bundles.

One of the provided features is an interface and implementation of a publish workflow checker with an accompanying interface that models can implement that want to support this checker.

Furthermore it provides a twig helper exposing several useful functions for twig templates to interact with PHPCR-ODM documents.

Configuration

Listing 14-1
```yaml
# app/config/config.yml
symfony_cmf_core:
    document_manager: default
    role``: IS_AUTHENTICATED_ANONYMOUSLY # used by the publish workflow checker
```

Publish workflow checker

The Bundle provides a `symfony_cmf_core.publish_workflow_checker` service which implements `PublishWorkflowCheckerInterface`. This interface defines a single method `checkIsPublished()`.

Listing 14-2
```php
$publishWorkflowChecker = $container->get('symfony_cmf_core.publish_workflow_checker');
$ignoreRole = false // if to ignore the role when deciding if to consider the document as
published
if ($publishWorkflowChecker->checkIsPublished($document, $ignoreRole)) {
    ..
}
```

1. https://github.com/symfony-cmf/CoreBundle#readme

Twig extension

Implements the following functions:

- `cmf_find`: find the document for the provided path and class
- `cmf_is_published`: checks if the provided document is published
- `cmf_prev`: returns the previous published document by examining the child nodes of the parent of the provided
- `cmf_next`: returns the next published document by examining the child nodes of the parent of the provided
- `cmf_children`: returns an array of all the children documents of the provided documents that are published
- `cmf_document_locales`: gets the locales of the provided document

Listing 14-3

```
{% set page = cmf_find('/some/path') %}

{% if cmf_is_published(page) %}
    {% set prev = cmf_prev(page) %}
    {% if prev %}
        <a href="{{ path(prev) }}">prev</a>
    {% endif %}

    {% set next = cmf_next(page) %}
    {% if next %}
        <span style="float: right; padding-right: 40px;"><a href="{{ path(next)
}}">next</a></span>
    {%  endif %}

    {% for news in cmf_children(page)|reverse %}
        <li><a href="{{ path(news) }}">{{ news.title }}</a> ({{ news.publishStartDate |
date('Y-m-d')  }})</li>
    {% endfor %}

    {% if 'de' in cmf_document_locales(page) %}
        <a href="{{ path(app.request.attributes.get('_route'),
app.request.attributes.get('_route_params')|merge(app.request.query.all)|merge({'_locale':
'de'})) }}">DE</a>
    {%  endif %}
    {% if 'fr' in cmf_document_locales(page) %}
        <a href="{{ path(app.request.attributes.get('_route'),
app.request.attributes.get('_route_params')|merge(app.request.query.all)|merge({'_locale':
'fr'})) }}">DE</a>
    {%  endif %}
{%  endif %}
```

Chapter 15
CreateBundle

The *CreateBundle*[1] integrates create.js and the createphp helper library into Symfony2.

Create.js is a comprehensive web editing interface for Content Management Systems. It is designed to provide a modern, fully browser-based HTML5 environment for managing content. Create can be adapted to work on almost any content management backend. See *http://createjs.org/*[2]

Createphp is a PHP library to help with RDFa annotating your documents/entities. See *https://github.com/flack/createphp*[3] for documentation on how it works.

Dependencies

This bundle includes create.js (which bundles all its dependencies like jquery, vie, hallo, backbone etc) as a git submodule. Do not forget to add the composer script handler to your composer.json as described below.

PHP dependencies are managed through composer. We use createphp as well as AsseticBundle, FOSRestBundle and by inference also JmsSerializerBundle. Make sure you instantiate all those bundles in your kernel and properly configure assetic.

Installation

This bundle is best included using Composer.

Edit your project composer file to add a new require for symfony-cmf/create-bundle. Then create a scripts section or add to the existing one:

Listing 15-1

```
1 {
2     "scripts": {
3         "post-install-cmd": [
4
```

1. https://github.com/symfony-cmf/CreateBundle
2. http://createjs.org/
3. https://github.com/flack/createphp

```
5  "Symfony\\Cmf\\Bundle\\SymfonyCmfCreateBundle\\Composer\\ScriptHandler::initSubmodules",
6              ...
7          ],
8          "post-update-cmd": [
9
10 "Symfony\\Cmf\\Bundle\\SymfonyCmfCreateBundle\\Composer\\ScriptHandler::initSubmodules",
11             ...
12         ]
       }
   }
```

Add this bundle (and its dependencies, if they are not already there) to your application's kernel:

Listing 15-2

```
1  // application/ApplicationKernel.php
2  public function registerBundles()
3  {
4      return array(
5          // ...
6          new Symfony\Bundle\AsseticBundle\AsseticBundle(),
7          new JMS\SerializerBundle\JMSSerializerBundle($this),
8          new FOS\RestBundle\FOSRestBundle(),
9          new Symfony\Cmf\Bundle\SymfonyCmfCreateBundle\SymfonyCmfCreateBundle(),
10         // ...
11     );
12 }
```

You also need to configure FOSRestBundle to handle json:

Listing 15-3

```
1  fos_rest:
2      view:
3          formats:
4              json: true
```

Concept

Createphp uses RDFa metadata about your domain classes, much like doctrine knows the metadata how an object is stored in the database. The metadata is modelled by the type class and can come from any source. Createphp provides metadata drivers that read XML, php arrays and one that just introspects objects and creates non-semantical metadata that will be enough for create.js to edit.

The RdfMapper is used to translate between your storage layer and createphp. It is passed the domain object and the relevant metadata object.

With the metadata and the twig helper, the content is rendered with RDFa annotations. create.js is loaded and enables editing on the entities. Save operations happen in ajax calls to the backend.

The REST controller handles those ajax calls, and if you want to be able to upload images, an image controller saves uploaded images and tells the image location.

Configuration

Listing 15-4

```yaml
# app/config/config.yml
symfony_cmf_create:
    # metadata loading

    # directory list to look for metadata
    rdf_config_dirs:
        - "%kernel.root_dir%/Resources/rdf-mappings"
    # look for mappings in <Bundle>/Resources/rdf-mappings
    # auto_mapping: true

    # use a different class for the REST handler
    # rest_controller_class: FQN\Classname
    # enable hallo development mode (see the end of this chapter)
    # use_coffee: false

    # image handling
    image:
        model_class: ~
        controller_class: ~

    # access check role for js inclusion, default REST and image controllers
    # role: IS_AUTHENTICATED_ANONYMOUSLY

    # enable the doctrine PHPCR-ODM mapper
    phpcr_odm: true

    # mapping from rdf type name => class name used when adding items to collections
    map:
        rdfname: FQN\Classname

    # stanbol url for semantic enhancement, otherwise defaults to the demo install
    # stanbol_url: http://dev.iks-project.eu:8081

    # fix the Hallo editor toolbar on top of the page
    # fixed_toolbar: true

    # RDFa types used for elements to be edited in plain text
    # plain_text_types: ['dcterms:title']

    # RDFa types for which to create the corresponding routes after
    # content of these types has been added with Create.js. This is
    # not necessary with the SimpleCmsBundle, as the content and the
    # routes are in the same repository tree.
    # create_routes_types: ['http://schema.org/NewsArticle']
```

The provided javascript file configures create.js and the hallo editor. It enables some plugins like the tag editor to edit `skos:related` collections of attributes. We hope to add some configuration options to tweak the configuration of create.js but you can also use the file as a template and do your own if you need larger customizations.

Metadata

createphp needs metadata information for each class of your domain model. By default, the create bundle uses the XML metadata driver and looks for metadata in the enabled bundles at <Bundle>/Resources/rdf-mappings. If you use a bundle that has no RDFa mapping, you can specify a list of rdf_config_dirs that will additionally be checked for metadata.

See the *documentation of createphp*[4] for the format of the XML metadata format.

Access control

If you use the default REST controller, everybody can edit content once you enabled the create bundle. To restrict access, specify a role other than the default IS_AUTHENTICATED_ANONYMOUSLY to the bundle. If you specify a different role, create.js will only be loaded if the user has that role and the REST handler (and image handler if enabled) will check the role.

If you need more fine grained access control, look into the mapper `isEditable` method. You can extend the mapper you use and overwrite isEditable to answer whether the passed domain object is editable.

Image Handling

Enable the default simplistic image handler with the image > model_class | controller_class settings. This image handler just throws images into the PHPCR-ODM repository and also serves them in requests.

If you need different image handling, you can either overwrite image.model_class and/or image.controller_class, or implement a custom ImageController and override the `symfony_cmf_create.image.controller` service with it.

Mapping requests to objects

For now, the bundle only provides a service to map to doctrine PHPCR-ODM. Enable it by setting *phpcr_odm* to true. If you need something else, you need to provide a service *symfony_cmf_create.object_mapper*. (If you need a wrapper for doctrine ORM, look at the mappers in the createphp library and do a pull request on that library, and another one to expose the ORM mapper as service in the create bundle).

Also note that createphp would support different mappers for different RDFa types. If you need that, dig into the createphp and create bundle and do a pull request to enable this feature.

To be able to create new objects, you need to provide a map between the RDFa types and the class names. (TODO: can we not index all mappings and do this automatically?)

Routing

Finally add the relevant routing to your configuration

Listing 15-5

```
1 create:
2     resource: "@SymfonyCmfCreateBundle/Resources/config/routing/rest.xml"
3 create_image:
4     resource: "@SymfonyCmfCreateBundle/Resources/config/routing/image.xml"
```

Alternative: Aloha Editor

Optional: Aloha Editor (create.js ships with the hallo editor, but if you prefer you can also use aloha)

> To use the Aloha editor, download the files here: *https://github.com/alohaeditor/Aloha-Editor/downloads/*[5] Unzip the contents of the "aloha" subfolder in the zip file as folder vendor/symfony-cmf/create-bundle/Symfony/Cmf/Bundle/CreateBundle/vendor/aloha Make sure you have just one aloha folder with the js, not aloha/aloha/... - you should have vendor/symfony-cmf/create-bundle/Symfony/Cmf/Bundle/CreateBundle/vendor/aloha/aloha.js

4. https://github.com/flack/createphp
5. https://github.com/alohaeditor/Aloha-Editor/downloads/

Usage

Adjust your template to load the editor js files if the current session is allowed to edit content.

Listing 15-6
```
{% render "symfony_cmf_create.jsloader.controller:includeJSFilesAction" %}
```

Plus make sure that assetic is rewriting paths in your css files, then include the base css files (and customize with your css as needed) with

Listing 15-7
```
{% include "SymfonyCmfCreateBundle::includecssfiles.html.twig" %}
```

The other thing you have to do is provide RDFa mappings for your model classes and adjust your templates to render with createphp so that create.js knows what content is editable.

Create XML metadata mappings in <Bundle>/Resources/rdf-mappings or a path you configured in rdf_config_dirs named after the full classname of your model classes with \\ replaced by a dot (.), i.e. Symfony.Cmf.Bundle.SimpleCmsBundle.Document.MultilangPage.xml. For an example mapping see the files in the cmf-sandbox. Reference documentation is in the *createphp library repository*[6].

To render your model, use the createphp twig tag:

Listing 15-8
```
{% createphp page as="rdf" %}
{{ rdf|raw }}
{% endcreatephp %}
```

Or if you need more control over the generated HTML:

Listing 15-9
```
{% createphp page as="rdf" %}
<div {{ createphp_attributes(rdf) }}>
    <h1 class="my-title" {{ createphp_attributes( rdf.title ) }}>{{ createphp_content(
rdf.title ) }}</h1>
    <div {{ createphp_attributes( rdf.body ) }}>{{ createphp_content( rdf.body ) }}</div>
</div>
{% endcreatephp %}
```

Developing the hallo wysiwyg editor

You can develop the hallo editor inside the Create bundle. By default, a minimized version of hallo that is bundled with create is used. To develop the actual code, you will need to checkout the full hallo repository first. You can do this by running the following command from the command line:

Listing 15-10
```
app/console cmf:create:init-hallo-devel
```

Then, set the `symfony_cmf_create > use_coffee` option to true in config.yml. This tells the jsloader to include the coffee script files from `Resources/public/vendor/hallo/src` with assetic, rather than the precompiled javascript from `Resources/public/vendor/create/deps/hallo-min.js`. This also means that you need to add a mapping for coffeescript in your assetic configuration and you need the *coffee compiler set up correctly*[7].

Listing 15-11
```
assetic:
    filters:
```

6. https://github.com/flack/createphp
7. http://coffeescript.org/#installation

```
        cssrewrite: ~
        coffee:
            bin: %coffee.bin%
            node: %coffee.node%
            apply_to: %coffee.extension%

symfony_cmf_create:
    # set this to true if you want to develop hallo and edit the coffee files
    use_coffee: true|false
```

In the cmf sandbox we did a little hack to not trigger coffee script compiling. In config.yml we make the coffee extension configurable. Now if the parameters.yml sets `coffee.extension` to `\.coffee` the coffeescript is compiled and the coffee compiler needs to be installed. If you set it to anything else like `\.nocoffee` then you do not need the coffee compiler installed.

The default values for the three parameters are

Listing 15-12
```
1 coffee.bin: /usr/local/bin/coffee
2 coffee.node: /usr/local/bin/node
3 coffee.extension: \.coffee
```

Chapter 16
DoctrinePHPCRBundle

The *DoctrinePHPCRBundle*[1] provides integration with the PHP content repository and optionally with Doctrine PHPCR-ODM to provide the ODM document manager in symfony.

This reference only explains the Symfony2 integration of PHPCR and PHPCR-ODM. To learn how to use PHPCR refer to *the PHPCR website*[2] and for Doctrine PHPCR-ODM to the *PHPCR-ODM documentation*[3].

This bundle is based on the AbstractDoctrineBundle and thus is similar to the configuration of the Doctrine ORM and MongoDB bundles.

Setup

See *Installing and configuring Doctrine PHPCR-ODM*

Configuration

If you want to only use plain PHPCR without the PHPCR-ODM, you can simply not configure the `odm` section to avoid loading the services at all. Note that most CMF bundles by default use PHPCR-ODM documents and thus need ODM enabled.

PHPCR Session Configuration

The session needs a PHPCR implementation specified in the `backend` section by the `type` field, along with configuration options to bootstrap the implementation. Currently we support `jackrabbit`,

1. `https://github.com/doctrine/DoctrinePHPCRBundle`
2. `http://phpcr.github.com/`
3. `http://docs.doctrine-project.org/projects/doctrine-phpcr-odm/en/latest/`

`doctrinedbal` and `midgard2`. Regardless of the backend, every PHPCR session needs a workspace, username and password.

Every PHPCR implementation should provide the workspace called *default*, but you can choose a different one. There is the `doctrine:phpcr:workspace:create` command to initialize a new workspace. See also *Services*.

The username and password you specify here are what is used on the PHPCR layer in the `PHPCR\SimpleCredentials`. They will usually be different from the username and password used by Midgard2 or Doctrine DBAL to connect to the underlying RDBMS where the data is actually stored.

If you are using one of the Jackalope backends, you can also specify `options`. They will be set on the Jackalope session. Currently this can be used to tune pre-fetching nodes by setting `jackalope.fetch_depth` to something bigger than 0.

Listing 16-1

```yaml
# app/config/config.yml
doctrine_phpcr:
    session:
        backend:
            # see below for how to configure the backend of your choice
        workspace: default
        username: admin
        password: admin
        ## tweak options for jackrabbit and doctrinedbal (all jackalope versions)
        # options:
        #    'jackalope.fetch_depth': 1
```

PHPCR Session with Jackalope Jackrabbit

The only setup required is to install Apache Jackrabbit (see *installing Jackrabbit*).

The configuration needs the `url` parameter to point to your jackrabbit. Additionally you can tune some other jackrabbit-specific options, for example to use it in a load-balanced setup or to fail early for the price of some round trips to the backend.

Listing 16-2

```yaml
# app/config/config.yml
doctrine_phpcr:
    session:
        backend:
            type: jackrabbit
            url: http://localhost:8080/server/
            ## jackrabbit only, optional. see https://github.com/jackalope/jackalope/blob/master/src/Jackalope/RepositoryFactoryJackrabbit.php
            # default_header: ...
            # expect: 'Expect: 100-continue'
            # enable if you want to have an exception right away if PHPCR login fails
            # check_login_on_server: false
            # enable if you experience segmentation faults while working with binary data in documents
            # disable_stream_wrapper: true
            # enable if you do not want to use transactions and you neither want the odm to automatically use transactions
            # its highly recommended NOT to disable transactions
            # disable_transactions: true
```

PHPCR Session with Jackalope Doctrine DBAL

This type uses Jackalope with a Doctrine database abstraction layer transport to provide PHPCR without any installation requirements beyond any of the RDBMS supported by Doctrine.

You need to configure a Doctrine connection according to the DBAL section in the *Symfony2 Doctrine documentation*[4].

Listing 16-3
```yaml
# app/config/config.yml
doctrine_phpcr:
    session:
        backend:
            type: doctrinedbal
            connection: doctrine.dbal.default_connection
            # enable if you want to have an exception right away if PHPCR login fails
            # check_login_on_server: false
            # enable if you experience segmentation faults while working with binary data
in documents
            # disable_stream_wrapper: true
            # enable if you do not want to use transactions and you neither want the odm
to automatically use transactions
            # its highly recommended NOT to disable transactions
            # disable_transactions: true
```

Once the connection is configured, you can create the database and you *need* to initialize the database with the `doctrine:phpcr:init:dbal` command.

Listing 16-4
```
app/console doctrine:database:create
app/console doctrine:phpcr:init:dbal
```

Of course, you can also use a different connection instead of the default. It is recommended to use a separate connection to a separate database if you also use Doctrine ORM or direct DBAL access to data, rather than mixing this data with the tables generated by jackalope-doctrine-dbal. If you have a separate connection, you need to pass the alternate connection name to the `doctrine:database:create` command with the `--connection` option. For doctrine PHPCR commands, this parameter is not needed as you configured the connection to use.

PHPCR Session with Midgard2

Midgard2 is an application that provides a compiled PHP extension. It implements the PHPCR API on top of a standard RDBMS.

To use the Midgard2 PHPCR provider, you must have both the [midgard2 PHP extension](*http://midgard-project.org/midgard2/#download*[5]) and [the midgard/phpcr package](*http://packagist.org/packages/midgard/phpcr*[6]) installed. The settings here correspond to Midgard2 repository parameters as explained in [the getting started document](*http://midgard-project.org/phpcr/#getting_started*[7]).

The session backend configuration looks as follows:

Listing 16-5
```yaml
# app/config/config.yml
doctrine_phpcr:
```

4. http://symfony.com/doc/current/book/doctrine.html
5. http://midgard-project.org/midgard2/#download
6. http://packagist.org/packages/midgard/phpcr
7. http://midgard-project.org/phpcr/#getting_started

```
    session:
        backend:
            type: midgard2
            db_type: MySQL
            db_name: midgard2_test
            db_host: "0.0.0.0"
            db_port: 3306
            db_username: ""
            db_password: ""
            db_init: true
            blobdir: /tmp/cmf-blobs
```

For more information, please refer to the *official Midgard PHPCR documentation*[8].

Doctrine PHPCR-ODM Configuration

This configuration section manages the Doctrine PHPCR-ODM system. If you do not configure anything here, the ODM services will not be loaded.

If you enable `auto_mapping`, you can place your mappings in `<Bundle>/Resources/config/doctrine/<Document>.phpcr.xml` resp. `...yml` to configure mappings for documents you provide in the `<Bundle>/Document` folder. Otherwise you need to manually configure the mappings section.

If `auto_generate_proxy_classes` is false, you need to run the `cache:warmup` command in order to have the proxy classes generated after you modified a document. You can also tune how and where to generate the proxy classes with the `proxy_dir` and `proxy_namespace` settings. The the defaults are usually fine here.

You can also enable *metadata caching*[9].

Listing 16-6
```
# app/config/config.yml
doctrine_phpcr:
    odm:
        configuration_id:     ~
        auto_mapping: true
        mappings:
            <name>:
                mapping:              true
                type:                 ~
                dir:                  ~
                alias:                ~
                prefix:               ~
                is_bundle:            ~
        auto_generate_proxy_classes: %kernel.debug%
        proxy_dir:            %kernel.cache_dir%/doctrine/PHPCRProxies
        proxy_namespace:      PHPCRProxies

        metadata_cache_driver:
            type:                 array
            host:                 ~
            port:                 ~
            instance_class:       ~
            class:                ~
            id:                   ~
```

8. http://midgard-project.org/phpcr/
9. http://symfony.com/doc/master/reference/configuration/doctrine.html

Translation configuration

If you are using multilingual documents, you need to configure the available languages. For more information on multilingual documents, see the *PHPCR-ODM documentation on Multilanguage*[10].

Listing 16-7
```yaml
# app/config/config.yml
doctrine_phpcr:
    odm:
        ...
        locales:
            en: [de, fr]
            de: [en, fr]
            fr: [en, de]
```

This block defines the order of alternative locales to look up if a document is not translated to the requested locale.

General Settings

If the *jackrabbit_jar* path is set, you can use the *doctrine:phpcr:jackrabbit* console command to start and stop jackrabbit.

You can tune the output of the *doctrine:phpcr:dump* command with *dump_max_line_length*.

Listing 16-8
```yaml
# app/config/config.yml
doctrine_phpcr:
    jackrabbit_jar:       /path/to/jackrabbit.jar
    dump_max_line_length:  120
```

Configuring Multiple Sessions

If you need more than one PHPCR backend, you can define `sessions` as child of the `session` information. Each session has a name and the configuration as you can use directly in `session`. You can also overwrite which session to use as `default_session`.

Listing 16-9
```yaml
# app/config/config.yml
doctrine_phpcr:
    session:
        default_session:      ~
        sessions:
            <name>:
                workspace:           ~ # Required
                username:            ~
                password:            ~
                backend:
                    # as above
                options:
                    # as above
```

If you are using the ODM, you will also want to configure multiple document managers.

Inside the odm section, you can add named entries in the `document_managers`. To use the non-default session, specify the session attribute.

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

Listing 16-10
```
odm:
    default_document_manager:  ~
    document_managers:
        <name>:
            # same keys as directly in odm, see above.
            session: <sessionname>
```

A full example looks as follows:

Listing 16-11
```
doctrine_phpcr:
    # configure the PHPCR sessions
    session:
        sessions:

            default:
                backend: %phpcr_backend%
                workspace: %phpcr_workspace%
                username: %phpcr_user%
                password: %phpcr_pass%

            website:
                backend:
                    type: jackrabbit
                    url: %magnolia_url%
                workspace: website
                username: %magnolia_user%
                password: %magnolia_pass%

            dms:
                backend:
                    type: jackrabbit
                    url: %magnolia_url%
                workspace: dms
                username: %magnolia_user%
                password: %magnolia_pass%
    # enable the ODM layer
    odm:
        document_managers:
            default:
                session: default
                mappings:
                    SandboxMainBundle: ~
                    SymfonyCmfContentBundle: ~
                    SymfonyCmfMenuBundle: ~
                    SymfonyCmfRoutingExtraBundle: ~

            website:
                session: website
                configuration_id: sandbox_magnolia.odm_configuration
                mappings:
                    SandboxMagnoliaBundle: ~

            dms:
                session: dms
                configuration_id: sandbox_magnolia.odm_configuration
                mappings:
                    SandboxMagnoliaBundle: ~

        auto_generate_proxy_classes: %kernel.debug%
```

This example also uses different configurations per repository (see the `repository_id` attribute). This case is explained in *Using a custom document class mapper with PHPCR-ODM*.

Services

You can access the PHPCR services like this:

Listing 16-12

```php
<?php

namespace Acme\DemoBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class DefaultController extends Controller
{
    public function indexAction()
    {
        // PHPCR session instance
        $session = $this->container->get('doctrine_phpcr.default_session');
        // PHPCR ODM document manager instance
        $documentManager =
$this->container->get('doctrine_phpcr.odm.default_document_manager');
    }
}
```

Events

You can tag services to listen to Doctrine PHPCR events. It works the same way as for Doctrine ORM. The only differences are

- use the tag name `doctrine_phpcr.event_listener` resp. `doctrine_phpcr.event_subscriber` instead of `doctrine.event_listener`.
- expect the argument to be of class DoctrineODMPHPCREventLifecycleEventArgs rather than in the ORM namespace.

You can register for the events as described in *the PHPCR-ODM documentation*[11].

```
services:
    my.listener:
        class: AcmeSearchBundleListenerSearchIndexer
            tags:
                - { name: doctrine_phpcr.event_listener, event: postPersist }
```

More information on the doctrine event system integration is in this *Symfony cookbook entry*[12].

11. http://docs.doctrine-project.org/projects/doctrine-phpcr-odm/en/latest/reference/events.html
12. http://symfony.com/doc/current/cookbook/doctrine/event_listeners_subscribers.html

Doctrine PHPCR Commands

All commands about PHPCR are prefixed with `doctrine:phpcr` and you can use the --session argument to use a non-default session if you configured several PHPCR sessions.

Some of these commands are specific to a backend or to the ODM. Those commands will only be available if such a backend is configured.

Use `app/console help <command>` to see all options each of the commands has.

- `doctrine:phpcr:workspace:create` Create a workspace in the configured repository
- `doctrine:phpcr:workspace:list` List all available workspaces in the configured repository
- `doctrine:phpcr:purge` Remove content from the repository
- `doctrine:phpcr:register-system-node-types` Register system node types in the PHPCR repository
- `doctrine:phpcr:register-node-types` Register node types in the PHPCR repository
- `doctrine:phpcr:fixtures:load` Load data fixtures to your PHPCR database.
- `doctrine:phpcr:import` Import xml data into the repository, either in JCR system view format or arbitrary xml
- `doctrine:phpcr:export` Export nodes from the repository, either to the JCR system view format or the document view format
- `doctrine:phpcr:dump` Dump the content repository
- `doctrine:phpcr:query` Execute a JCR SQL2 statement
- `doctrine:phpcr:mapping:info` Shows basic information about all mapped documents

To use the `doctrine:phpcr:fixtures:load` command, you additionally need to install the *DoctrineFixturesBundle*[13] and its dependencies. See that documentation page for how to use fixtures.

Jackrabbit specific commands

If you are using jackalope-jackrabbit, you also have a command to start and stop the jackrabbit server:

- `jackalope:run:jackrabbit` Start and stop the Jackrabbit server

Doctrine DBAL specific commands

If you are using jackalope-doctrine-dbal, you have a command to initialize the database:

- `jackalope:init:dbal` Prepare the database for Jackalope Doctrine DBAL

Note that you can also use the doctrine dbal command to create the database.

Some example command runs

Running *SQL2 queries*[14] against the repository

Listing 16-13
```
1 app/console doctrine:phpcr:query "SELECT title FROM [nt:unstructured] WHERE NAME() = 'home'"
```

Dumping nodes under /cms/simple including their properties

Listing 16-14

13. http://symfony.com/doc/current/bundles/DoctrineFixturesBundle/index.html
14. http://www.h2database.com/jcr/grammar.html

```
app/console doctrine:phpcr:dump /cms/simple --props=yes
```

Chapter 17
MenuBundle

The *MenuBundle*[1] provides menus from a doctrine object manager with the help of KnpMenuBundle.

Dependencies

This bundle is extending the *KnpMenuBundle*[2].

Unless you change defaults and provide your own implementations, this bundle also depends on

- `SymfonyRoutingExtraBundle` for the router service `symfony_cmf_routing_extra.dynamic_router`. Note that you need to explicitly enable the dynamic router as per default it is not loaded. See the *documentation of the routing extra bundle* for how to do this.
- *PHPCR-ODM* to load route documents from the content repository

Configuration

If you want to use default configurations, you do not need to change anything. The values are:

Listing 17-1
```yaml
symfony_cmf_menu:
    menu_basepath:        /cms/menu
    document_manager_name:  default
    admin_class:          ~
    document_class:       ~
    content_url_generator:  router
    content_key:          ~ # (resolves to DynamicRouter::CONTENT_KEY)
    route_name:           ~ # cmf routes are created by content instead of name
    content_basepath:     ~ # defaults to symfony_cmf_core.content_basepath
    use_sonata_admin:     auto # use true/false to force using / not using sonata admin
    multilang:                # the whole multilang section is optionnal
```

1. https://github.com/symfony-cmf/MenuBundle#readme
2. https://github.com/knplabs/KnpMenuBundle

```
12          use_sonata_admin:     auto # use true/false to force using / not using sonata admin
13          admin_class:          ~
14          document_class:       ~
15          locales:              [] # if you use multilang, you have to define at least one
    locale
```

If you want to render the menu from twig, make sure you have not disabled twig in the `knp_menu` configuration section.

If `sonata-project/doctrine-phpcr-admin-bundle` is added to the composer.json require section, the menu documents are exposed in the SonataDoctrinePhpcrAdminBundle. For instructions on how to configure this Bundle see *SonataDoctrinePhpcrAdminBundle*.

By default, `use_sonata_admin` is automatically set based on whether SonataDoctrinePhpcrAdminBundle is available but you can explicitly disable it to not have it even if sonata is enabled, or explicitly enable to get an error if Sonata becomes unavailable.

Menu entries

`MenuItem` document defines menu entries. You can build menu items based on symfony routes, absolute or relative urls or referenceable PHPCR-ODM content documents.

The menu tree is built from documents under [menu_basepath]/[menuname]. You can use different document classes for menu items, as long as they implement `Knp\Menu\NodeInterface` to integrate with KnpMenuBundle. The default `MenuNode` document discards children that do not implement this interface.

The currently highlighted entry is determined by checking if the content associated with a menu document is the same as the content the DynamicRouter has put into the request.

Usage

Adjust your twig template to load the menu.

Listing 17-2
```
1  {{ knp_menu_render('simple') }}
```

The menu name is the name of the node under `menu_basepath`. For example if your repository stores the menu nodes under `/cms/menu` , rendering "main" would mean to render the menu that is at `/cms/menu/main`

How to use non-default other components

If you use the cmf menu with PHPCR-ODM, you just need to store Route documents under `menu_basepath`. If you use a different object manager, you need to make sure that the menu root document is found with

Listing 17-3
```
1  $dm->find($menu_document_class, $menu_basepath . $menu_name)
```

The route document must implement `Knp\Menu\NodeInterface` - see `MenuNode` document for an example. You probably need to specify menu_document_class too, as only PHPCR-ODM can determine the document from the database content.

If you use the cmf menu with the DynamicRouter, you need no route name as the menu document just needs to provide a field content_key in the options. If you want to use a different service to generate URLs, you need to make sure your menu entries provide information in your selected content_key that the url generator can use to generate the url. Depending on your generator, you might need to specify a route_name too. Note that if you just want to generate normal symfony routes with a menu that is in the database, you can pass the core router service as content_url_generator, make sure the content_key never matches and make your menu documents provide the route name and eventual routeParameters.

Chapter 18
RoutingExtraBundle

The *RoutingExtraBundle*[1] integrates dynamic routing into Symfony using *Routing*.

The `ChainRouter` is meant to replace the default Symfony Router. All it does is collect a prioritized list of routers and try to match requests and generate URLs with all of them. One of the routers in that chain can of course be the default router so you can still use the standard way for some of your routes.

Additionally, this bundle delivers useful router implementations. Currently, there is the `DynamicRouter` that routes based on a custom loader logic for Symfony2 Route objects. The provider can be implemented using a database, for example with Doctrine *PHPCR-ODM*[2] or Doctrine ORM. This bundle provides a default implementation for Doctrine *PHPCR-ODM*[3].

The DynamicRouter service is only made available when explicitly enabled in the application configuration.

Finally this bundles provides route documents for Doctrine *PHPCR-ODM*[4] and a controller for redirection routes.

Dependencies

- *Symfony CMF routing*[5]

ChainRouter

The ChainRouter can replace the default symfony routing system with a chain- enabled implementation. It does not route anything on its own, but only loops through all chained routers. To handle standard configured symfony routes, the symfony default router can be put into the chain.

1. https://github.com/symfony-cmf/RoutingExtraBundle#readme
2. https://github.com/doctrine/phpcr-odm
3. https://github.com/doctrine/phpcr-odm
4. https://github.com/doctrine/phpcr-odm
5. https://github.com/symfony-cmf/Routing#readme

Configuration

In your app/config/config.yml, you can specify which router services you want to use. If you do not specify the routers_by_id map at all, by default the chain router will just load the built-in symfony router. When you specify the routers_by_id list, you need to have an entry for `router.default` if you want the Symfony2 router (that reads the routes from app/config/routing.yml).

The format is `service_name: priority` - the higher the priority number the earlier this router service is asked to match a route or to generate a url

Listing 18-1

```
# app/config/config.yml
symfony_cmf_routing_extra:
    chain:
        routers_by_id:
            # enable the DynamicRouter with high priority to allow overwriting configured
routes with content
            symfony_cmf_routing_extra.dynamic_router: 200
            # enable the symfony default router with a lower priority
            router.default: 100
        # whether the chain router should replace the default router. defaults to true
        # if you set this to false, the router is just available as service
        # symfony_cmf_routing_extra.router and you  need to do something to trigger it
        # replace_symfony_router: true
```

Loading routers with tagging

Your routers can automatically register, just add it as a service tagged with *router* 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 18-2

```
services:
    my_namespace.my_router:
        class: %my_namespace.my_router_class%
        tags:
            - { name: router, priority: 300 }
```

See also official Symfony2 *documentation for DependencyInjection tags*[6]

Dynamic Router

This implementation of a router uses the NestedMatcher which loads routes from a RouteProviderInterface. The provider interface can be easily implemented with Doctrine.

The router works with extended UrlMatcher and UrlGenerator classes that add loading routes from the database and the concept of referenced content.

The NestedMatcher service is set up with a route provider. See the configuration section for how to change the route_repository_service and the following section on more details for the default *PHPCR-ODM*[7] based implementation.

You may want to configure route enhancers to decide what controller is used to handle the request, to avoid hard coding controller names into your route documents.

6. http://symfony.com/doc/2.1/reference/dic_tags.html
7. https://github.com/doctrine/phpcr-odm

The minimum configuration required to load the dynamic router as service `symfony_cmf_routing_extra.dynamic_router` is to have `enabled: true` in your config.yml (the router is automatically enabled as soon as you add any other configuration to the *dynamic* entry). Without enabling it, the dynamic router service will not be loaded at all, allowing you to use the ChainRouter with your own routers

Listing 18-3
```yaml
# app/config/config.yml
symfony_cmf_routing_extra:
    dynamic:
        enabled: true
```

PHPCR-ODM integration

This bundle comes with a route repository implementation for *PHPCR-ODM*[8]. PHPCR is well suited to the tree nature of the data. If you use *PHPCR-ODM*[9] with a route document like the one provided, you can just leave the repository service at the default.

The default repository loads the route at the path in the request and all parent paths to allow for some of the path segments being parameters. If you need a different way to load routes or for example never use parameters, you can write your own repository implementation to optimize (see cmf_routing.xml for how to configure the service).

Match Process

Most of the match process is described in the documentation of the *CMF Routing component*[10]. The only difference is that the bundle will place the `contentDocument` in the request attributes instead of the route defaults.

Your controllers can (and should) declare the parameter $contentDocument in their `Action` methods if they are supposed to work with content referenced by the routes. See `Symfony\Cmf\Bundle\ContentBundle\Controller\ContentController` for an example.

Configuration

To configure what controller is used for which content, you can specify route enhancers. Presence of each of any enhancer configuration makes the DI container inject the respective enhancer into the DynamicRouter.

The possible enhancements are (in order of precedence):

- **(Explicit controller): If there is a _controller set in getRouteDefaults(),**

  no enhancer will overwrite it.

- **Explicit template: requires the route document to return a '_template'**

  parameter in getRouteDefaults. The configured generic controller is set by the enhancer.

- **Controller by alias: requires the route document to return a 'type' value in**

  getRouteDefaults()

8. https://github.com/doctrine/phpcr-odm
9. https://github.com/doctrine/phpcr-odm
10. https://github.com/symfony-cmf/Routing

- **Controller by class: requires the route document to return an object for**

  getRouteContent(). The content document is checked for being `instanceof` the class names in the map and if matched that controller is used. Instanceof is used instead of direct comparison to work with proxy classes and other extending classes.

- **Template by class: requires the route document to return an object for**

  getRouteContent(). The content document is checked for being `instanceof` the class names in the map and if matched that template will be set as '_template' in the $defaults and the generic controller used as controller.

Listing 18-4

```
# app/config/config.yml
symfony_cmf_routing_extra:
    dynamic:
        generic_controller: symfony_cmf_content.controller:indexAction
        controllers_by_type:
            editablestatic: sandbox_main.controller:indexAction
        controllers_by_class:
            Symfony\Cmf\Bundle\ContentBundle\Document\StaticContent:
symfony_cmf_content.controller::indexAction
        templates_by_class:
            Symfony\Cmf\Bundle\ContentBundle\Document\StaticContent:
SymfonyCmfContentBundle:StaticContent:index.html.twig

        # the route provider is responsible for loading routes.
        manager_registry: doctrine_phpcr
        manager_name: default

        # if you use the default doctrine route repository service, you
        # can use this to customize the root path for the `PHPCR-ODM`_
        # RouteProvider. This base path will be injected by the
        # Listener\IdPrefix - but only to routes matching the prefix,
        # to allow for more than one route source.
        routing_repositoryroot: /cms/routes

        # If you want to replace the default route provider or content repository
        # you can specify their service IDs here.
        route_provider_service_id: my_bundle.provider.endpoint
        content_repository_service_id: my_bundle.repository.endpoint

        # an orm provider might need different configuration. look at
        # cmf_routing.xml for an example if you need to define your own
        # service
```

To see some examples, please look at the *CMF sandbox*[11] and specifically the routing fixtures loading.

Using the PHPCR-ODM route document

All route classes must extend the Symfony core `Route` class. The documents can either be created by code (for example a fixtures script) or with a web interface like the one provided for Sonata PHPCR-ODM admin (see below).

PHPCR-ODM maps all features of the core route to the storage, so you can use setDefault, setRequirement, setOption and setHostnamePattern like normal. Additionally when creating a route, you can define whether .{_format} should be appended to the pattern and configure the required _format

11. `https://github.com/symfony-cmf/cmf-sandbox`

with a requirements. The other constructor option lets you control whether the route should append a trailing slash because this can not be expressed with a PHPCR name. The default is to have no trailing slash.

All routes are located under a configured root path, for example '/cms/routes'. A new route can be created in PHP code as follows:

Listing 18-5
```php
use Symfony\Cmf\Bundle\RoutingExtraBundle\Document\Route;
$route = new Route;
$route->setParent($dm->find(null, '/routes'));
$route->setName('projects');
// set explicit controller (both service and Bundle:Name:action syntax work)
$route->setDefault('_controller', 'sandbox_main.controller:specialAction');
```

The above example should probably be done as a route configured in a Symfony xml/yml file however, unless the end user is supposed to change the URL or the controller.

To link a content to this route, simply set it on the document.

Listing 18-6
```php
$content = new Content('my content'); // Content must be a mapped class
$route->setRouteContent($content);
```

This will put the document into the request parameters and if your controller specifies a parameter called `$contentDocument`, it will be passed this document.

You can also use variable patterns for the URL and define requirements and defaults.

Listing 18-7
```php
// do not forget leading slash if you want /projects/{id} and not /projects{id}
$route->setVariablePattern('/{id}');
$route->setRequirement('id', '\d+');
$route->setDefault('id', 1);
```

This will give you a route that matches the URL /projects/<number> but also /projects as there is a default for the id parameter. This will match `/projects/7` as well as `/projects` but not `/projects/x-4`. The document is still stored at `/routes/projects`. This will work because, as mentioned above, the route provider will look for route documents at all possible paths and pick the first that matches. In our example, if there is a route document at /routes/projects/7 that matches (no further parameters) it is selected. Otherwise we check if /routes/projects has a pattern that matches. If not, the top document at /routes is checked.

Of course you can also have several parameters, like with normal Symfony routes. The semantics and rules for patterns, defaults and requirements are exactly the same as in core routes.

Your controller can expect the $id parameter as well as the $contentDocument as we set a content on the route. The content could be used to define an intro section that is the same for each project or other shared data. If you don't need content, you can just not set it in the document.

Sonata Admin Configuration

If `sonata-project/doctrine-phpcr-admin-bundle` is added to the composer.json require section and the SonataDoctrinePhpcrAdminBundle is loaded in the application kernel, the route documents are exposed in the SonataDoctrinePhpcrAdminBundle. For instructions on how to configure this Bundle see *SonataDoctrinePhpcrAdminBundle*.

By default, `use_sonata_admin` is automatically set based on whether `SonataDoctrinePhpcrAdminBundle` is available, but you can explicitly disable it to not have it even if sonata is enabled, or explicitly enable to get an error if Sonata becomes unavailable.

If you want to use the admin, you want to configure the `content_basepath` to point to the root of your content documents.

Listing 18-8
```yaml
# app/config/config.yml
symfony_cmf_routing_extra:
    use_sonata_admin: auto # use true/false to force using / not using sonata admin
    content_basepath: ~ # used with sonata admin to manage content, defaults to
symfony_cmf_core.content_basepath
```

Form Type

The bundle defines a form type that can be used for classical "accept terms" checkboxes where you place urls in the label. Simply specify *symfony_cmf_routing_extra_terms_form_type* as the form type name and specify a label and an array with content_ids in the options

Listing 18-9
```php
$form->add('terms', 'symfony_cmf_routing_extra_terms_form_type', array(
    'label' => 'I have seen the <a href="%team%">Team</a> and <a href="%more%">More</a>
pages ...',
    'content_ids' => array('%team%' => '/cms/content/static/team', '%more%' => '/cms/
content/static/more')
));
```

The form type automatically generates the routes for the specified content and passes the routes to the trans twig helper for replacement in the label.

Further notes

See the documentation of the *CMF Routing component*[12] for information on the RouteObjectInterface, redirections and locales.

Notes:

- **RouteObjectInterface: The provided documents implement this interface to map content to routes and to (optional) provide**

  a custom route name instead of the symfony core compatible route name.

- Redirections: This bundle provides a controller to handle redirections.

Listing 18-10
```yaml
# app/config/config.yml
symfony_cmf_routing_extra:
    controllers_by_class:
        Symfony\Cmf\Component\Routing\RedirectRouteInterface:
symfony_cmf_routing_extra.redirect_controller:redirectAction
```

Customize

You can add more ControllerMapperInterface implementations if you have a case not handled by the provided ones.

12. https://github.com/symfony-cmf/Routing

If you use an ODM / ORM different to *PHPCR-ODM*[13], you probably need to specify the class for the route entity (in *PHPCR-ODM*[14], the class is automatically detected). For more specific needs, have a look at DynamicRouter and see if you want to extend it. You can also write your own routers to hook into the chain.

Learn more from the Cookbook

- *Using a custom route repository with Dynamic Router*

Further notes

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

- *Routing* for an introductory guide on Routing bundle
- *Routing* for most of the actual functionality implementation
- Symfony2's *Routing*[15] component page

13. https://github.com/doctrine/phpcr-odm
14. https://github.com/doctrine/phpcr-odm
15. http://symfony.com/doc/current/components/routing/introduction.html

Chapter 19
SearchBundle

The *SearchBundle*[1] provides integration with *LiipSearchBundle*[2] to provide a site wide search.

Dependencies

- *LiipSearchBundle*[3]

Configuration

The configuration key for this bundle is `symfony_cmf_search`

Listing 19-1
```yaml
# app/config/config.yml
symfony_cmf_search:
    document_manager_name:  default
    translation_strategy: child # can also be set to an empty string or attribute
    translation_strategy: attribute
    search_path: /cms/content
    search_fields:
        title: title
        summary: body
```

1. https://github.com/symfony-cmf/SearchBundle#readme
2. https://github.com/liip/LiipSearchBundle
3. https://github.com/liip/LiipSearchBundle#readme

Chapter 20
SimpleCmsBundle

The *SimpleCmsBundle*[1] provides a simplistic CMS on top of the CMF components and bundles.

While the core CMF components focus on flexibility, the simple CMS trades away some of that flexibility in favor of simplicity.

The SimpleCmsBundle provides a solution to easily map content, routes and menu items based on a single tree structure in the content repository.

For a simple example installation of the bundle check out the *Symfony CMF Standard Edition*[2]

You can find an introduction to the bundle in the *Getting started*[3] section.

The *CMF website*[4] is another application using the SimpleCmsBundle.

Dependencies

As specified in the bundle `composer.json` this bundle depends on most CMF bundles.

Configuration

The configuration key for this bundle is `symfony_cmf_simple_cms`

The `use_menu` option automatically enables a service to provide menus out of the simple cms if the MenuBundle is enabled. You can also explicitly disable it if you have the menu bundle but do not want to use the default service, or explicitly enable to get an error if the menu bundle becomes unavailable.

The routing section is configuring what template or controller to use for a content class. This is reusing what routing extra does, please see the corresponding *routing configuration section*. It also explains the `generic_controller`.

See the section below for multilanguage support.

1. https://github.com/symfony-cmf/SimpleCmsBundle#readme
2. https://github.com/symfony-cmf/symfony-cmf-standard
3. #cmf-getting-started-simplecms
4. https://github.com/symfony-cmf/symfony-cmf-website/

Listing 20-1
```yaml
# app/config/config.yml
symfony_cmf_simple_cms:
    use_menu:             auto # use true/false to force providing / not providing a menu
    use_sonata_admin:     auto # use true/false to force using / not using sonata admin
    document_class:       Symfony\Cmf\Bundle\SimpleCmsBundle\Document\Page
    # controller to use to render documents with just custom template
    generic_controller:   symfony_cmf_content.controller:indexAction
    # where in the PHPCR tree to store the pages
    basepath:             /cms/simple
    routing:
        content_repository_id:  symfony_cmf_routing_extra.content_repository
        controllers_by_class:
            # ...
        templates_by_class:
            # ...
    multilang:
        locales:              []
```

If you have the Sonata PHPCR-ODM admin bundle enabled but do *NOT* want to show the default admin provided by this bundle, you can add the following to your configuration

Listing 20-2
```yaml
symfony_cmf_simple_cms:
    use_sonata_admin: false
```

Multi-language support

The multi-language-mode is enabled by providing the list of allowed locales in the `multilang > locales` field.

In multi-language-mode the Bundle will automatically use the `Symfony\Cmf\Bundle\SimpleCmsBundle\Document\MultilangPage` as the `document_class` unless a different class is configured explicitly.

This class will by default prefix all routes with `/{_locale}`. This behavior can be disabled by setting the second parameter in the constructor of the model to false.

Furthermore the routing layer will be configured to use `Symfony\Cmf\Bundle\SimpleCmsBundle\Document\MultilangRouteRepository` which will ensure that even with the locale prefix the right content node will be found. Furthermore it will automatically add a `_locale` requirement listing the current available locales for the matched route.

Since SimpleCmsBundle only provides a single tree structure, all nodes will have the same node name for all languages. So a url `http://foo.com/en/bar` for english content will look like `http://foo.com/de/bar` for german content. At times it might be most feasible to use integers as the node names and simple append the title of the node in the given locale as an anchor. So for example `http://foo.com/de/1#my title` and `http://foo.com/de/1#mein title`. If you need language specific URLs, you want to use the CMF routing bundle and content bundle directly to have a separate route document per language.

Rendering

You can specify the template to render a SimpleCms page, or use a controller where you then give the page document to the template. A simple example for such a template is

Listing 20-3
```
{% block content %}

    <h1>{{ page.title }}</h1>

    <div>{{ page.body|raw }}</div>

    <ul>
    {% foreach tag in page.tags %}
        <li>{{ tag }}</li>
    {% endforeach %}
    </ul>

{% endblock %}
```

If you have the CreateBundle enabled, you can also output the document with RDFa annotations, allowing you to edit the content as well as the tags in the frontend. The most simple form is the following twig block:

Listing 20-4
```
{% block content %}

    {% createphp page as="rdf" %}
        {{ rdf|raw }}
    {% endcreatephp %}

{% endblock %}
```

If you want to control more detailed what should be shown with RDFa, see chapter *CreateBundle*.

Extending the Page class

The default Page document `Symfony\Cmf\Bundle\SimpleCmsBundle\Document\Page` is relatively simple, shipping with a handful of the most common properties for building a typical page: title, body, tags, publish dates etc.

If this is not enough for your project you can easily provide your own document by extending the default Page document and explicitly setting the configuration parameter to your own document class:

Listing 20-5
```
# app/config/config.yml
symfony_cmf_simple_cms:
    ...
    document_class:       Acme\DemoBundle\Document\MySuperPage
    ...
```

Alternatively, the default Page document contains an `extras` property. This is a key - value store (where value must be string or null) which can be used for small trivial additions, without having to extend the default Page document.

For example:

Listing 20-6

```php
$page = new Page();

$page->setTitle('Hello World!');
$page->setBody('Really interesting stuff...');

// set extras
$extras = array(
    'subtext' => 'Add CMS functionality to applications built with the Symfony2 PHP
framework.',
    'headline-icon' => 'exclamation.png',
);

$page->setExtras($extras);

$documentManager->persist($page);
```

These properties can then be accessed in your controller or templates via the `getExtras()` or `getExtra($key)` methods.

Chapter 21
SonataDoctrinePhpcrAdminBundle

The *SonataDoctrinePhpcrAdminBundle*[1] provides integration with the SonataAdminBundle to enable easy creation of admin UIs.

Dependencies

- *SonataAdminBundle*[2]
- *TreeBundle*[3]

Configuration

Listing 21-1

```
sonata_doctrine_phpcr_admin:
    templates:
        form:

            # Default:
            - SonataDoctrinePHPCRAdminBundle:Form:form_admin_fields.html.twig
        filter:

            # Default:
            - SonataDoctrinePHPCRAdminBundle:Form:filter_admin_fields.html.twig
        types:
            list:

                # Prototype
                name:                 []
    document_tree:
        # Prototype
```

1. https://github.com/sonata-project/SonataDoctrinePhpcrAdminBundle#readme
2. https://github.com/sonata-project/SonataAdminBundle
3. https://github.com/symfony-cmf/TreeBundle#readme

```
18          class: # name of the class
19              valid_children:       []
20              image:
```

Chapter 22
TreeBrowserBundle

The *TreeBrowserBundle*[1] provides a tree navigation on top of a PHPCR repository.

This bundle consists of two parts:

- Generic Tree Browser with a TreeInterface
- PHPCR tree implementation and GUI for a PHPCR browser

Dependencies

- *FOSJsRoutingBundle*[2]
- Install jQuery. *SonatajQueryBundle*[3] strongly suggested.

Configuration

The configuration key for this bundle is `symfony_cmf_tree_browser`

Listing 22-1
```yaml
# app/config/config.yml
symfony_cmf_tree_browser:
    session:  default
```

Routing

The bundle will create routes for each tree implementation found. In order to make those routes available you need to include the following in your routing configuration:

Listing 22-2

1. https://github.com/symfony-cmf/TreeBrowserBundle#readme
2. https://github.com/FriendsOfSymfony/FOSJsRoutingBundle
3. https://github.com/sonata-project/SonatajQueryBundle

```yaml
# app/config/routing.yml
symfony_cmf_tree:
    resource: .
    type: 'symfony_cmf_tree'
```

Usage

You have select.js and init.js which are a wrapper to build a jquery tree. Use them with SelectTree.initTree resp. AdminTree.initTree

- SelectTree in select.js is a tree to select a node to put its id into a field
- AdminTree in init.js is a tree to create, move and edit nodes

Both have the following options when creating:

- config.selector: jquery selector where to hook in the js tree
- config.rootNode: id to the root node of your tree, defaults to "/"
- config.selected: id of the selected node
- config.ajax.children_url: Url to the controller that provides the children of a node
- config.routing_defaults: array for route parameters (such as _locale etc.)
- config.path.expanded: tree path where the tree should be expanded to at the moment
- config.path.preloaded: tree path what node should be preloaded for faster user experience

select.js only

- config.output: where to write the id of the selected node

init.js only

- config.labels: array containing the translations for the labels of the context menu (keys 'createItem' and 'deleteItem')
- config.ajax.move_url: Url to the controller for moving a child (i.e. giving it a new parent node)
- config.ajax.reorder_url: Url to the controller for reordering siblings
- config.types: array indexed with the node types containing information about valid_children, icons and available routes, used for the creation of context menus and checking during move operations.

Examples

Look at the templates in the Sonata Admin Bundle for examples how to build the tree:

- *init.js*[4]
- *select.js*[5] (look for doctrine_phpcr_type_tree_model_widget)

4. https://github.com/sonata-project/SonataDoctrinePhpcrAdminBundle/blob/master/Resources/views/Tree/tree.html.twig
5. https://github.com/sonata-project/SonataDoctrinePhpcrAdminBundle/blob/master/Resources/views/Form/form_admin_fields.html.twig

In the same bundle the *PhpcrOdmTree*[6] implements the tree interface and gives an example how to implement the methods.

Here are some common tips about TreeBrowser utilization :

Define tree elements

The first step, is to define all the elements allowed in the tree and their children. Have a look at the *cmf-sandbox configuration*[7], the section document_tree in sonata_doctrine_phpcr_admin.

This configuration is set for all your application trees regardless their type (admin or select).

Listing 22-3
```
sonata_doctrine_phpcr_admin:
    document_tree_defaults: [locale]
    document_tree:
        Doctrine\PHPCR\Odm\Document\Generic:
            valid_children:
                - all
        Symfony\Cmf\Bundle\ContentBundle\Document\MultilangStaticContent:
            valid_children:
                - Symfony\Cmf\Bundle\BlockBundle\Document\SimpleBlock
                - Symfony\Cmf\Bundle\BlockBundle\Document\ContainerBlock
                - Symfony\Cmf\Bundle\BlockBundle\Document\ReferenceBlock
                - Symfony\Cmf\Bundle\BlockBundle\Document\ActionBlock
        Symfony\Cmf\Bundle\BlockBundle\Document\ReferenceBlock:
            valid_children: []
        ...
```

How to add an admin tree to your page

This can be done either in an action template or in a custom block.

You have to specify the tree root and the selected item, this allows you to have different type of content in your tree.

In this example, we will have the menu elements :

Listing 22-4
```
{% render 'sonata.admin.doctrine_phpcr.tree_controller:treeAction' with { 'root':
websiteId~"/menu", 'selected': menuNodeId } %}
```

How to customize the tree behaviour

The TreeBrowserBundle is based on *jsTree*[8]. jsTree works with events, dispatched everytime the user does an action.

A simple way to customize the tree behavior is to bind your actions to those events.

If you have a look at init.js and select.js, you will notice that actions are already bound to some of the tree events. If the default behavior is not what you need, JQuery provide the unbind function to solve the problem.

Here is a simple way to remove the context menu from the admin tree :

Listing 22-5
```
{% render 'sonata.admin.doctrine_phpcr.tree_controller:treeAction' with { 'root':
websiteId~"/menu", 'selected': menuNodeId } %}
```

6. https://github.com/sonata-project/SonataDoctrinePhpcrAdminBundle/blob/master/Tree/PhpcrOdmTree.php
7. https://github.com/symfony-cmf/cmf-sandbox/blob/master/app/config/config.yml
8. http://www.jstree.com/documentation

```
<script type="text/javascript">
    $(document).ready(function() {
        $('#tree').bind("before.jstree", function (e, data) {
            if (data.plugin === "contextmenu") {
                e.stopImmediatePropagation();
                return false;
            }
        });
    });
</script>
```

By default, the item selection open the edit route of the admin class of the element. This action is bind to the "select_node.jstree".

If you want to remove it, you just need to call the unbind function on this event :

Listing 22-6
```
<script type="text/javascript">
    $(document).ready(function() {
        $('#tree').unbind('select_node.jstree');
    });
</script>
```

Then you can bind it on another action.

For example, if your want to open a custom action :

Listing 22-7
```
$('#tree').bind("select_node.jstree", function (event, data) {
    if ((data.rslt.obj.attr("rel") == 'Symfony_Cmf_Bundle_MenuBundle_Document_MenuNode'
        || data.rslt.obj.attr("rel") ==
'Symfony_Cmf_Bundle_MenuBundle_Document_MultilangMenuNode')
        && data.rslt.obj.attr("id") != '{{ menuNodeId }}'
    ) {
        var routing_defaults = {'locale': '{{ locale }}', '_locale': '{{ _locale }}'};
        routing_defaults["id"] = data.rslt.obj.attr("url_safe_id");
        window.location = Routing.generate('presta_cms_page_edit', routing_defaults);
    }
});
```

Don't forget to add your custom route to the fos_js_routing.routes_to_expose configuration :

Listing 22-8
```
fos_js_routing:
    routes_to_expose:
        - symfony_cmf_tree_browser.phpcr_children
        - symfony_cmf_tree_browser.phpcr_move
        - sonata.admin.doctrine_phpcr.phpcrodm_children
        - sonata.admin.doctrine_phpcr.phpcrodm_move
        - presta_cms_page_edit
```

Chapter 23

Using a custom document class mapper with PHPCR-ODM

The default document class mapper of PHPCR-ODM uses the attribute `phpcr:class` to store and retrieve the document class of a node. When accessing an existing PHPCR repository, you might need different logic to decide on the class.

You can extend the `DocumentClassMapper` or implement `DocumentClassMapperInterface` from scratch. The important methods are getClassName that needs to find the class name and writeMetadata that needs to make sure the class of a newly stored document can be determined when loading it again.

Then you can overwrite the doctrine.odm_configuration service to call setDocumentClassMapper on it. An example from the *symfony cmf sandbox*[1] (magnolia_integration branch):

Listing 23-1

```
# Resources/config/services.yml

# if you want to overwrite default configuration, otherwise use a
# custom name and specify in odm configuration block

doctrine.odm_configuration:
    class: %doctrine_phpcr.odm.configuration.class%
    calls:
        - [ setDocumentClassMapper, [@sandbox_magnolia.odm_mapper] ]

sandbox_magnolia.odm_mapper:
    class: "Sandbox\MagnoliaBundle\Document\MagnoliaDocumentClassMapper"
    arguments:
        - 'standard-templating-kit:pages/stkSection': 'Sandbox\MagnoliaBundle\Document\Section'
```

Here we create a mapper that uses a configuration to read node information and map that onto a document class.

If you have several repositories, you can use one configuration per repository. See *Configuring Multiple Sessions*.

1. `https://github.com/symfony-cmf/cmf-sandbox/tree/magnolia_integration`

Chapter 24

Using a custom route repository with Dynamic Router

The Dynamic Router allows you to customize the route Repository (i.e. the class responsible for retrieving routes from the database), and by extension, the Route objects.

Creating the route repository

The route repository must implement the *RouteRepositoryInterface* The following class provides a simple solution using an ODM Repository.

Listing 24-1

```php
<?php

namespace MyVendor\Bundle\MyBundle\Repository;
use Doctrine\ODM\PHPCR\DocumentRepository;
use Symfony\Cmf\Component\Routing\RouteRepositoryInterface;
use Symfony\Component\Routing\RouteCollection;
use Symfony\Component\Routing\Route as SymfonyRoute;

class RouteRepository extends DocumentRepository implements RouteRepositoryInterface

{
    // this method is used to find routes matching the given URL
    public function findManyByUrl($url)
    {
        // for simplicity we retrieve one route
        $myDocument = $this->findOneBy(array(
            'url' => $url,
        ));

        $pattern = $myDocument->getUrl(); // e.g. "/this/is/a/url"

        $collection = new RouteCollection();

```

```
        // create a new Route and set our document as
        // a default (so that we can retrieve it from the request)
        $route = new SymfonyRoute($ep->getPath(), array(
            'document' => $document,
        ));

        // add the route to the RouteCollection using
        // a unique ID as the key.
        $collection->add('my_route_'.uniqid(), $route);

        return $collection;
    }

    // this method is used to generate URLs, e.g. {{ path('foobar') }}
    public function getRouteByName($name, $params = array())
    {
        $document = $this->findOneBy(array(
            'name' => $name,
        ));

        if ($route) {
            $route = new SymfonyRoute($route->getPattern(), array(
                'document' => $document,
            ));
        }

        return $route;
    }
}
```

As you may have noticed we return a *RouteCollection* object - why not return a single *Route*? The Dynamic Router allows us to return many *candidate* routes, in other words, routes that *might* match the incoming URL. This is important to enable the possibility of matching *dynamic* routes, */page/{page_id}/edit* for example. In our example we match the given URL exactly and only ever return a single *Route*.

Replacing the default CMF repository

To replace the default *RouteRepository* it is necessary to modify your configuration as follows:

Listing 24-2
```
# app/config/config.yml
symfony_cmf_routing_extra:
    dynamic:
        enabled: true
        route_repository_service_id: my_bundle.repository.endpoint
```

Where *my_bundle.repository.endpoint* is the service ID of your repository. See *Creating and configuring services in the container*[1] for information on creating custom services.

1. http://symfony.com/doc/current/book/service_container.html#creating-configuring-services-in-the-container/

Chapter 25
Installing the CMF sandbox

This tutorial shows how to install the Symfony CMF Sandbox, a demo platform aimed at showing the tool's basic features running on a demo environment. This can be used to evaluate the platform or to see actual code in action, helping you understand the tool's internals.

While it can be used as such, this sandbox does not intend to be a development platform. If you are looking for installation instructions for a development setup, please refer to:

- *Installing the Symfony CMF Standard Edition* page for instructions on how to quickly install the CMF (recommended for development)
- *Installing and configuring the CMF core* for step-by-step installation and configuration details (if you want to know all the details)

Preconditions

As Symfony CMF Sandbox is based on Symfony2, you should make sure you meet the *Requirements for running Symfony2*[1]. *Git 1.6+*[2], *Curl*[3] and PHP Intl are also needed to follow the installation steps listed below.

If you wish to use Jackalope + Apache JackRabbit as the storage medium (recommended), you will also need Java (JRE). For other mechanisms and its requirements, please refer to their respective sections.

Installation

Apache Jackrabbit

The Symfony CMF Sandbox uses Jackalope with Apache JackRabbit by default. Alternative storage methods can be configured, but this is the most tested, and should be the easiest to setup.

1. http://symfony.com/doc/current/reference/requirements.html
2. http://git-scm.com/
3. http://curl.haxx.se/

You can get the latest Apache Jackrabbit version from the project's *official download page*[4]. To start it, use the following command

Listing 25-1
```
java -jar jackrabbit-standalone-*.jar
```

By default the server is listening on the 8080 port, you can change this by specifying the port on the command line.

Listing 25-2
```
java -jar jackrabbit-standalone-*.jar --port 8888
```

For unix systems, you can get the start-stop script for /etc/init.d *here*[5]

Getting the sandbox code

The Symfony CMF Sandbox source code is available on github. To get it use

Listing 25-3
```
git clone git://github.com/symfony-cmf/cmf-sandbox.git
```

Move into the folder and copy the default configuration files

Listing 25-4
```
cd cmf-sandbox
cp app/config/parameters.yml.dist app/config/parameters.yml
cp app/config/phpcr_jackrabbit.yml.dist app/config/phpcr.yml
```

These two files include the default configuration parameters for the sandbox storage mechanism. You can modify them to better fit your needs

The second configuration file refers to specific jackalope + jackrabbit configuration. There are other files available for different stack setups.

Next, get composer and install and the necessary bundles (this may take a while)

Listing 25-5
```
curl -s http://getcomposer.org/installer | php --
php composer.phar install
```

On Windows you need to run the shell as Administrator or edit the composer.json and change the line "symfony-assets-install": "symlink" to "symfony-assets-install": "" If you fail to do this you might receive:

Listing 25-6
```
[Symfony\Component\Filesystem\Exception\IOException]
Unable to create symlink due to error code 1314: 'A required privilege is not held by the
client'. Do you have the required Administrator-rights?
```

4. http://jackrabbit.apache.org/downloads.html
5. https://github.com/sixty-nine/Jackrabbit-startup-script

Preparing the PHPCR repository

Now that you have all the code, you need to setup your PHPCR repository. PHPCR organizes data in workspaces, and sandbox uses the "default" workspace, which is exists by default in Jackrabbit. If you use other applications that require Jackrabbit, or if you just wish to change the workspace name, you can do so in app/config/phpcr.yml. The following command will create a new workspace named "sandbox" in Jackrabbit. If you decide to use the "default" workspace, you can skip it.

Listing 25-7
```
app/console doctrine:phpcr:workspace:create sandbox
```

Once your workspace is set up, you need to *register the node types*[6] for phpcr-odm:

Listing 25-8
```
app/console doctrine:phpcr:register-system-node-types
```

Import the fixtures

The admin backend is still in an early stage. Until it improves, the easiest is to programmatically create data. The best way to do that is with the doctrine data fixtures. The DoctrinePHPCRBundle included in the symfony-cmf repository provides a command to load fixtures.

Listing 25-9
```
app/console -v doctrine:phpcr:fixtures:load
```

Run this to load the fixtures from the Sandbox MainBundle, which will populate your repository with dummy data, i.e. loads the demo pages.

Accessing your sandbox

The sandbox should now be accessible on your web server.

Listing 25-10
```
http://localhost/app_dev.php
```

In order to run the sandbox in production mode you need to generate the doctrine proxies and dump the assetic assets:

Listing 25-11
```
app/console cache:warmup --env=prod --no-debug
app/console assetic:dump --env=prod --no-debug
```

Alternative storage mechanisms

Symfony CMF and the sandbox are storage agnostic, which means you can change the storage mechanism without having to change your code. The default storage mechanism for the sandbox is Jackalope + Apache Jackrabbit, as it's the most tested and stable setup. However, other alternatives are available.

Jackalope + Doctrine DBAL

6. https://github.com/doctrine/phpcr-odm/wiki/Custom-node-type-phpcr%3Amanaged

By default, when using Doctrine DBAL, data is stored using a *Sqlite*[7] database. Refer to the project's page for installation instructions. If you wish to use other database systems, change the configuration parameters in app/config/parameters.yml. Refer to *Symfony's page on Doctrine DBAL configuration*[8] or *Doctrine's documentation*[9] for more information.

Move into the sandbox folder and copy the default configuration file for Doctrine DBAL setup:

Listing 25-12
```
cd cmf-sandbox
cp app/config/phpcr_doctrine_dbal.yml.dist app/config/phpcr.yml
```

Next, you need to install the actual Doctrine DBAL bundle required by jackalope:

Listing 25-13
```
php composer.phar require jackalope/jackalope-doctrine-dbal:dev-master
```

And create and init your database:

Listing 25-14
```
app/console doctrine:database:create
app/console doctrine:phpcr:init:dbal
```

After this, your should follow the steps in Preparing the PHPCR repository.

Doctrine caching

Optionally, to improve performance and enable the meta data, you can install LiipDoctrineCacheBundle by typing the following command:

Listing 25-15
```
php composer.phar require liip/doctrine-cache-bundle:dev-master
```

And adding the following entry to your app/AppKernel.php:

Listing 25-16
```
// app/AppKernel.php
public function registerBundles()
{
  $bundles = array(
      // ...
      new Liip\DoctrineCacheBundle\LiipDoctrineCacheBundle(),
      // ...
  );
}
```

Finally uncomment the caches settings in the phpcr.yml as well as the liip_doctrine_cache settings in config.yml.

Listing 25-17
```
# app/config/phpcr.yml
caches:
    meta: liip_doctrine_cache.ns.meta
    nodes: liip_doctrine_cache.ns.nodes
```

Listing 25-18

7. http://www.sqlite.org/
8. http://symfony.com/doc/current/reference/configuration/doctrine.html#doctrine-dbal-configuration
9. http://docs.doctrine-project.org/projects/doctrine-dbal/en/latest/reference/configuration.html

```
# app/config/config.yml

# jackalope doctrine caching
liip_doctrine_cache:
    namespaces:
        meta:
            type: file_system
        nodes:
            type: file_system
```

Midgard2 PHPCR provider

If you want to run the CMF sandbox with the *Midgard2 PHPCR*[10] provider instead of Jackrabbit, you need to install the midgard2 PHP extension. On current Debian / Ubuntu systems, this is simply done with

Listing 25-19
```
sudo apt-get install php5-midgard2
```

On OS X you can install it using either *Homebrew*[11] with

Listing 25-20
```
brew install midgard2-php
```

or *MacPorts*[12] with

Listing 25-21
```
sudo port install php5-midgard2
```

You also need to download *midgard_tree_node.xml*[13] and *midgard_namespace_registry.xml*[14] schema files, and place them into "<your-midgard2-folder>/schema" (defaults to "/usr/share/midgard2/schema")

To have the Midgard2 PHPCR implementation installed run the following additional command:

Listing 25-22
```
php composer.phar require midgard/phpcr:dev-master
```

Finally, switch to one of the Midgard2 configuration file:

Listing 25-23
```
cp app/config/phpcr_midgard_mysql.yml.dist app/config/phpcr.yml
```

or

Listing 25-24
```
cp app/config/phpcr_midgard_sqlite.yml.dist app/config/phpcr.yml
```

After this, your should follow the steps in Preparing the PHPCR repository to continue the installation process.

10. http://midgard-project.org/phpcr/
11. http://mxcl.github.com/homebrew/
12. http://www.macports.org/
13. https://raw.github.com/midgardproject/phpcr-midgard2/master/data/share/schema/midgard_tree_node.xml
14. https://github.com/midgardproject/phpcr-midgard2/raw/master/data/share/schema/midgard_namespace_registry.xml

Chapter 26
Routing

The *Symfony CMF Routing component*[1] library extends the Symfony2 core routing component. Even though it has Symfony in its name, it does not need the full Symfony2 framework and can be used in standalone projects. For integration with Symfony we provide *RoutingExtraBundle*.

At the core of the Symfony CMF Routing component is the `ChainRouter`, that is used instead of the Symfony2's default routing system. The ChainRouter can chain several `RouterInterface` implementations, one after the other, to determine what should handle each request. The default Symfony2 router can be added to this chain, so the standard routing mechanism can still be used.

Additionally, this component is meant to provide useful implementations of the routing interfaces. Currently, it provides the `DynamicRouter`, which uses a `RequestMatcherInterface` to dynamically load Routes, and can apply `RouteEnhancerInterface` strategies in order to manipulate them. The provided `NestedMatcher` can dynamically retrieve Symfony2 *Route*[2] objects from a `RouteProviderInterface`. This interfaces abstracts a collection of Routes, that can be stored in a database, like Doctrine PHPCR-ODM or Doctrine ORM. The `DynamicRouter` also uses a `UrlGenerator` instance to generate Routes and an implementation is provided under `ProviderBasedGenerator` that can generate routes loaded from a `RouteProviderInterface` instance, and the `ContentAwareGenerator` on top of it to determine the route object from a content object.

To use this component outside of the Symfony2 framework context, have a look at the core Symfony2 *Routing*[3] to get a fundamental understanding of the component. CMF Routing just extends the basic behaviour.

Dependencies

This component uses *composer*[4]. It needs the Symfony2 Routing component and the Symfony2 HttpKernel (for the logger interface and cache warm-up interface).

1. `https://github.com/symfony-cmf/Routing`
2. `http://api.symfony.com/master/Symfony/Component/Routing/Route.html`
3. `https://github.com/symfony/Routing`
4. `http://getcomposer.org`

For the `DynamicRouter` you will need something to implement the `RouteProviderInterface` with. We suggest using Doctrine as this provides an easy way to map classes into a database.

ChainRouter

At the core of Symfony CMF's Routing component sits the `ChainRouter`. It's used as a replacement for Symfony2's default routing system, and is responsible for determining the parameters for each request. Typically you need to determine which Controller will handle this request - in the full stack Symfony2 framework, this is identified by the `_controller` field of the parameters.

The `ChainRouter` works by accepting a set of prioritized routing strategies, *RouterInterface*[5] implementations, commonly referred to as "Routers".

When handling an incoming request, the ChainRouter iterates over the configured Routers, by their configured priority, until one of them is able to *match*[6] the request and provide the request parameters.

Routers

The `ChainRouter` is incapable of, by itself, making any actual routing decisions. It's sole responsibility is managing the given set of Routers, which are the true responsible for matching a request and determining its parameters.

You can easily create your own Routers by implementing *RouterInterface*[7] but Symfony CMF already includes a powerful route matching system that you can extend to your needs.

If you are using this as part of a full Symfony CMF project, please refer to *RoutingExtraBundle* for instructions on how to add Routers to the `ChainRouter`. Otherwise, use the `ChainRouter`'s `add` method to configure new Routers.

Symfony2 Default Router

The Symfony2 routing mechanism is itself a `RouterInterface` implementation, which means you can use it as a Router in the `ChainRouter`. This allows you to use the default routing declaration system.

Dynamic Router

The Symfony2 default Router was developed to handle static Route definitions, as they are traditionally declared in configuration files, prior to execution. This makes it a poor choice to handle dynamically defined routes, and to handle those situations, this bundle comes with the `DynamicRouter`. It is capable of handling Routes from more dynamic data sources, like database storage, and modify the resulting parameters using a set of enhancers that can be easily configured, greatly extending Symfony2's default functionality.

Matcher

The `DynamicRouter` uses a `RequestMatcherInterface` or `UrlMatcherInterface` instance to match the received Request or URL, respectively, to a parameters array. The actual matching logic depends on the underlying implementation you choose. You can easily use you own matching strategy by pass it to the

5. `http://api.symfony.com/2.1/Symfony/Component/Routing/RouterInterface.html`
6. `http://api.symfony.com/2.1/Symfony/Component/Routing/RouterInterface.html#method_match`
7. `http://api.symfony.com/master/Symfony/Component/Routing/RouterInterface.html`

`DynamicRouter` constructor. As part of this bundle, a `NestedMatcher` is already provided which you can use straight away, or as reference for your own implementation.

Its other feature are the `RouteEnhancerInterface` strategies used to infer routing parameters from the information provided by the match (see below).

NestedMatcher

The provided `RequestMatcherInterface` implementation is `NestedMatcher`. It is suitable for use with `DynamicRouter`, and it uses a multiple step matching process to determine the resulting routing parameters from a given *Request*[8].

It uses a `RouteProviderInterface` implementation, which is capable of loading candidate *Route*[9] objects for a Request dynamically from a data source. Although it can be used in other ways, the `RouteProviderInterface`'s main goal is to be easily implemented on top of Doctrine PHPCR ODM or a relational database, effectively allowing you to store and manage routes dynamically from database.

The `NestedMatcher` uses a 3-step matching process to determine which Route to use when handling the current Request:

- Ask the `RouteProviderInterface` for the collection of `Route` instances potentially matching the `Request`
- Apply all `RouteFilterInterface` to filter down this collection
- Let the `FinalMatcherInterface` instance decide on the best match among the remaining `Route` instances and transform it into the parameter array.

RouteProviderInterface

Based on the `Request`, the `NestedMatcher` will retrieve an ordered collection of `Route` objects from the `RouteProviderInterface`. The idea of this provider is to provide all routes that could potentially match, but **not** to do any elaborate matching operations yet - this is the job of the later steps.

The underlying implementation of the `RouteProviderInterface` is not in the scope of this bundle. Please refer to the interface declaration for more information. For a functional example, see *RoutingExtraBundle*[10].

RouteFilterInterface

The `NestedMatcher` can apply user provided `RouteFilterInterface` implementations to reduce the provided `Route` objects, e.g. for doing content negotiation. It is the responsibility of each filter to throw the `ResourceNotFoundException` if no more routes are left in the collection.

FinalMatcherInterface

The `FinalMatcherInterface` implementation has to determine exactly one Route as the best match or throw an exception if no adequate match could be found. The default implementation uses the *UrlMatcher*[11] of the Symfony Routing Component.

Route Enhancers

Optionally, and following the matching process, a set of `RouteEnhancerInterface` instances can be applied by the `DynamicRouter`. The aim of these is to allow you to manipulate the parameters from the matched route. They can be used, for example, to dynamically assign a controller or template to a `Route` or to "upcast" a request parameter to an object. Some simple Enhancers are already packed with the bundle, documentation can be found inside each class file.

---

8. http://api.symfony.com/master/Symfony/Component/HttpFoundation/Request.html
9. http://api.symfony.com/master/Symfony/Component/Routing/Route.html
10. https://github.com/symfony-cmf/RoutingExtraBundle
11. http://api.symfony.com/2.1/Symfony/Component/Routing/Matcher/UrlMatcher.html

Linking a Route with a Content

Depending on your application's logic, a requested url may have an associated content from the database. Those Routes should implement the `RouteObjectInterface`, and can optionally return a model instance. If you configure the `RouteContentEnhancer`, it will included that content in the match array, with the `_content` key. Notice that a Route can implement the above mentioned interface but still not to return any model instance, in which case no associated object will be returned.

Furthermore, routes that implement this interface can also provide a custom Route name. The key returned by `getRouteKey` will be used as route name instead of the Symfony core compatible route name and can contain any characters. This allows you, for example, to set a path as the route name. Both UrlMatchers provided with the NestedMatcher replace the _route key with the route instance and put the provided name into _route_name.

All routes still need to extend the base class `Symfony\Component\Routing\Route`.

Redirections

You can build redirections by implementing the `RedirectRouteInterface`. It can redirect either to an absolute URI, to a named Route that can be generated by any Router in the chain or to another Route object provided by the Route.

Notice that the actual redirection logic is not handled by the bundle. You should implement your own logic to handle the redirection. For an example on implementing that redirection under the full Symfony2 stack, refer to *RoutingExtraBundle*.

Generating URLs

Apart from matching an incoming request to a set of parameters, a Router is also responsible for generating an URL from a Route and its parameters. The `ChainRouter` iterates over its known routers until one of them is able to generate a matching URL.

Apart from using `RequestMatcherInterface` or `UrlMatcherInterface` to match a Request/URL to its corresponding parameters, the `DynamicRouter` also uses an `UrlGeneratorInterface` instance, which allows it to generate an URL from a Route.

The included `ProviderBasedGenerator` extends Symfony2's default *UrlGenerator*[12] (which, in turn, implements `UrlGeneratorInterface`) and - if $name is not already a `Route` object - loads the route from the `RouteProviderInterface`. It then lets the core logic generate the URL from that Route.

The bundle also include the `ContentAwareGenerator`, which extends the `ProviderBasedGenerator` to check if $name is an object implementing `RouteAwareInterface` and, if so, gets the Route from the content. Using the `ContentAwareGenerator`, you can generate urls for your content in three ways:

- Either pass a `Route` object as $name
- Or pass a `RouteAwareInterface` object that is your content as $name
- Or provide an implementation of `ContentRepositoryInterface` and pass the id of the content object as parameter `content_id` and `null` as $name.

ContentAwareGenerator and locales

You can use the `_locale` default value in a Route to create one Route per locale, all referencing the same multilingual content instance. The `ContentAwareGenerator` respects the `_locale` when generating routes from content instances. When resolving the route, the `_locale` gets into the request and is picked up by the Symfony2 locale system.

12. `http://api.symfony.com/master/Symfony/Component/Routing/Generator/UrlGenerator.html`

Under PHPCR-ODM, Routes should never be translatable documents, as one Route document represents one single url, and serving several translations under the same url is not recommended.

If you need translated URLs, make the locale part of the route name.

Customization

The Routing bundles allows for several customization options, depending on your specific needs:

- You can implement your own RouteProvider to load routes from a different source
- Your Route parameters can be easily manipulated using the existing Enhancers
- You can also add your own Enhancers to the DynamicRouter
- You can add RouteFilterInterface instances to the NestedMatcher
- The `DynamicRouter` or its components can be extended to allow modifications
- You can implement your own Routers and add them to the `ChainRouter`

If you feel like your specific Enhancer or Router can be useful to others, get in touch with us and we'll try to include it in the bundle itself

Symfony2 integration

Like mentioned before, this bundle was designed to only require certain parts of Symfony2. However, if you wish to use it as part of your Symfony CMF project, an integration bundle is also available. We strongly recommend that you take a look at *RoutingExtraBundle*.

For a starter's guide to the Routing bundle and its integration with Symfony2, refer to *Routing*

We strongly recommend reading Symfony2's *Routing*[13] component documentation page, as it's the base of this bundle's implementation.

Authors

- Filippo De Santis (p16)
- Henrik Bjornskov (henrikbjorn)
- Claudio Beatrice (omissis)
- Lukas Kahwe Smith (lsmith77)
- David Buchmann (dbu)
- Larry Garfield (Crell)
- *And others*[14]

The original code for the chain router was contributed by Magnus Nordlander.

13. `http://symfony.com/doc/current/components/routing/introduction.html`
14. `https://github.com/symfony-cmf/Routing/contributors`

Chapter 27
Contributing

The Symfony2 CMF team follows all the rules and guidelines of the core Symfony2 *development process*[1].

Resources / Links

- *GitHub*[2]
- *Website*[3]
- *Wiki*[4]
- *Issue Tracker*[5]
- *IRC channel*[6]
- *Users mailing list*[7]
- *Devs mailing list*[8]

1. http://symfony.com/doc/current/contributing/index.html
2. https://github.com/symfony-cmf
3. http://cmf.symfony.com/
4. https://github.com/symfony-cmf/symfony-cmf/wiki
5. http://github.com/symfony-cmf/symfony-cmf/issues
6. #cmf-contributing-irc:--freenode--symfony-cmf
7. http://groups.google.com/group/symfony-cmf-users
8. http://groups.google.com/group/symfony-cmf-devs

Chapter 28
Licensing

The Symfony2 CMF aims to provide liberal open source licenses for its entire stack.

Code

The code stack is covered by the *Apache license*[1] for Jackalope and PHPCR, the rest of the stack, notably the Symfony2 code, PHPCR-ODM, create.js and Hallo.js are *MIT licensed*[2]. Please refer to the relevant LICENSE files in the given source packages.

Documentation

The Symfony2 documentation is licensed under a Creative Commons *Attribution-Share Alike 3.0 Unported License*[3].

You are free:

- to *Share* — to copy, distribute and transmit the work;
- to *Remix* — to adapt the work.

Under the following conditions:

- *Attribution* — You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work);
- *Share Alike* — If you alter, transform, or build upon this work, you may distribute the resulting work only under the same or similar license to this one.

With the understanding that:

- *Waiver* — Any of the above conditions can be waived if you get permission from the copyright holder;

1. http://en.wikipedia.org/wiki/Apache_license
2. http://en.wikipedia.org/wiki/MIT_License
3. http://creativecommons.org/licenses/by-sa/3.0/

- *Public Domain* — Where the work or any of its elements is in the public domain under applicable law, that status is in no way affected by the license;
- *Other Rights* — In no way are any of the following rights affected by the license:

    - Your fair dealing or fair use rights, or other applicable copyright exceptions and limitations;
    - The author's moral rights;
    - Rights other persons may have either in the work itself or in how the work is used, such as publicity or privacy rights.

- *Notice* — For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to this web page.

This is a human-readable summary of the *Legal Code (the full license)*[4].

4. `http://creativecommons.org/licenses/by-sa/3.0/legalcode`