1. Getting started

With the introduction of Symfony 2 as the framework powering eZ Publish 5, the whole eZ Publish 4.x extensions system is changing. Pretty much everything needs to be done with entirely new concepts. In this chapter, we will see two ways of customizing eZ Publish 5: command line scripts (for import scripts, for instance), and custom controllers, the eZ Publish 5 equivalent of eZ Publish 4.x custom modules.

Symfony bundle

In order to test and use Public API code, you will need to build a custom bundle. Bundles are Symfony's extensions, and are therefore also used to extend eZ Publish. Symfony 2 provides code generation tools that will let you create your own bundle and get started in a few minutes.

In this chapter, we will show how to create a custom bundle, and implement both: a command line script and a custom route with its own controller action and view. All shell commands assume that you use some Linux shell, but those commands would of course also work on Windows systems.

Generating a new bundle

First, change the directory to your eZ Publish root.

```
$ cd /path/to/ezpublish5
```

Then use the app/console application with the `generate:bundle` command to start the bundle generation wizard.

Let's follow the instructions provided by the wizard. Our objective is to create a bundle named `EzSystems/Bundles/CookBookBundle`, located in the `src` directory.

```
$ php ezpublish/console generate:bundle
```

The wizard will first ask about our bundle's namespace. Each bundle's namespace should feature a vendor name (in our own case: `EzSystems`), optionally followed by a sub-namespace (we could have chosen to use `Bundle`), and end with the actual bundle's name, suffixed with Bundle: `Co okbookBundle`.

Bundle namespace

```
Your application code must be written in bundles. This command helps you generate them
easily.

Each bundle is hosted under a namespace (like Acme/Bundle/BlogBundle).

The namespace should begin with a "vendor" name like your company name, your project
name, or your client name, followed by one or more optional category sub-namespaces,
and it should end with the bundle name itself (which must have Bundle as a suffix).

See http://symfony.com/doc/current/cookbook/bundles/best_practices.html#index-1 for
more details on bundle naming conventions.

Use / instead of \ for the namespace delimiter to avoid any problem.

Bundle namespace: EzSystems/CookbookBundle
```

You will then be asked about the Bundle's name, used to reference your bundle in your code. We can go with the default, `EzSystemsCookbookBundle`. Just hit enter to accept the default.

Bundle name

```
In your code, a bundle is often referenced by its name. It can be the concatenation of
all namespace parts but it's really up to you to come up with a unique name (a good
practice is to start with the vendor name).

Based on the namespace, we suggest EzSystemsCookbookBundle.

Bundle name [EzSystemsCookbookBundle]:
```

The next question is your bundle's location. By default, the script offers to place it in the `src` folder. This is perfectly acceptable unless you have a good reason to place it somewhere else. Just hit enter to accept the default.

Bundle directory

```
The bundle can be generated anywhere. The suggested default directory uses the
standard conventions.


Target directory [/path/to/ezpublish5/src]:
```

Next, you need to choose the generated configuration's format, out of YAML, XML, PHP or annotations. We mostly use yaml in eZ Publish 5, and we will use it in this cookbook. Enter 'yml', and hit enter.

Configuration format

```
Determine the format to use for the generated configuration.


Configuration format (yml, xml, php, or annotation) [annotation]: yml
```

The last choice is to generate code snippets demonstrating the Symfony directory structure. If you're learning Symfony, it is a good idea to accept, as it will pre-create a controller, yaml files, etc.

Generate snippets & directory structure

```
To help you get started faster, the command can generate some code snippets for you.

Do you want to generate the whole directory structure [no]? yes
```

The generator will then summarize the previous choices, and ask for confirmation. Hit enter to confirm.

Summary and confirmation

```
You are going to generate a "EzSystems\Bundle\CookbookBundle\EzSystemsCookbookBundle"
bundle in "/path/to/ezpublish5/src/" using the "yml" format.

Do you confirm generation [yes]? yes
```

The wizard will generate the bundle, check autoloading, and ask about the activation of your bundle. Hit enter to both questions to have your bundle automatically added to your Kernel (`ezpublish/EzPublishKernel.php`) and routes from your bundle added to the existing routes (`ezpublish/config/routing.yml`).

Activation and generation

```
Bundle generation

Generating the bundle code: OK
Checking that the bundle is autoloaded: OK
Confirm automatic update of your Kernel [yes]?
Enabling the bundle inside the Kernel: OK
Confirm automatic update of the Routing [yes]?
Importing the bundle routing resource: OK

  You can now start using the generated code!
```

Your bundle should be generated and activated. Let's now see how you can interact with the Public API by creating a command line script, and a custom controller route and action.

Creating a command line script in your bundle

Writing a command line script with Symfony 2 is *very* easy. The framework and its bundles ship with a few scripts. They are all started using `php ezpublish/console <command>` (`app/console` in a default symfony 2 application). You can get the complete list of existing command line scripts by executing `php ezpublish/console list` from the eZ Publish 5 root.

In this chapter, we will create a new command, identified as `ezpublish:cookbook:hello`, that takes an optional name argument, and greets that name. To do so, we need one thing: a class with a name ending with "Command" that extends `Symfony\Component\Console\Command\Command`. Note that in our case, we use `ContainerAwareCommand` instead of `Command`, since we need the dependency injection container to interact with the Public API). In your bundle's directory (`src/EzSystems/CookbookBundle`), create a new directory named `Command`, and in this directory, a new file named `HelloCommand.php`.

Add this code to the file:

HelloCommand.php

```
<?php
namespace EzSystems\CookBookBundle\Command;

use Symfony\Component\Console\Input\InputInterface;
use Symfony\Component\Console\Output\OutputInterface;
use Symfony\Component\Console\Input\InputArgument;

class HelloCommand extends
\Symfony\Bundle\FrameworkBundle\Command\ContainerAwareCommand
{
    /**
     * Configures the command
     */
    protected function configure()
    {
    }

    /**
     * Executes the command
     * @param InputInterface $input
     * @param OutputInterface $output
     */
    protected function execute( InputInterface $input, OutputInterface $output )
    {
    }
}
```

This is the skeleton for a command line script.

One class with a name ending with "Command" (`HelloCommand`), extends `Symfony\Bundle\FrameworkBundle\Command\Command`, and is part of our bundle's Command namespace. It has two methods: `configure()`, and `execute()`. We also import several classes & interfaces with the use keyword. The first two, `InputInterface` and `OutputInterface` are used to 'typehint' the objects that will allow us to provide input & output management in our script.

Configure will be used to set your command's name, as well as its options and arguments. Execute will contain the actual implementation of your command. Let's start by creating the `configure()` method.

TestCommand::configure()

```
protected function configure()
{
    $this->setName( 'ezpublish:cookbook:hello' );
    $this->setDefinition(
        array(
            new InputArgument( 'name', InputArgument::OPTIONAL, 'An argument' )
        )
    );
}
```

First, we use setName() to set our command's name to "`ezpublish:cookbook:hello`". We then use `setDefinition()` to add an argument, named `name`, to our command.

You can read more about arguments definitions and further options in the Symfony 2 Console documentation. Once this is done, if you run `php ezpublish/console list`, you should see `ezpublish:cookbook:hello` listed in the available commands. If you run it, it should just do nothing.

Let's just add something very simple to our execute() method so that our command actually does something.

TestCommand::execute()

```
protected function execute( InputInterface $input, OutputInterface $output )
{
    // fetch the input argument
    if ( !$name = $input->getArgument( 'name' ) )
    {
        $name = "World";
    }
    $output->writeln( "Hello $name" );
}
```

You can now run the command from the eZ Publish 5 root.

Hello world

```
$ php ezpublish/console ezpublish:cookbook:hello world
Hello world
```

Creating a custom route with a controller action

In this short chapter, we will see how to create a new route that will catch a custom URL and execute a controller action. We want to create a new route, `/cookbook/test`, that displays a simple 'Hello world' message. This tutorial is a simplified version of the official one that can be found on http://symfony.com/doc/current/book/controller.html.

> **eZ Publish 4 equivalent**
> This eZ Publish 5 extension would have been a custom module, with its own `module.php` file, in eZ Publish 4.

During our bundle's generation, we have chosen to generate the bundle with default code snippets. Fortunately, almost everything we need is part of those default snippets. We just need to do some editing, in particular in two locations: `src/EzSystems/CookbookBundle/Resources/config/routing.yml` and `src/EzSystems/CookbookBundle/Controllers/DefaultController.php`. The first one will be used to configure our route (`/cookbook/test`) as well as the controller action the route should execute, while the latter will contain the actual action's code.

routing.yml

This is the file where we define our action's URL matching. The generated file contains this YAML block

Generated routing.yml

```
ez_systems_cookbook_homepage:
    pattern:  /hello/{name}
    defaults: { _controller: EzSystemsCookbookBundle:Default:index }
```

We can safely remove this default code, and replace it with this

Edited routing.yml

```
ezsystems_cookbook_hello:
    pattern:  /cookbook/hello/{name}
    defaults: { _controller: EzSystemsCookbookBundle:Default:hello }
```

We define a route that matches the URI /cookbook/* and executes the action hello in the Default controller of our bundle. The next step is to create this method in the controller.

DefaultController.php

This controller was generated by the bundle generator. It contains one method, `helloAction()`, that matched the YAML configuration we have changed in the previous part. Let's just rename the `indexAction()` method so that we end up with this code.

DefaultController::helloAction()

```
public function helloAction( $name )
{
    $response = new \Symfony\Component\HttpFoundation\Response;
    $response->setContent( "Hello $name" );
    return $response;
}
```

We won't go into details about controllers in this cookbook, but let's walk through the code a bit. This method receives the parameter defined in `routing.yml`. It is named "name" in the route definition, and must be named $name in the matching action. Since the action is named "hello" in `routing.yml`, the expected method name is `helloAction`.

Controller actions **must** return a Response object that will contain the response's content, the headers, and various optional properties that affect the action's behavior. In our case, we simply set the content, using `setContent()`, to "Hello $name". Simple. Go to http://ezpublish5/cookbook/hello/YourName, and you should get "Hello YourName".

The custom EzPublishCoreBundle Controller
For convenience, a custom controller is available at eZ\Bundle\EzPublishCoreBundle\Controller. It gives you with a few commodity methods:

- getRepository()
  Returns the Public API repository, that gives you access to the various services through getContentService(), getLocationService() and so on;
- getLegacyKernel()
  Returns an instance of the eZ\Publish\Core\MVC\Legacy\Kernel, that you can use to interact with the Legacy eZ Publish kernel
- getConfigResolver()
  Returns the ConfigResolver that gives you access to configuration data.

You are encouraged to use it for your custom controllers that interact with eZ Publish.

With both command line scripts and HTTP routes, you have the basics you need to start writing Public API code.