



Symfony

tutorialspoint

SIMPLY EASY LEARNING

www.tutorialspoint.com



<https://www.facebook.com/tutorialspointindia>



<https://twitter.com/tutorialspoint>

About the Tutorial

Symfony is an open-source PHP web application framework, designed for developers who need a simple and elegant toolkit to create full-featured web applications. Symfony is sponsored by SensioLabs. It was developed by Fabien Potencier in 2005. This tutorial will give you a quick introduction to Symfony framework and make you comfortable with its various components.

Audience

This tutorial has been prepared for beginners who want to learn the fundamental concepts of Symfony framework. The readers will get enough understanding on how to create and develop a website using Symfony.

Prerequisites

Before proceeding with the various types of components given in this tutorial, it is being assumed that the readers are already aware about what a Framework is. In addition to this, it will also be very helpful if you have a sound knowledge on HTML, PHP, and the OOPS concepts.

Copyright & Disclaimer

© Copyright 2017 by Tutorials Point (I) Pvt. Ltd.

All the content and graphics published in this e-book are the property of Tutorials Point (I) Pvt. Ltd. The user of this e-book is prohibited to reuse, retain, copy, distribute or republish any contents or a part of contents of this e-book in any manner without written consent of the publisher.

We strive to update the contents of our website and tutorials as timely and as precisely as possible, however, the contents may contain inaccuracies or errors. Tutorials Point (I) Pvt. Ltd. provides no guarantee regarding the accuracy, timeliness or completeness of our website or its contents including this tutorial. If you discover any errors on our website or in this tutorial, please notify us at contact@tutorialspoint.com

Table of Contents

About the Tutorial.....	i
Audience	i
Prerequisites	i
Copyright & Disclaimer	i
Table of Contents	ii
 1. SYMFONY – INTRODUCTION.....	 1
 2. SYMFONY – INSTALLATION	 3
System Requirements	3
Symfony Installer	3
Composer-based Installation.....	5
Running the Application.....	5
 3. SYMFONY – ARCHITECTURE	 6
Web Framework.....	7
 4. SYMFONY – COMPONENTS	 9
Installing a Symfony Component.....	9
Details of Symfony Components	10
 5. SYMFONY – SERVICE CONTAINER	 23
 6. SYMFONY – EVENTS & EVENTLISTENER.....	 30
 7. SYMFONY – EXPRESSION	 35
 8. SYMFONY – BUNDLES.....	 37
Structure of a Bundle	37
Creating a Bundle	37

9. SYMFONY – CREATING A SIMPLE WEB APPLICATION.....	40
Controller	40
Create a Route	41
10. SYMFONY – CONTROLLERS.....	42
Request Object.....	42
Response Object	43
FrontController	44
11. SYMFONY – ROUTING.....	46
Annotations	46
Routing Concepts	46
Redirecting to a Page	49
12. SYMFONY – VIEW ENGINE	51
Templates	51
Twig Engine	51
Layouts.....	58
Assets.....	58
13. SYMFONY – DOCTRINE ORM	60
Database Model	60
Doctrine ORM	60
Doctrine ORM Example	60
14. SYMFONY – FORMS	71
Form Fields.....	71
Form Helper Function.....	76
Student Form Application	77

15. SYMFONY – VALIDATION	85
Validation Constraints	85
Validation Example	89
16. SYMFONY – FILE UPLOADING	96
17. SYMFONY – AJAX CONTROL.....	102
AJAX – Working Example	102
18. SYMFONY – COOKIES & SESSION MANAGEMENT	106
Cookie	106
Session	107
19. SYMFONY – INTERNATIONALIZATION	108
20. SYMFONY – LOGGING.....	111
21. SYMFONY – EMAIL MANAGEMENT	112
22. SYMFONY – UNIT TESTING	114
PHPUnit	114
Unit test	114
23. SYMFONY – ADVANCED CONCEPTS	116
HTTP Cache	116
Debug	119
Profiler	119
Security	120
Workflow	126
24. SYMFONY – SYMFONY REST EDITION	131

25. SYMFONY – SYMFONY CMF EDITION.....	132
26. SYMFONY – COMPLETE WORKING EXAMPLE	134

1. Symfony – Introduction

A PHP web framework is a collection of classes, which helps to develop a web application. Symfony is an open-source MVC framework for rapidly developing modern web applications. Symfony is a full-stack web framework. It contains a set of reusable PHP components. You can use any Symfony components in applications, independently from the framework.

Symfony has a huge amount of functionality and active community. It has a flexible configuration using YAML, XML, or annotations. Symfony integrates with an independent library and PHP Unit. Symfony is mainly inspired by Ruby on Rails, Django, and Spring web application frameworks. Symfony components are being used by a lot of open source projects that include Composer, Drupal, and phpBB.

The Symfony framework consists of several components, such as the HttpFoundation component that understands HTTP and offers a nice request and response object used by the other components. Others are merely helper components, such as the Validator, that helps to validate data. Kernel component is the heart of the system. Kernel is basically the 'main class' that manages the environment and has the responsibility of handling a http request.

Symfony's well-organized structure, clean code, and good programming practices make web development easier. Symfony is very flexible, used to build micro-sites and handle enterprise applications with billions of connections.

Symfony Framework – Features

Symfony is designed to optimize the development of web applications and grows in features with every release.

Some of the salient features of Symfony Framework is as follows:

- Model-View-Controller based system
- High-performance PHP framework
- Flexible URI routing
- Code reusable and easier to maintain
- Session management
- Error logging
- Full-featured database classes with support for several platforms
- Supports a huge and active community
- Set of decoupled and reusable components
- Standardization and interoperability of applications
- Security against cross-site request forgery and other attacks
- Twig template engine.

Symfony offers a lot of flexibility to developers. It has great features for debugging, code readability, and developing extensible programs.

Symfony is a full-stack web framework; it is a very effective tool for creating web applications. Numerous companies offer Symfony services to clients.

Following are some of the benefits that you get by using the Symfony Framework.

- **Microframework** - Symfony can be used to develop a specific functionality. You don't need to redevelop or install the entire framework.
- Reduces development time overhead.
- Extremely mature templating engine and quickly delivers content to the users.
- **Compatible and extensible** – Programmers can easily extend all framework classes.

Symfony Framework – Applications

Symfony components can be used as a part of other applications such as Drupal, Laravel, phpBB, Behat, Doctrine, and Joomla.

- **Drupal 8** – Drupal is an open source content management PHP framework. Drupal 8 uses core layers of Symfony and extends it to provide support for Drupal modules.
- **Thelia** – Thelia is a Symfony-based e-commerce solution. Initially, Thelia was written in PHP code and MySQL, however, it was lagging to produce faster applications. To overcome this drawback, Thelia integrated with Symfony to develop the applications in a customizable way.
- **Dailymotion** – Dailymotion is one of the world's largest independent video entertainment website based in France. Once they decided to migrate open source framework with a large community, Dailymotion developers decided to use Symfony components features for its flexibility.

2. Symfony – Installation

This chapter explains how to install Symfony framework on your machine. Symfony framework installation is very simple and easy. You have two methods to create applications in Symfony framework. First method is using Symfony Installer, an application to create a project in Symfony framework. Second method is composer-based installation. Let's go through each of the methods one by one in detail in the following sections.

System Requirements

Before moving to installation, you require the following system requirements.

- Web server (Any one of the following)
 - WAMP (Windows)
 - LAMP (Linux)
 - XAMP (Multi-platform)
 - MAMP (Macintosh)
 - Nginx (Multi-platform)
 - Microsoft IIS (Windows)
 - PHP built-in development web server (Multi-platform)
- Operating System: Cross-platform
- Browser Support: IE (Internet Explorer 8+), Firefox, Google Chrome, Safari, Opera
- PHP Compatibility: PHP 5.4 or later. To get the maximum benefit, use the latest version.

We will use PHP built-in development web server for this tutorial.

Symfony Installer

Symfony Installer is used to create web applications in Symfony framework. Now, let's configure the Symfony installer using the following command.

```
$ sudo mkdir -p /usr/local/bin  
  
$ sudo curl -LsS https://symfony.com/installer -o /usr/local/bin/symfony  
  
$ sudo chmod a+x /usr/local/bin/symfony
```

Now, you have installed Symfony installer on your machine.

Create Your First Symfony Application

Following syntax is used to create a Symfony application in the latest version.

Syntax

```
symfony new app_name
```

Here, app_name is your new application name. You can specify any name you want.

Example

```
symfony new HelloWorld
```

After executing the above command, you will see the following response.

```
Downloading Symfony...
```

```
0 B/5.5 MiB ██████████
```

```
.....
```

```
.....
```

```
Preparing project...
```

```
✓  Symfony 3.2.7 was successfully installed. Now you can:
```

```
*  Change your current directory to /Users/../../workspace/firstapp
```

```
*  Configure your application in app/config/parameters.yml file.
```

```
*  Run your application:
```

```
    1. Execute the php bin/console server:run command.
```

```
    2. Browse to the http://localhost:8000 URL.
```

```
*  Read the documentation at http://symfony.com/doc
```

This command creates a new directory called "firstapp/" that contains an empty project of Symfony framework latest version.

Install Specific Version

If you need to install a specific Symfony version, use the following command.

```
symfony new app_name 2.8  
symfony new app_name 3.1
```

Composer-based Installation

You can create Symfony applications using the Composer. Hopefully, you have installed the composer on your machine. If the composer is not installed, download and install it.

The following command is used to create a project using the composer.

```
$ composer create-project symfony/framework-standard-edition app_name
```

If you need to specify a specific version, you can specify in the above command.

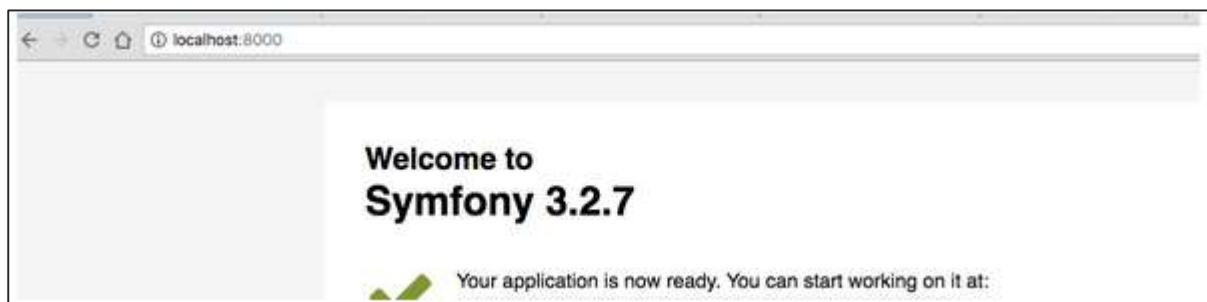
Running the Application

Move to the project directory and run the application using the following command.

```
cd HelloWorld  
php bin/console server:run
```

After executing the above command, open your browser and request the url <http://localhost:8000/>. It produces the following result.

Result

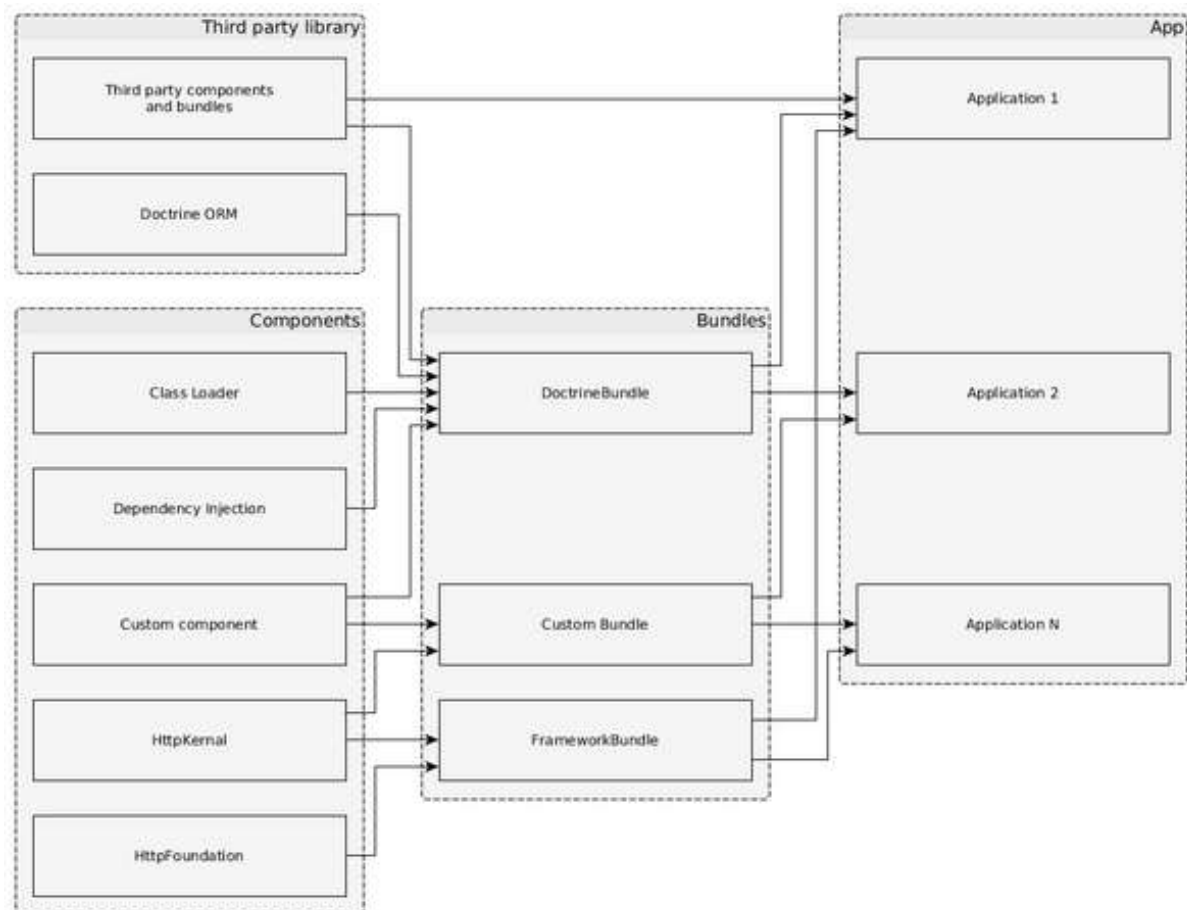


3. Symfony – Architecture

Symfony is basically a collection of high quality components and bundles. Components are collection of classes providing a single core functionality. For example, **Cache component** provides cache functionality, which can be added to any application. Components are building blocks of a Symfony application. Symfony has 30+ high quality components, which are used in many PHP framework such as Laravel, Silex, etc.

Bundles are similar to plugin but easy to create and easy to use. Actually, a Symfony application is itself a bundle composed of other bundles. A single bundle can use any number of Symfony component and also third-party components to provide features such as Webframework, database access, etc. Symfony core web-framework is a bundle called FrameworkBundle and there is a bundle called FrameworkExtraBundle, which provides more sophisticated options to write a web application.

The relationship between the Components, Bundles, and Symfony application is specified in the following diagram.



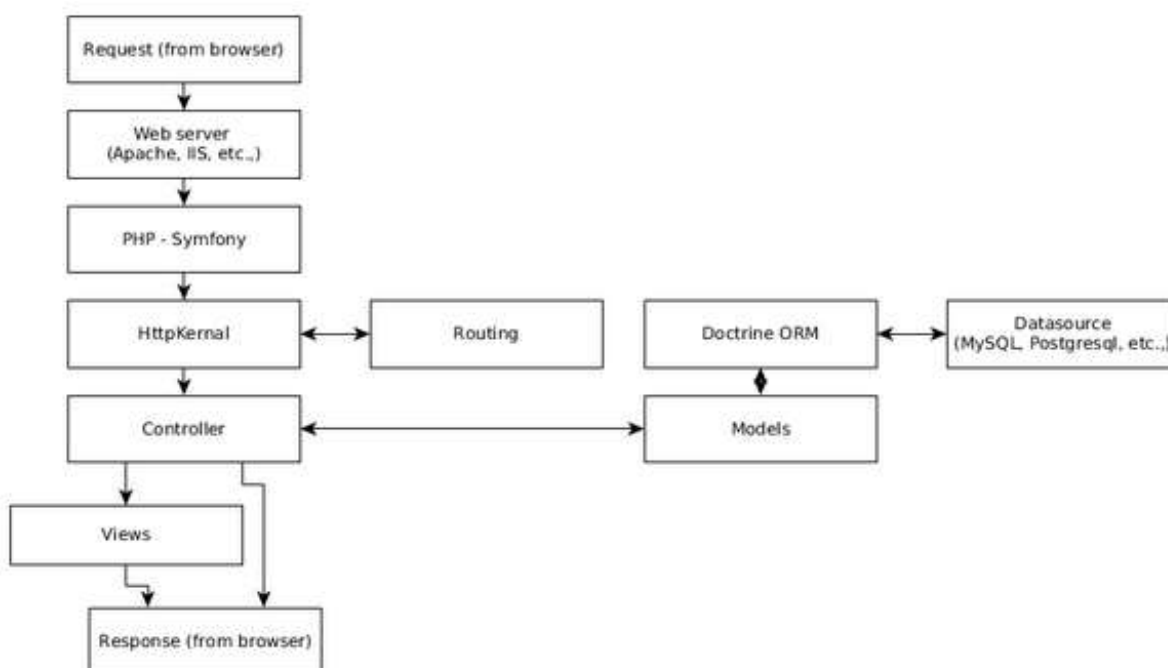
Web Framework

Symfony is mainly designed to write high-quality web application with relative ease. It provides various options to write different types of web application from simple web site to advanced REST based web services. Symfony provides web framework as separate bundles. The common bundles used in Symfony web framework are as follows:

- FrameworkBundle
- FrameworkExtraBundle
- DoctrineBundle

Symfony web framework is based on Model-View-Controller (MVC) architecture. **Model** represents the structure of our business entities. **View** shows the models to the user in the best possible way depending on the situation. **Controller** handles all the request from the user, does the actual work by interacting with Model and finally provides the View with the necessary data to show it to the user.

Symfony web framework provides all the high-level features required for an enterprise-grade application. Following is a simple workflow of Symfony web application.



The workflow consists of the following steps.

Step 1: The user sends a request to the application through the browser, say <http://www.symfonyexample.com/index>.

Step 2: The browser will send a request to the web server, say Apache web server.

Step 3: The web server will forward the request to the underlying PHP, which in turn sends it to Symfony web framework.

Step 4: HttpKernel is the core component of the Symfony web framework. HttpKernel resolves the controller of the given request using Routing component and forward the request to the target controller.

Step 5: All the business logic takes place in the target controller.

Step 6: The controller will interact with Model, which in turn interacts with Datasource through Doctrine ORM.

Step 7: Once the controller completes the process, it either generates the response itself or through View Engine, and sends it back to the web server.

Step 8: Finally, the response will be sent to the requested browser by the web server.

4. Symfony – Components

As discussed earlier, Symfony components are standalone PHP library providing a specific feature, which can be used in any PHP application. Useful new components are being introduced in each and every release of Symfony. Currently, there are 30+ high quality components in Symfony framework. Let us learn about the usage of Symfony components in this chapter.

Installing a Symfony Component

Symfony components can be installed easily using the composer command. Following generic command can be used to install any Symfony component.

```
cd /path/to/project/dir
composer require symfony/<component_name>
```

Let us create a simple php application and try to install **Filesystem** component.

Step 1: Create a folder for the application, **filesystem-example**

```
cd /path/to/dev/folder
mkdir filesystem-example
cd filesystem-example
```

Step 2: Install Filesystem component using the following command.

```
composer require symfony/filesystem
```

Step 3: Create a file **main.php** and enter the following code.

```
<?php
require_once __DIR__ . '/vendor/autoload.php';
use Symfony\Component\Filesystem\Filesystem;
use Symfony\Component\Filesystem\Exception\IOExceptionInterface;
$fs = new Filesystem();
try {
    $fs->mkdir('./sample-dir');
    $fs->touch('./sample-dir/text.txt');
} catch (IOExceptionInterface $e) {
    echo $e;
}
?>
```

The first line is very important, which loads all the necessary classes from all the components installed using the Composer command. The next lines use the Filesystem class.

Step 4: Run the application using the following command and it will create a new folder **sample-dir** and a file **test.txt** under it.

```
php main.php
```

Details of Symfony Components

Symfony provides components ranging from simple feature, say file system to advanced feature, say events, container technology, and dependency injection. Let us know about all the components one by one in the following sections.

Filesystem

Filesystem component provides a basic system command related to files and directories such as file creation, folder creation, file existence, etc. Filesystem component can be installed using the following command.

```
composer require symfony/filesystem
```

Finder

Finder component provides fluent classes to find files and directories in a specified path. It provides an easy way to iterate over the files in a path. Finder component can be installed using the following command.

```
composer require symfony/finder
```

Console

Console component provides various options to easily create commands, which can be executed in a terminal. Symfony uses the **Command** component extensively to provide various functionalities such as creating a new application, creating a bundle, etc. Even the PHP build in web server can be invoked using Symfony command, **php bin/console server:run** as seen in the installation section. The **Console** component can be installed using the following command.

```
composer require symfony/console
```

Let us create a simple application and create a command, **HelloCommand** using the **Console** component and invoke it.

Step 1: Create a project using the following command.

```
cd /path/to/project  
composer require symfony/console
```


Step 2: Create a file **main.php** and include the following code.

```
<?php
require __DIR__ . '/vendor/autoload.php';
use Symfony\Component\Console\Application;
$app = new Application();
$app->run();
?>
```

Application class sets up the necessary functionality of a bare-bone console application.

Step 3: Run the application, **php main.php**, which will produce the following result.

```
Console Tool

Usage:
    command [options] [arguments]

Options:
    -h, --help            Display this help message
    -q, --quiet            Do not output any message
    -V, --version          Display this application version
        --ansi            Force ANSI output
        --no-ansi         Disable ANSI output
    -n, --no-interaction  Do not ask any interactive question
    -v|vv|vvv, --verbose  Increase the verbosity of messages: 1 for normal
output, 2 for more verbose output and 3 for debug

Available commands:
    help  Displays help for a command
    list  Lists commands
```

Step 4: Create a class called **HelloCommand** extending **Command** class in the **main.php** itself.

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

```
class HelloCommand extends Command
{
}
```

The application uses following four classes available in **Command** component.

- **Command** - Used to create a new command
- **InputInterface** - Used to set user inputs
- **InputArgument** - Used to get user inputs
- **OutputInterface** - Used to print output to the console

Step 5: Create a function **configure()** and set name, description, and help text.

```
protected function configure()
{
    $this
        ->setName('app:hello')
        ->setDescription('Sample command, hello')
        ->setHelp('This command is a sample command')
}
```

Step 6: Create an input argument, **user** for the command and set as mandatory.

```
protected function configure()
{
    $this
        ->setName('app:hello')
        ->setDescription('Sample command, hello')
        ->setHelp('This command is a sample command')
        ->addArgument('name', InputArgument::REQUIRED, 'name of the user');
}
```

Step 7: Create a function **execute()** with two arguments **InputArgument** and **OutputArgument**.

```
protected function execute(InputInterface $input, OutputInterface $output)
{
}
```

Step 8: Use **InputArgument** to get the user details entered by the user and print it to the console using **OutputArgument**.

```
protected function execute(InputInterface $input, OutputInterface $output)
{
    $name = $input->getArgument('name');
    $output->writeln('Hello, ' . $name);
}
```

Step 9: Register the **HelloCommand** into the application using the **add** method of **Application** class.

```
$app->add(new HelloCommand());
```

The complete application is as follows.

```
<?php
require __DIR__ . '/vendor/autoload.php';
use Symfony\Component\Console\Application;
use Symfony\Component\Console\Command\Command;
use Symfony\Component\Console\Input\InputInterface;
use Symfony\Component\Console\Output\OutputInterface;
use Symfony\Component\Console\Input\InputArgument;

class HelloCommand extends Command
{
    protected function configure()
    {
        $this
            ->setName('app:hello')
            ->setDescription('Sample command, hello')
            ->setHelp('This command is a sample command')
            ->addArgument('name', InputArgument::REQUIRED, 'name of the user');
    }

    protected function execute(InputInterface $input, OutputInterface $output)
    {
        $name = $input->getArgument('name');
        $output->writeln('Hello, ' . $name);
    }
}
```

```

}

$app = new Application();
$app->add(new HelloCommand());
$app->run();
?>

```

Step 10: Now, execute the application using the following command and the result will be Hello, Jon as expected.

```
php main.php app:hello Jon
```

Symfony comes with a pre-built binary called **console** in the bin directory of any Symfony web application, which can be used to invoke the commands in an application.

Process

Process component provides options to run any system command in a sub-process, in a safe and efficient manner. Process component can be installed using the following command.

```
composer require symfony/process
```

ClassLoader

ClassLoader component provides implementation for both **PSR-0** and **PSR-4** class loader standard. It can be used to auto-load the classes. It will be depreciated in the near future. Composer-based class loader is preferred over this component. ClassLoader component can be installed using the following command.

```
composer require symfony/class-loader
```

PropertyAccess

PropertyAccess component provides various options to read and write an object and array details using the string notation. For example, an array **Product** with key **price** can be accessed dynamically using **[price]** string.

```

$product = array(
    'name' => 'Cake'
    'price' => 10
);

var priceObj = $propertyAccesserObj->getValue($product, '[price]');

```

PropertyAccess component can be installed using the following command.

```
composer require symfony/property-access
```

PropertyInfo

PropertyInfo component is similar to PropertyAccess component, however it only works with PHP objects and provides much more functionality.

```
class Product
{
    private $name = 'Cake';
    private $price = 10;

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

    public function getPrice()
    {
        return $this->price;
    }
}

$class = Product::class;
$properties = $propertyInfoObj->getProperties($class);

/*
Example Result
-----
array(2) {
    [0] => string(4) "name"
    [1] => string(5) "price"
}
*/
```

PropertyInfo component can be installed using the following command.

```
composer require symfony/property-info
```

EventDispatcher

EventDispatcher component provides an event-based programming in PHP. It enables the objects to communicate with each other by dispatching events and listening to them. We will learn how to create event and listen to them in the Events and Event Listener chapter.

EventDispatcher component can be installed using the following command.

```
composer require symfony/event-dispatcher
```

DependencyInjection

DependencyInjection component provides an easy and efficient mechanism to create an object with its dependency. When a project grows, it features a lot of classes with deep dependency, which needs to be handled correctly. Otherwise, the project fails. DependencyInjection provides a simple and robust container to handle the dependency. We will learn about the containers and the dependency injection concept in Service Container chapter.

DependencyInjection component can be installed using the following command.

```
composer require symfony/dependency-injection
```

Serializer

Serializer component provides an option to convert a PHP object into a specific format such as XML, JSON, Binary, etc., and then allows it to convert it back into an original object without any data loss.

Serializer component can be installed using the following command.

```
composer require symfony/serializer
```

Config

Config component provides options to load, parse, read, and validate configurations of type XML, YAML, PHP and ini. It provides various options to load configuration details from database as well. This is one of the important components useful in configuring web application in a clear and concise manner. Config component can be installed using the following command.

```
composer require symfony/config
```

ExpressionLanguage

ExpressionLanguage component provides a full-fledged expression engine. Expressions are one-liner intended to return a value. The expression engine enables to easily compile, parse, and get the value from an expression. It enables one or more expression to be used in a configuration environment (file) by a non-PHP programmer, say a system administrator. ExpressionLanguage component can be installed using the following command.

```
composer require symfony/expression-language
```

OptionsResolver

OptionsResolver component provides a way to validate the option system used in our system. For example, database setting is placed in an array, dboption with host, username, password, etc., as keys. You need to validate the entries before using it to connect to a database. OptionsResolver simplifies this task by providing a simple class OptionsResolver and a method resolver, which resolves the database setting and if there is any validation issue, it will report it.

```
$options = array(
    'host'      => '<db_host>',
    'username' => '<db_user>',
    'password' => '<db_password>',
);

$resolver = new OptionsResolver();
$resolver->setDefaults(array(
    'host'      => '<default_db_host>',
    'username' => '<default_db_user>',
    'password' => '<default_db_password>',
));

$resolved_options = $resolver->resolve($options);
```

OptionsResolver component can be installed using the following command.

```
composer require symfony/options-resolver
```

Dotenv

Dotenv component provides various options to parse **.env** files and the variable defined in them to be accessible via **getenv()**, **\$_ENV**, or **\$_SERVER**. Dotenv component can be installed using the following command.

```
composer require symfony/dotenv
```

Cache

Cache component provides an extended **PSR-6** implementation. It can be used to add cache functionality to our web application. Since it follows PSR-6, it is easy to get started and it can be easily used in place of another **PSR-6** based cache component. Cache component can be installed using the following command.

```
composer require symfony/cache
```

Intl

Intl component is the replacement library for C Intl extension. Intl component can be installed using the following command.

```
composer require symfony/intl
```

Translation

Translation component provides various options to internationalize our application. Normally, the translation details of different languages will be stored in a file, one file per language, and it will be loaded dynamically during runtime of the application. There are different formats to write a translation file. Translation component provides various options to load any type of format, such as plain PHP file, CSV, ini, Json, Yaml, ICU Resource file, etc. Translation component can be installed using the following command.

```
composer require symfony/translation
```

Workflow

Workflow component provides advanced tools to process a finite state machine. By providing this functionality in a simple and object-oriented way, Workflow component enables advanced programming in PHP with relative ease. We will learn about it in detail in the Advanced Concept chapter.

Workflow component can be installed using the following command.

```
composer require symfony/workflow
```

Yaml

Yaml component provides an option that parses the YAML file format and converts it into PHP arrays. It also able to write YAML file from plain php array. Yaml component can be installed using the following command.

```
composer require symfony/yaml
```


Ldap

Ldap component provides PHP classes to connect to a LDAP or Active directory server and authenticate the user against it. It provides an option to connect to a Windows domain controller. Ldap component can be installed using the following command.

```
composer require symfony/ldap
```

Debug

Debug component provides various options to enable debugging in PHP environment. Normally, debugging PHP code is hard but the debug component provides simple classes to ease the process of debugging and make it clean and structured. Debug component can be installed using the following command.

```
composer require symfony/debug
```

Stopwatch

Stopwatch component provides Stopwatch class to profile our PHP code. A simple usage is as follows.

```
use Symfony\Component\Stopwatch\Stopwatch;

$stopwatch = new Stopwatch();
$stopwatch->start('somename');

// our code to profile
$profiled_data = $stopwatch->stop('somename');

echo $profiled_data->getPeriods()
```

Stopwatch component can be installed using the following command.

```
composer require symfony/stopwatch
```

VarDumper

VarDumper component provides better **dump()** function. Just include the VarDumper component and use the dump function to get the improved functionality. VarDumper component can be installed using the following command.

```
composer require symfony/var-dumper
```

BrowserKit

BrowserKit component provides an abstract browser client interface. It can be used to test web application programmatically. For example, it can request a form, enter the sample

data and submit it to find any issue in the form programmatically. BrowserKit component can be installed using the following command.

```
composer require symfony/browser-kit
```

PHPUnit Bridge

PHPUnit Bridge component provides many options to improve the PHPUnit testing environment. PHPUnit Bridge component can be installed using the following command.

```
composer require symfony/phpunit-bridge
```

Asset

Asset component provides a generic asset handling in a web application. It generates URL for the assets such as CSS, HTML, JavaScript and also performs version maintenance. We will check the asset component in detail in View Engine chapter. Asset component can be installed using the following command.

```
composer require symfony/asset
```

CssSelector

CssSelector component provides an option to convert CSS based Selectors into XPath expression. A web developer knows the CSS based Selectors expression more than XPath expression, but the most efficient expression to find an element in HTML and XML document is **XPath Expression**.

CssSelector enables the developer to write the expression in *CSS Selectors*, however, the component converts it to XPath expression before executing it. Thus, the developer has an advantage of simplicity of CSS Selectors and efficiency of XPath expression.

CssSelector component can be installed using the following command.

```
composer require symfony/css-selector
```

DomCrawler

DomCrawler component provides various options to find the element in HTML and XML document using DOM concept. It also provides option to use XPath expression to find an element. DomCrawler component can be used along with CssSelector component to use CSS selectors instead of XPath expression. DomCrawler component can be installed using the following command.

```
composer require symfony/dom-crawler
```

Form

Form component enables easy creation of form in a web application. We will learn form programming in detail in Form chapter. Form component can be installed using the following command.

```
composer require symfony/form
```

HttpFoundation

HttpFoundation component provides an object-oriented layer to the HTTP specification. By default, PHP provides HTTP request and response details as array-based object such as **\$_GET**, **\$_POST**, **\$_FILES**, **\$_SESSION**, etc. HTTP based functionality such as setting a cookie can be done using simple, plain old function **setCookie()**. HttpFoundation provides all HTTP related functionality in a small set of classes like Request, Response, RedirectResponse, etc., We will learn about these classes in the later chapters.

HttpFoundation component can be installed using the following command.

```
composer require symfony/http-foundation
```

HttpKernel

HttpKernel component is the core component in the Symfony web setup. It provides all the functionalities required for a web application - from receiving the **Request** object to sending back the **Response** object. The complete architecture of the Symfony web application is provided by HttpKernel as discussed in the architecture of a Symfony web framework.

HttpKernel component can be installed using the following command.

```
composer require symfony/http-kernel
```

Routing

Routing component maps the HTTP request to a pre-defined set of configuration variables. Routing decides which part of our application should handle a request. We will learn more about the routing in Routing chapter.

Routing component can be installed using the following command.

```
composer require symfony/filesystem
```

Templating

Templating component provides the necessary infrastructure to build an efficient template system. Symfony uses the Templating component for its View engine implementation. We will learn more about Templating component in View engine chapter.

Templating component can be installed using the following command.

```
composer require symfony/templating
```

Validator

Validator component provides an implementation of **JSR-303 Bean Validation Specification**. It can be used to validate a form in a web environment. We will learn more about Validator in Validation chapter.

Validator component can be installed using the following command.

```
composer require symfony/validator
```

Security

Security component provides complete security system for our web application, be it HTTP basic authentication, HTTP digest authentication, interactive form based authentication, X.509 certification login, etc. It also provides authorization mechanism based on the user role through in-built ACL system. We will learn more in detail in the Advanced Concept chapter.

Security component can be installed using the following command.

```
composer require symfony/security
```

5. Symfony – Service Container

In any application, objects tend to increase as the application grows. As objects increase, the dependency between the objects also increases. Object dependency needs to be handled properly for a successful application.

As discussed in the Components chapter, Symfony provides an easy and efficient component, **DependencyInjection** to handle object dependency. A service container is a container of objects with properly resolved dependency between them. Let us learn how to use DependencyInjection component in this chapter.

Let us create a **Greeter** class. The purpose of the Greeter class is to greet the user as shown in the following example.

```
$greeter = new Greeter('Hi');  
$greeter->greet('Jon'); // print "Hi, Jon"
```

The complete code of the Greeter class is as follows.

```
class Greeter  
{  
    private $greetingText;  
    public function __construct($greetingText)  
    {  
        $this->greetingText = $greetingText;  
    }  
  
    public function greet($name)  
    {  
        echo $this->greetingText . ", " . $name . "\r\n";  
    }  
}
```

Now, let us add Greeter class to the service container. Symfony provides **ContainerBuilder** to create a new container. Once the container is created, Greeter class can be registered into it using the container's register method.

```
use Symfony\Component\DependencyInjection\ContainerBuilder;  
$container = new ContainerBuilder();  
$container  
    ->register('greeter', 'Greeter')  
    ->addArgument('Hi');
```

Here, we have used static argument to specify the greeting text, **Hi**. Symfony provides a dynamic setting of parameter as well. To use a dynamic parameter, we need to choose a name and specify it between **%** and the parameter can be set using container's **setParameter** method.

```
$container = new ContainerBuilder();
$container
    ->register('greeter', 'Greeter')
    ->addArgument('%greeter.text%');

$container->setParameter('greeter.text', 'Hi');
```

We have registered a Greeter class with proper setting. Now, we can ask the container to provide a properly configured Greeter object using the container **get** method.

```
$greeter = $container->get('greeter');
$greeter->greet('Jon'); // prints "Hi, Jon"
```

We have successfully registered a class, Greeter into container, fetched it from the container and used it. Now, let us create another class **User**, which use Greeter class and see how to register it.

```
class User
{
    private $greeter;

    public $name;
    public $age;

    public function setGreeter(\Greeter $greeter)
    {
        $this->greeter = $greeter;
    }

    public function greet()
    {
        $this->greeter->greet($this->name);
    }
}
```

The User class gets the *Greeter* class using one of its setter method, **setGreeter**. For this scenario, Symfony provides a method, **addMethodCall** and a class, **Reference** to refer another class as shown in the following code.

```
use Symfony\Component\DependencyInjection\Reference;

$container
    ->register('user', 'User')
    ->addMethodCall('setGreeter', array(new Reference('greeter')));
```

Finally, we have registered two classes, **Greeter** and **User** having a strong relation between them. Now, we can safely fetch the User object with properly configured Greeter class from the container as shown in the following code.

```
$container->setParameter('greeter.text', 'Hi');
$user = $container->get('user');
$user->name = "Jon";
$user->age = 20;
$user->greet(); // Prints "Hi, Jon"
```

We have seen how to configure an object in a container using PHP itself. Symfony provides other mechanisms as well. They are XML and YAML configuration files. Let us see how to configure a container using YAML. For this, install **symfony/config** and **symfony/yaml** components along with **symfony/dependency-injection** components.

```
cd /path/to/dir
mkdir dependency-injection-example
cd dependency-injection-example
composer require symfony/dependency-injection
composer require symfony/config
composer require symfony/yaml
```

YAML configuration will be written in a separate file, **services.yml**. YAML configuration consists of two sections, **parameters** and **services**. Parameters section defines all required parameters. Services section defines all objects. Services section is further divided into multiple sections namely, **class**, **arguments**, and **calls**. Class specifies the actual class. Arguments specifies the constructor's arguments. Finally, calls specify the setter methods. Another class can be referred using @ symbol, @greeter.

```
parameters:
    greeter.text: 'Hello'
services:
    greeter:
        class: Greeter
```

```

    arguments: ['%greeter.text%']
user:
    class: User
    calls:
        - [setGreeter, ['@greeter']]

```

Now, **services.yml** can be loaded and configured using **FileLoader** and **YamlFileLoader** as shown in the following code.

```

use Symfony\Component\Config\FileLocator;
use Symfony\Component\DependencyInjection\Loader\YamlFileLoader;

$yamlContainer = new ContainerBuilder();
$loader = new YamlFileLoader($yamlContainer, new FileLocator(__DIR__));
$loader->load('services.yml');

$yamlUser = $yamlContainer->get('user');
$yamlUser->name = "Jon";
$yamlUser->age = 25;
$yamlUser->greet();

```

The complete code listing is as follows.

main.php

```

<?php

require __DIR__ . '/vendor/autoload.php';

use Symfony\Component\DependencyInjection\ContainerBuilder;
use Symfony\Component\Config\FileLocator;
use Symfony\Component\DependencyInjection\Loader\YamlFileLoader;
use Symfony\Component\DependencyInjection\Reference;

class Greeter
{
    private $greetingText;

    public function __construct($greetingText)
    {

```



```
        $this->greetingText = $greetingText;
    }

    public function greet($name)
    {
        echo $this->greetingText . ", " . $name . "\r\n";
    }
}

class User
{
    private $greeter;

    public $name;
    public $age;

    public function setGreeter(\Greeter $greeter)
    {
        $this->greeter = $greeter;
    }

    public function greet()
    {
        $this->greeter->greet($this->name);
    }
}

$container = new ContainerBuilder();
$container
    ->register('greeter', 'Greeter')
    ->addArgument('%greeter.text%');

$container
    ->register('user', 'User')
    ->addMethodCall('setGreeter', array(new Reference('greeter')));
```

```

$container->setParameter('greeter.text', 'Hi');
$greeter = $container->get('greeter');
$greeter->greet('Jon');
$user = $container->get('user');
$user->name = "Jon";
$user->age = 20;
$user->greet();

$yamlContainer = new ContainerBuilder();
$loader = new YamlFileLoader($yamlContainer, new FileLocator(__DIR__));
$loader->load('services.yml');

$yamlHello = $yamlContainer->get('greeter');
$yamlHello->greet('Jon');
$yamlUser = $yamlContainer->get('user');
$yamlUser->name = "Jon";
$yamlUser->age = 25;
$yamlUser->greet();

?>

```

services.yml

```

parameters:
    greeter.text: 'Hello'
services:
    greeter:
        class: Greeter
        arguments: ['%greeter.text%']
    user:
        class: User
        calls:
            - [setGreeter, ['@greeter']]

```

Symfony web framework uses the dependency injection component extensively. All the components are bound by the centralized service container. Symfony web framework exposes the container in all its **Controller** through **container** property. We can get all object registered in it, say logger, mailer, etc., through it.

```
$logger = $this->container->get('logger');  
$logger->info('Hi');
```

To find the object registered in the container, use the following command.

```
cd /path/to/app  
php bin/console debug:container
```

There are around 200+ objects in the **hello** web app created in the installation chapter.

6. Symfony – Events & EventListener

Symfony provides event-based programming through its **EventDispatcher** component. Any enterprise application needs event-based programming to create a highly customizable application. Events is one of the main tools for the objects to interact with each other. Without events, an object does not interact efficiently.

The process of event based programming can be summarized as - An object, called **Event source** asks the central dispatcher object to register an event, say user.registered. One or more objects, called listener asks the central dispatcher object that it wants to listen to a specific event, say user.registered. At some point of time, the Event source object asks the central dispatcher object to dispatch the event, say user.registered along with an Event object with the necessary information. The central dispatcher informs all listener objects about the event, say user.registered *and* its Event* object.

In event-based programming, we have four types of objects: Event Source, Event Listener, Event Dispatcher, and the Event itself.

Let us write a simple application to understand the concept.

Step 1: Create a project, **event-dispatcher-example**.

```
cd /path/to/dir
mkdir event-dispatcher-example
cd event-dispatcher-example
composer require symfony/event-dispatcher
```

Step 2: Create a class, **User**.

```
class User
{
    public $name;
    public $age;
}

$user = new User();
$user->name = "Jon";
$user->age = 25;
```

Step 3: Create an event, **UserRegisteredEvent**.

```
use Symfony\Component\EventDispatcher\Event;

class UserRegisteredEvent extends Event
{

```

```

    const NAME = 'user.registered';

    protected $user;

    public function __construct(User $user)
    {
        $this->user = $user;
    }

    public function getUser()
    {
        return $this->user;
    }
}

$event = new UserRegisteredEvent($user);

```

Here, **UserRegisteredEvent** has access to **User** object. The name of the event is **user.registered**.

Step 4: Create a listener, **UserListener**.

```

class UserListener
{
    public function onUserRegistrationAction(Event $event)
    {
        $user = $event->getUser();
        echo $user->name . "\r\n";
        echo $user->age . "\r\n";
    }
}

$listener = new UserListener();

```

Step 5: Create an event dispatcher object.

```

use Symfony\Component\EventDispatcher\EventDispatcher;

$dispatcher = new EventDispatcher();

```

Step 6: Connect listener and event using dispatcher object and its method, **addListener**.

```
$dispatcher
    ->addListener(
        UserRegisteredEvent::NAME,
        array($listener, 'onUserRegistrationAction'));
```

We can also add an anonymous function as event listener as shown in the following code.

```
$dispatcher
    ->addListener(
        UserRegisteredEvent::NAME,
        function(Event $event) {
            $user = $event->getUser();
            echo $user->name . "\r\n";
        });
```

Step 7: Finally, fire / dispatch the event using event dispatcher's method, **dispatch**.

```
$dispatcher->dispatch(UserRegisteredEvent::NAME, $event);
```

The complete code listing is as follows.

main.php

```
<?php

require __DIR__ . '/vendor/autoload.php';

use Symfony\Component\EventDispatcher\EventDispatcher;
use Symfony\Component\EventDispatcher\Event;

class User
{
    public $name;
    public $age;
}

class UserRegisteredEvent extends Event
{

```

```
const NAME = 'user.registered';

protected $user;

public function __construct(User $user)
{
    $this->user = $user;
}

public function getUser()
{
    return $this->user;
}
}

class UserListener
{
    public function onUserRegistrationAction(Event $event)
    {
        $user = $event->getUser();
        echo $user->name . "\r\n";
        echo $user->age . "\r\n";
    }
}

$user = new User();
$user->name = "Jon";
$user->age = 25;

$event = new UserRegisteredEvent($user);
$listener = new UserListener();

$dispatcher = new EventDispatcher();

$dispatcher
    ->addListener(
        UserRegisteredEvent::NAME,
```

```
function(Event $event) {  
    $user = $event->getUser();  
    echo $user->name . "\r\n";  
});  
  
$dispatcher  
    ->addListener(  
        UserRegisteredEvent::NAME,  
        array($listener, 'onUserRegistrationAction'));  
  
$dispatcher->dispatch(UserRegisteredEvent::NAME, $event);  
?>
```

Result

```
Jon  
Jon  
25
```

Symfony web framework has a lot of events and one can register listener for those events and program it accordingly. One of the sample event is *kernel.exception* and the corresponding event is **GetResponseForExceptionEvent**, which holds the response object (the output of a web request). This is used to catch the exception and modify the response with generic error information instead of showing runtime error to the users.

7. Symfony – Expression

As we discussed earlier, expression language is one of the salient features of Symfony application. Symfony expression is mainly created to be used in a configuration environment. It enables a non-programmer to configure the web application with little effort. Let us create a simple application to test an expression.

Step 1: Create a project, **expression-language-example**.

```
cd /path/to/dir
mkdir expression-language-example
cd expression-language-example
composer require symfony/expression-language
```

Step 2: Create an expression object.

```
use Symfony\Component\ExpressionLanguage\ExpressionLanguage;
$language = new ExpressionLanguage();
```

Step 3: Test a simple expression.

```
echo "Evaluated Value: " . $language->evaluate('10 + 12') . "\r\n" ;
echo "Compiled Code: " . $language->compile('130 % 34') . "\r\n" ;
```

Step 4: Symfony expression is powerful such that it can intercept a PHP object and its property as well in the expression language.

```
class Product
{
    public $name;
    public $price;
}

$product = new Product();
$product->name = 'Cake';
$product->price = 10;

echo "Product price is " . $language
    ->evaluate('product.price', array('product' => $product,)) . "\r\n";

echo "Is Product price higher than 5: " . $language
    ->evaluate('product.price > 5', array('product' => $product,)) . "\r\n";
```

Here, the expression **product.price** and **product.price > 5** intercept **\$product** object's property **price** and evaluate the result.

The complete coding is as follows.

main.php

```
<?php
require __DIR__ . '/vendor/autoload.php';
use Symfony\Component\ExpressionLanguage\ExpressionLanguage;
$language = new ExpressionLanguage();

echo "Evaluated Value: " . $language->evaluate('10 + 12') . "\r\n" ;
echo "Compiled Code: " . $language->compile('130 % 34') . "\r\n" ;

class Product
{
    public $name;
    public $price;
}

$product = new Product();
$product->name = 'Cake';
$product->price = 10;

echo "Product price is " . $language
    ->evaluate('product.price', array('product' => $product,)) . "\r\n";
echo "Is Product price higher than 5: " . $language
    ->evaluate('product.price > 5', array('product' => $product,)) . "\r\n";
?>
```

Result

```
Evaluated Value: 22
Compiled Code: (130 % 34)
Product price is 10
Is Product price higher than 5: 1
```

8. Symfony – Bundles

A Symfony bundle is a collection of files and folders organized in a specific structure. The bundles are modeled in such a way that it can be reused in multiple applications. The main application itself is packaged as a bundle and it is generally called **AppBundle**.

A bundle may be packaged specific to an application such as AdminBundle (admin section), BlogBundle (site's blog), etc. Such bundles cannot be shared between an application. Instead, we can model a certain part of the application such as blogs as generic bundle so that we can simply copy the bundle from one application to another application to reuse the blog functionality.

Structure of a Bundle

The basic structure of a bundle is as follows.

- **Controller** - All controller need to be placed here.
- **DependencyInjection** - All dependency injection related code and configuration need to be placed here.
- **Resources/config** - Bundle related configurations are placed here.
- **Resources/view** - Bundle related view templates are placed here.
- **Resources/public** - Bundle related stylesheets, JavaScripts, images, etc., are placed here.
- **Tests** - Bundle related unit test files are placed here.

Creating a Bundle

Let us create a simple bundle, **TutorialspointDemoBundle** in our **HelloWorld** application.

Step 1: Choose a namespace. Namespace of a bundle should include vendor name and bundle name. In our case, it is **Tutorialspoint\DemoBundle**.

Step 2: Create an empty class, **TutorialspointDemoBundle** by extending **Bundle** class and place it under **src/Tutorialspoint/DemoBundle**.

```
namespace Tutorialspoint\DemoBundle;

use Symfony\Component\HttpKernel\Bundle\Bundle;

class TutorialspointDemoBundle extends Bundle
{
}
```

Step 3: Register the class in the list of bundle supported by the application in **AppKernel** class.

```
public function registerBundles()
{
    $bundles = array(
        // ...
        // register your bundle
        new Tutorialspoint\DemoBundle\TutorialspointDemoBundle(),
    );
    return $bundles;
}
```

This is all is needed to create an empty bundle and all other concepts are the same as that of the application. Symfony also provides a console command **generate:bundle** to simplify the process of creating a new bundle, which is as follows.

```
php bin/console generate:bundle --namespace=Tutorialspoint/DemoBundle
```

Result

```
Welcome to the Symfony bundle generator!

Are you planning on sharing this bundle across multiple applications? [no]: no

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

Give your bundle a descriptive name, like BlogBundle.
Bundle name [Tutorialspoint/DemoBundle]:

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 TutorialspointDemoBundle.

Bundle name [TutorialspointDemoBundle]:
Bundles are usually generated into the src/ directory. Unless you're
doing something custom, hit enter to keep this default!
```

Target Directory [src/]:

What format do you want to use for your generated configuration?

Configuration format (annotation, yaml, xml, php) [annotation]:

Bundle generation

```
> Generating a sample bundle skeleton into app/../src/Tutorialspoint/DemoBundle
created ./app/../src/Tutorialspoint/DemoBundle/
created ./app/../src/Tutorialspoint/DemoBundle/TutorialspointDemoBundle.php
created ./app/../src/Tutorialspoint/DemoBundle/Controller/
created ./app/../src/Tutorialspoint/DemoBundle/Controller/DefaultController.php
created ./app/../tests/TutorialspointDemoBundle/Controller/
created ./app/../tests/TutorialspointDemoBundle/Controller/DefaultControllerTest.php
created ./app/../src/Tutorialspoint/DemoBundle/Resources/views/Default/
created ./app/../src/Tutorialspoint/DemoBundle/Resources/views/Default/index.html.twig
created ./app/../src/Tutorialspoint/DemoBundle/Resources/config/
created ./app/../src/Tutorialspoint/DemoBundle/Resources/config/services.yml
> Checking that the bundle is autoloading
> Enabling the bundle inside app/AppKernel.php
updated ./app/AppKernel.php
> Importing the bundle's routes from the app/config/routing.yml file
updated ./app/config/routing.yml
> Importing the bundle's services.yml from the app/config/config.yml file
updated ./app/config/config.yml
```

Everything is OK! Now get to work :).

9. Symfony – Creating a Simple Web Application

This chapter explains how to create a simple application in Symfony framework. As discussed earlier, you know how to create a new project in Symfony.

We can take an example of “student” details. Let’s start by creating a project named “student” using the following command.

```
symfony new student
```

After executing the command, an empty project is created.

Controller

Symfony is based on the Model-View-Controller (MVC) development pattern. MVC is a software approach that separates application logic from presentation. Controller plays an important role in the Symfony Framework. All the webpages in an application need to be handled by a controller.

DefaultController class is located at “**src/AppBundle/Controller**”. You can create your own Controller class there.

Move to the location “**src/AppBundle/Controller**” and create a new **StudentController** class.

Following is the basic syntax for **StudentController** class.

StudentController.php

```
namespace AppBundle\Controller;
use Symfony\Component\HttpFoundation\Response;

class StudentController
{
}
```

Now, you have created a StudentController. In the next chapter, we will discuss more about the Controller in detail.

Create a Route

Once the Controller has been created, we need to route for a specific page. Routing maps request URI to a specific controller's method.

Following is the basic syntax for routing.

```
namespace AppBundle\Controller;

use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;

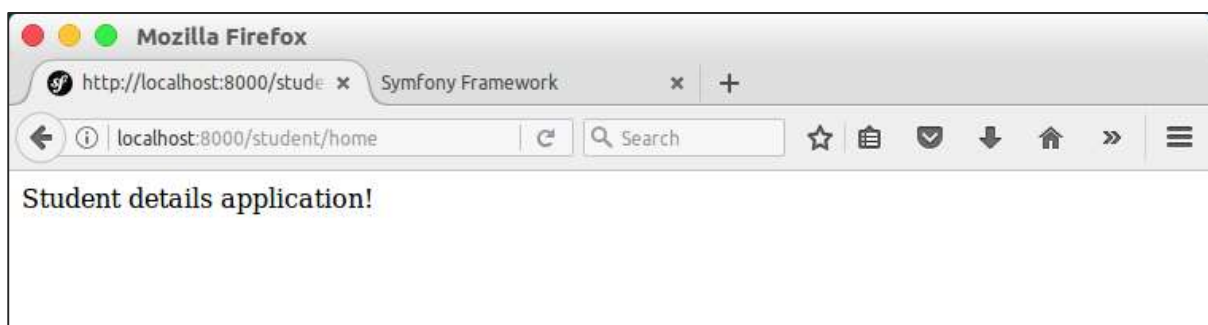
class StudentController
{
    /**
     * @Route("/student/home")
     */
    public function homeAction()
    {
        return new Response('Student details application!');
    }
}
```

In the above syntax, **@Route("/student/home")** is the route. It defines the URL pattern for the page.

homeAction() is the action method, where you can build the page and return a Response object.

We will cover routing in detail in the upcoming chapter. Now, request the url "http://localhost:8000/student/home" and it produces the following result.

Result



10. Symfony – Controllers

Controller is responsible for handling each request that comes into Symfony application. Controller reads an information from the request. Then, creates and returns a response object to the client.

According to Symfony, **DefaultController** class is located at "**src/AppBundle/Controller**". It is defined as follows.

DefaultController.php

```
<?php
namespace AppBundle\Controller;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Response;

class DefaultController extends Controller
{

}
```

Here, the **HttpFoundation** component defines an object-oriented layer for the HTTP specification, and the **FrameworkBundle** contains most of the "base" framework functionality.

Request Object

The Request class is an object-oriented representation of the HTTP request message.

Creating a Request Object

Request can be created using **createFromGlobals()** method.

```
use Symfony\Component\HttpFoundation\Request;
$request = Request::createFromGlobals();
```

You can simulate a request using Globals. Instead of creating a request based on the PHP globals, you can also simulate a request.

```
$request = Request::create(
    '/student',
    'GET',
    array('name' => 'student1')
);
```


Here, the **create()** method creates a request based on a URI, a method, and some parameters.

Overriding a Request Object

You can override the PHP global variables using the **overrideGlobals()** method. It is defined as follows.

```
$request->overrideGlobals();
```

Accessing a Request Object

Request of a web page can be accessed in a controller (action method) using **getRequest()** method of the base controller.

```
$request = $this->getRequest();
```

Identifying a Request Object

If you want to identify a request in your application, "**PathInfo**" method will return the unique identity of the request url. It is defined as follows.

```
$request->getPathInfo();
```

Response Object

The only requirement for a controller is to return a Response object. A Response object holds all the information from a given request and sends it back to the client.

Following is a simple example.

Example

```
use Symfony\Component\HttpFoundation\Response;  
$response = new Response('Default'.$name, 10);
```

You can define the Response object in JSON as follows.

```
$response = new Response(json_encode(array('name' => $name)));  
$response->headers->set('Content-Type', 'application/json');
```

Response Constructor

The constructor contains three arguments:

- The response content
- The status code
- An array of HTTP headers

Following is the basic syntax.

```
use Symfony\Component\HttpFoundation\Response;

$response = new Response(
    'Content',
    Response::HTTP_OK,
    array('content-type' => 'text/html')
);
```

For example, you can pass the content argument as,

```
$response->setContent('Student details');
```

Similarly, you can pass other arguments as well.

Sending Response

You can send a response to the client using the **send()** method. It is defined as follows.

```
$response->send();
```

To redirect the client to another URL, you can use the **RedirectResponse** class.

It is defined as follows.

```
use Symfony\Component\HttpFoundation\RedirectResponse;

$response = new RedirectResponse('http://tutorialspoint.com/');
```

FrontController

A single PHP file that handles every request coming into your application. FrontController executes the routing of different URLs to internally different parts of the application.

Following is the basic syntax for FrontController.

```
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;

$request = Request::createFromGlobals();

$path = $request->getPathInfo(); // the URI path being requested
```

```
if (in_array($path, array('', '/')))  
{  
    $response = new Response('Student home page.');
```

```
} elseif ('/about' === $path) {  
    $response = new Response('Student details page');
```

```
} else {  
    $response = new Response('Page not found.', Response::HTTP_NOT_FOUND);  
}  
$response->send();
```

Here, the **in_array()** function searches an array for a specific value.

11. Symfony – Routing

Routing maps request URI to a specific controller's method. In general, any URI has the following three parts -

- Hostname segment
- Path segment
- Query segment

For example, in URI / URL, **http://www.tutorialspoint.com/index?q=data**, **www.tutorialspoint.com** is the host name segment, **index** is the path segment and **q=data** is the query segment. Generally, routing checks the page segment against a set of constraints. If any constraint matches, then it returns a set of values. One of the main value is the controller.

Annotations

Annotation plays an important role in the configuration of Symfony application. Annotation simplifies the configuration by declaring the configuration in the coding itself. Annotation is nothing but providing meta information about class, methods, and properties. Routing uses annotation extensively. Even though routing can be done without annotation, annotation simplifies routing to a large extent.

Following is a sample annotation.

```
/**
 * @Route("/student/home")
 */
public function homeAction()
{
    // ...
}
```

Routing Concepts

Consider the *StudentController* class created in "student" project.

StudentController.php

```
// src/AppBundle/Controller/StudentController.php
namespace AppBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;
```

```

use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
class StudentController extends Controller
{
    /**
     * @Route("/student/home")
     */
    public function homeAction()
    {
        // ...
    }

    /**
     * @Route("/student/about")
     */
    public function aboutAction()
    {

    }
}

```

Here, the routing performs two steps. If you go to **/student/home**, the first route is matched then **homeAction()** is executed. Otherwise, If you go to **/student/about**, the second route is matched and then **aboutAction()** is executed.

Adding Wildcard Formats

Consider, you have a paginated list of student records with URLs like */student/2* and */student/3* for page 2 and 3 correspondingly. Then, if you want to change the route's path, you can use wildcard formats.

Example

```

// src/AppBundle/Controller/BlogController.php
namespace AppBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;

class StudentController extends Controller
{

```

```

/**
 * @Route("/student/{page}", name="student_about", requirements={"page": "\d+"})
 */
public function aboutAction($page)
{
    // ...
}
}

```

Here, the `\d+` is a regular expression that matches a digit of any length.

Assign Placeholder

You can assign a placeholder value in routing. It is defined as follows.

```

// src/AppBundle/Controller/BlogController.php
namespace AppBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;

class StudentController extends Controller
{
    /**
     * @Route("/student/{page}", name="student_about", requirements={"page": "\d+"})
     */
    public function aboutAction($page = 1)
    {
        // ...
    }
}

```

Here, if you go to `/student`, the **student_about route** will match and **\$page** will default to a value of 1.

Redirecting to a Page

If you want to redirect the user to another page, use the **redirectToRoute()** and **redirect()** methods.

```
public function homeAction()
{
    // redirect to the "homepage" route
    return $this->redirectToRoute('homepage');

    // redirect externally
    return $this->redirect('http://example.com/doc');
}
```

Generating URLs

To generate a URL, consider a route name, **student_name** and wildcard name, **student-names** used in the path for that route. The complete listing for generating a URL is defined as follows.

```
class StudentController extends Controller
{
    public function aboutAction($name)
    {
        // ...

        // /student/student-names
        $url = $this->generateUrl(
            'student_name',
            array('name' => 'student-names')
        );
    }
}
```

StudentController

Consider a simple example for routing in StudentController class as follows.

StudentController.php

```
<?php

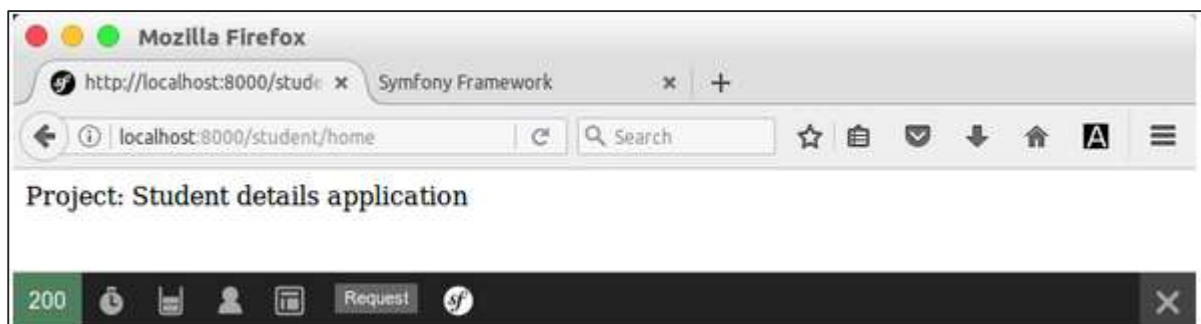
namespace AppBundle\Controller;

use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class StudentController
{
    /**
     * @Route("/student/home")
     */
    public function homeAction()
    {
        $name = 'Student details application';
        return new Response(
            '<html><body>Project: '.$name.'</body></html>'
        );
    }
}
```

Now, request the url, "http://localhost:8000/student/home" and it produces the following result.

Result



Similarly, you can create another route for **aboutAction()** as well.

12. Symfony – View Engine

A View Layer is the presentation layer of the MVC application. It separates the application logic from the presentation logic.

When a controller needs to generate HTML, CSS, or any other content then, it forwards the task to the templating engine.

Templates

Templates are basically text files used to generate any text-based documents such as HTML, XML, etc. It is used to save time and reduce errors.

By default, templates can reside in two different locations:

app/Resources/views/ - The application's views directory can contain your application's layouts and templates of the application bundle. It also overrides third party bundle templates.

vendor/path/to/Bundle/Resources/views/ - Each third party bundle contains its templates in its "Resources/views/" directory.

Twig Engine

Symfony uses a powerful templating language called **Twig**. Twig allows you to write concise and readable templates in a very easy manner. Twig templates are simple and won't process PHP tags. Twig performs whitespace control, sandboxing, and automatic HTML escaping.

Syntax

Twig contains three types of special syntax:

- **{{ ... }}** - Prints a variable or the result of an expression to the template.
- **{% ... %}** - A tag that controls the logic of the template. It is mainly used to execute a function.
- **{# ... #}** - Comment syntax. It is used to add a single or multi-line comments.

The twig base template is located at "**app/Resources/views/base.html.twig**".

Example

Let's go through a simple example using twig engine.

StudentController.php

```
<?php

namespace AppBundle\Controller;

use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class StudentController extends Controller
{
    /**
     * @Route("/student/home")
     */
    public function homeAction()
    {
        return $this->render('student/home.html.twig');
    }
}
```

Here, the **render()** method renders a template and puts that content into a Response object.

Now move to the "views" directory and create a folder "student" and inside that folder create a file "home.html.twig". Add the following changes in the file.

home.html.twig

```
//app/Resources/views/student/home.html.twig

<h3>Student application!</h3>
```

You can obtain the result by requesting the url "http://localhost:8000/student/home".

By default, Twig comes with a long list of tags, filters, and functions. Let's go through one by one in detail.

Tags

Twig supports the following important tags:

Do

The **do** tag performs similar functions as regular expression with the exception that it doesn't print anything. Its syntax is as follows:

```
{% do 5 + 6 %}
```

Include

The include statement includes a template and returns the rendered content of that file into the current namespace. Its syntax is as follows:

```
{% include 'template.html' %}
```

Extends

The extends tag can be used to extend a template from another one. Its syntax is as follows:

```
{% extends "template.html" %}
```

Block

Block acts as a placeholder and replaces the contents. Block names consists of alphanumeric characters and underscores. For example,

```
<title>{% block title %}{% endblock %}</title>
```

Embed

The **embed** tag performs a combination of both include and extends. It allows you to include another template's contents. It also allows you to override any block defined inside the included template, such as when extending a template. Its syntax is as follows:

```
{% embed "new_template.twig" %}
    {# These blocks are defined in "new_template.twig" #}
    {% block center %}
        Block content
    {% endblock %}
{% endembed %}
```

Filter

Filter sections allow you to apply regular Twig filters on a block of template data. For example,

```
{% filter upper %}
    symfony framework
{% endfilter %}
```

Here, the text will be changed to upper case.

For

For loop fetches each item in a sequence. For example,

```
{% for x in 0..10 %}
    {{ x }}
{% endfor %}
```

If

The **if** statement in Twig is similar to PHP. The expression evaluates to true or false. For example,

```
{% if value == true %}
    <p>Simple If statement</p>
{% endif %}
```

Filters

Twig contains filters. It is used to modify content before being rendered. Following are some of the notable filters.

Length

The length filter returns the length of a string. Its syntax is as follows:

```
{% if name|length > 5 %}
    ...
{% endif %}
```

Lower

The lower filter converts a value to lowercase. For example,

```
{{ 'SYMFONY'|lower }}
```

It would produce the following result:

```
symfony
```

Similarly, you can try for upper case.

Replace

The replace filter formats a given string by replacing the placeholders. For example,

```
{{ "tutorials point site %si% and %te%."|replace({'%si%': web, '%te%': "site"}) }}
```

It will produce the following result:

```
tutorials point website
```

Title

The title filter returns a titlecase version of the value. For example,

```
{{ 'symfony framework '|title }}
```

It will produce the following result:

```
Symfony Framework
```

Sort

The sort filter sorts an array. Its syntax is as follows:

```
{% for user in names|sort %}
    ...
{% endfor %}
```

Trim

The trim filter trims whitespace (or other characters) from the beginning and the end of a string. For example,

```
{{ '  Symfony!  '|trim }}
```

It will produce the following result:

```
Symfony!
```

Functions

Twig supports functions. It is used to obtain a particular result. Following are some of the important Twig functions.

Attribute

The **attribute** function can be used to access a “dynamic” attribute of a variable. Its syntax is as follows:

```
{{ attribute(object, method) }}
{{ attribute(object, method, arguments) }}
{{ attribute(array, item) }}
```

For example,

```
{{ attribute(object, method) is defined ? 'Method exists' : 'Method does not exist' }}
```

Constant

Constant function returns the constant value for a specified string. For example,

```
{{ constant('Namespace\\Classname::CONSTANT_NAME') }}
```

Cycle

The cycle function cycles on an array of values. For example,

```
{% set months = ['Jan', 'Feb', 'Mar'] %}

{% for x in 0..12 %}
    {{ cycle(months, x) }}
{% endfor %}
```

Date

Converts an argument to a date to allow date comparison. For example,

```
<p>Choose your location before {{ 'next Monday'|date('M j, Y') }}</p>
```

It will produce the following result:

```
Choose your location before May 15, 2017
```

The argument must be in one of PHP’s supported date and time formats.

You can pass a timezone as the second argument.

Dump

The dump function dumps information about a template variable. For example,

```
{{ dump(user) }}
```

Max

The max function returns the largest value of a sequence. For example,

```
{{ max(1, 5, 9, 11, 15) }}
```

Min

The min function returns the smallest value of a sequence. For example,

```
{{ min(1, 3, 2) }}
```

Include

The include function returns the rendered content of a template. For example,

```
{{ include('template.html') }}
```

Random

The random function generates a random value. For example,

```
{{ random(['Jan', 'Feb', 'Mar', 'Apr']) }}  
{# example output: Jan #}
```

Range

Range function returns a list containing an arithmetic progression of integers. For example,

```
{% for x in range(1, 5) %}  
    {{ x }},  
{% endfor %}
```

It will produce the following result:

```
1,2,3,4,5
```

Layouts

A Layout represents the common parts of multiple views, i.e. for example, page header, and footer.

Template Inheritance

A template can be used by another one. We can achieve this using template inheritance concept. Template inheritance allows you to build a base "layout" template that contains all the common elements of web site defined as blocks.

Let's take a simple example to understand more about template inheritance.

Example

Consider the base template located at `"app/Resources/views/base.html.twig"`. Add the following changes in the file.

base.html.twig

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>{% block title %}Parent template Layout{% endblock %}</title>
    </head>
</html>
```

Now move to the index template file located at `"app/Resources/views/default/index.html.twig"`. Add the following changes in it.

index.html.twig

```
{% extends 'base.html.twig' %}

{% block title %}Child template Layout{% endblock %}
```

Here, the **{% extends %}** tag informs the templating engine to first evaluate the base template, which sets up the layout and defines the block. The child template is then rendered. A child template can extend the base layout and override the title block. Now, request the url `"http://localhost:8000"` and you can obtain its result.

Assets

The Asset manages URL generation and versioning of web assets such as CSS stylesheets, JavaScript files, and image files.

JavaScript

To include JavaScript files, use the **javascripts** tag in any template.

```
{# Include javascript #}
{% block javascripts %}
    {% javascripts '@AppBundle/Resources/public/js/*' %}
        <script src="{{ asset_url }}"></script>
    {% endjavascripts %}
{% endblock %}
```

Stylesheets

To include stylesheet files, use the **stylesheets** tag in any template

```
{# include style sheet #}
{% block stylesheets %}
    {% stylesheets 'bundles/app/css/*' filter='cssrewrite' %}
        <link rel="stylesheet" href="{{ asset_url }}" />
    {% endstylesheets %}
{% endblock %}
```

Images

To include an image, you can use the image tag. It is defined as follows.

```
{% image '@AppBundle/Resources/public/images/example.jpg' %}
    
{% endimage %}
```

Compound Assets

You can combine many files into one. This helps to reduce the number of HTTP requests, and produces greater front-end performance.

```
{% javascripts
    '@AppBundle/Resources/public/js/*'
    '@AcmeBarBundle/Resources/public/js/form.js'
    '@AcmeBarBundle/Resources/public/js/calendar.js' %}
    <script src="{{ asset_url }}"></script>
{% endjavascripts %}
```

13. Symfony – Doctrine ORM

In Symfony web framework, model plays an important role. They are the business entities. They are either provided by customers or fetched from back-end database, manipulated according to business rules and persisted back into the database. They are the data presented by Views. Let us learn about models and how they interact with back-end system in this chapter.

Database Model

We need to map our models to the back-end relational database items to safely and efficiently fetch and persist the models. This mapping can be done with an Object Relational Mapping (ORM) tool. Symfony provides a separate bundle, **DoctrineBundle**, which integrates Symfony with third party PHP database ORM tool, **Doctrine**.

Doctrine ORM

By default, Symfony framework doesn't provide any component to work with databases. But, it integrates tightly with **Doctrine ORM**. Doctrine contains several PHP libraries used for database storage and object mapping.

Following example will help you understand how Doctrine works, how to configure a database and how to save and retrieve the data.

Doctrine ORM Example

In this example, we will first configure the database and create a Student object, then perform some operations in it.

To do this we need to adhere to the following steps.

Step 1: Create a Symfony Application

Create a Symfony application, **dbsample** using the following command.

```
symfony new dbsample
```

Step 2: Configure a Database

Generally, the database information is configured in "app/config/parameters.yml" file.

Open the file and add the following changes.

parameter.yml

```
parameters:
    database_host: 127.0.0.1
    database_port: null
```

```
database_name: studentsdb
database_user: <user_name>
database_password: <password>
mailer_transport: smtp
mailer_host: 127.0.0.1
mailer_user: null
mailer_password: null
secret: 037ab82c601c10402408b2b190d5530d602b5809
doctrine:
  dbal:
    driver:   pdo_mysql
    host:     '%database_host%'
    dbname:   '%database_name%'
    user:     '%database_user%'
    password: '%database_password%'
    charset:  utf8mb4
```

Now, Doctrine ORM can connect to the database.

Step 3: Create a Database

Issue the following command to generate “studentsdb” database. This step is used to bind the database in Doctrine ORM.

```
php bin/console doctrine:database:create
```

After executing the command, it automatically generates an empty “studentsdb” database. You can see the following response on your screen.

```
Created database `studentsdb` for connection named default
```

Step 4: Map Information

Mapping information is nothing but “metadata”. It is a collection of rules that informs Doctrine ORM exactly how the Student class and its properties are mapped to a specific database table.

Well, this metadata can be specified in a number of different formats, including YAML, XML or you can directly pass Student class using annotations. It is defined as follows.

Student.php

Add the following changes in the file.

```
<?php

namespace AppBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity
 * @ORM\Table(name="students")
 */
class Student
{
    /**
     * @ORM\Column(type="integer")
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="AUTO")
     */
    private $id;

    /**
     * @ORM\Column(type="string", length=50)
     */
    private $name;

    /**
     * @ORM\Column(type="text")
     */
    private $address;
}
```

Here, the table name is optional. If the table name is not specified, then it will be determined automatically based on the name of the entity class.

Step 5: Bind an Entity

Doctrine creates simple entity classes for you. It helps you build any entity.

Issue the following command to generate an entity.

```
php bin/console doctrine:generate:entities AppBundle/Entity/Student
```

Then you will see the following result and the entity will be updated.

```
Generating entity "AppBundle\Entity\Student"
> backing up Student.php to Student.php~
> generating AppBundle\Entity\Student
```

Student.php

```
<?php
namespace AppBundle\Entity;
use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity
 * @ORM\Table(name="students")
 */
class Student
{
    /**
     * @ORM\Column(type="integer")
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="AUTO")
     */
    private $id;

    /**
     * @ORM\Column(type="string", length=50)
     */
    private $name;

    /**
     * @ORM\Column(type="text")
     */
}
```

```
private $address;

/**
 * Get id
 *
 * @return integer
 */
public function getId()
{
    return $this->id;
}

/**
 * Set name
 *
 * @param string $name
 *
 * @return Student
 */
public function setName($name)
{
    $this->name = $name;

    return $this;
}

/**
 * Get name
 *
 * @return string
 */
public function getName()
{
    return $this->name;
}

/**
```

```
* Set address
*
* @param string $address
*
* @return Student
*/
public function setAddress($address)
{
    $this->address = $address;

    return $this;
}

/**
 * Get address
 *
 * @return string
 */
public function getAddress()
{
    return $this->address;
}
}
```

Step 6: Map Validation

After creating entities, you should validate the mappings using the following command.

```
php bin/console doctrine:schema:validate
```

It will produce the following result:

```
[Mapping] OK - The mapping files are correct.
[Database] FAIL - The database schema is not in sync with the current mapping
file.
```

Since we have not created the students table, the entity is out of sync. Let us create the students table using the Symfony command in the next step.

Step 7: Create a Schema

Doctrine can automatically create all the database tables needed for *Student* entity. This can be done using the following command.

```
php bin/console doctrine:schema:update --force
```

After executing the command, you can see the following response.

```
Updating database schema...  
Database schema updated successfully! "1" query was executed
```

This command compares what your database should look like with how it actually looks, and executes the SQL statements needed to update the database schema to where it should be.

Now, again validate the schema using the following command.

```
php bin/console doctrine:schema:validate
```

It will produce the following result:

```
[Mapping] OK - The mapping files are correct.  
[Database] OK - The database schema is in sync with the mapping files.
```

Step 8: Getter and setter

As seen in the Bind an Entity section, the following command generates all the getters and setters for the Student class.

```
$ php bin/console doctrine:generate:entities AppBundle\Entity/Student
```

Step 9: Persist Objects to the Database

Now, we have mapped the Student entity to its corresponding Student table. We should now be able to persist Student objects to the database. Add the following method to the StudentController of the bundle.

StudentController.php

```
<?php  
  
namespace AppBundle\Controller;  
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;  
use Symfony\Bundle\FrameworkBundle\Controller\Controller;  
use Symfony\Component\HttpFoundation\Response;  
  
use AppBundle\Entity\Student;  
class StudentController extends Controller
```



```

{
    /**
     * @Route("/student/add")
     */
    public function addAction()
    {
        $stud = new Student();
        $stud->setName('Adam');
        $stud->setAddress('12 north street');

        $doct = $this->getDoctrine()->getManager();

        // tells Doctrine you want to save the Product
        $doct->persist($stud);

        //executes the queries (i.e. the INSERT query)
        $doct->flush();
        return new Response('Saved new student with id ' . $stud->getId());
    }
}

```

Here, we accessed the doctrine manager using `getManager()` method through `getDoctrine()` of base controller and then persist the current object using **`persist()`** method of doctrine manager. `persist()` method adds the command to the queue, but the **`flush()`** method does the actual work (persisting the student object).

Step 10: Fetch Objects from the Database

Create a function in StudentController that will display the student details.

StudentController.php

```

/**
 * @Route("/student/display")
 */
public function displayAction()
{
    $stud = $this->getDoctrine()
        ->getRepository('AppBundle:Student')
        ->findAll();
}

```

```

        return $this->render('student/display.html.twig', array('data' => $stud));
    }

```

Step 11: Create a View

Let's create a view that points to display action. Move to the views directory and create a file "display.html.twig". Add the following changes in the file.

display.html.twig

```

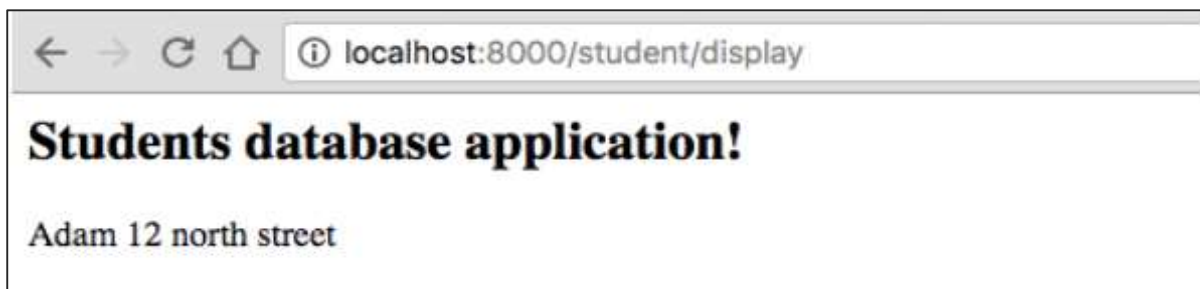
<style>
.table { border-collapse: collapse; }
.table th, td {
    border-bottom: 1px solid #ddd;
    width: 250px;
    text-align: left;
    align: left;
}
</style>
<h2>Students database application!</h2>

<table class="table">
    <tr>
        <th>Name</th>
        <th>Address</th>
    </tr>
    {% for x in data %}
        <tr>
            <td>{{ x.Name }}</td>
            <td>{{ x.Address }}</td>
        </tr>
    {% endfor %}
</table>

```

You can obtain the result by requesting the URL "http://localhost:8000/student/display" in a browser.

It wil produce the following output on screen:



Step 12: Update an Object

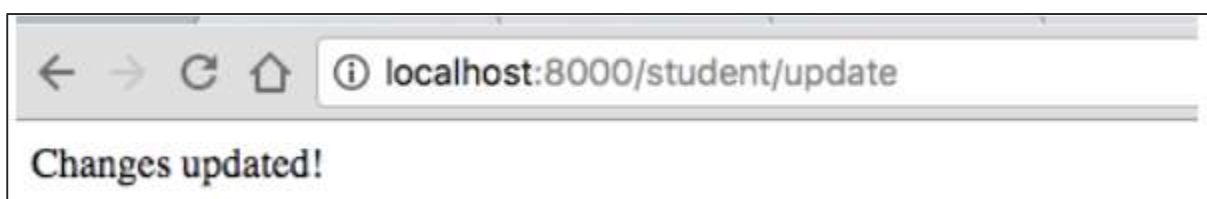
To update an object in StudentController, create an action and add the following changes.

```
/**
 * @Route("/student/update/{id}")
 */
public function updateAction($id)
{
    $doct = $this->getDoctrine()->getManager();
    $stud = $doct->getRepository('AppBundle:Student')->find($id);

    if (!$stud) {
        throw $this->createNotFoundException(
            'No student found for id '.$id
        );
    }
    $stud->setAddress('7 south street');
    $doct->flush();
    return new Response('Changes updated!');
}
```

Now, request the URL "http://localhost:8000/Student/update/1" and it will produce the following result.

It will produce the following output on screen:



Step 13: Delete an Object

Deleting an object is similar and it requires a call to the `remove()` method of the entity (doctrine) manager.

This can be done using the following command.

```
/**
 * @Route("/student/delete/{id}")
 */
public function deleteAction($id)
{
    $doct = $this->getDoctrine()->getManager();
    $stud = $doct->getRepository('AppBundle:Student')->find($id);

    if (!$stud) {
        throw $this->createNotFoundException(
            'No student found for id '.$id
        );
    }

    $doct->remove($stud);
    $doct->flush();

    return new Response('Record deleted!');
}
```

14. Symfony – Forms

Symfony provides various in-built tags to handle HTML forms easily and securely. Symfony's Form component performs form creation and validation process. It connects the model and the view layer. It provides a set of form elements to create a full-fledged html form from pre-defined models. This chapter explains about Forms in detail.

Form Fields

Symfony framework API supports large group of field types. Let's go through each of the field types in detail.

FormType

It is used to generate a form in Symfony framework. Its syntax is as follows:

```
use Symfony\Component\Form\Extension\Core\Type\TextType;
use Symfony\Component\Form\Extension\Core\Type\EmailType;
use Symfony\Component\Form\Extension\Core\Type\FormType;
// ...

$builder = $this->createFormBuilder($studentinfo);
$builder
    ->add('title', TextType::class);
```

Here, **\$studentinfo** is an entity of type Student. **createFormBuilder** is used to create a HTML form. **add** method is used to add input elements inside the form. **title** refers to student title property. **TextType::class** refers to html text field. Symfony provides classes for all html elements.

TextType

The TextType field represents the most basic input text field. Its syntax is as follows:

```
use Symfony\Component\Form\Extension\Core\Type\TextType;
$builder->add('name', TextType::class);
```

Here, the name is mapped with an entity.

TextareaType

Renders a textarea HTML element. Its syntax is as follows:

```
use Symfony\Component\Form\Extension\Core\Type\TextareaType;
$builder->add('body', TextareaType::class, array(
    'attr' => array('class' => 'tinymce'),
));
```

EmailType

The EmailType field is a text field that is rendered using the HTML5 email tag. Its syntax is as follows:

```
use Symfony\Component\Form\Extension\Core\Type\EmailType;
$builder->add('token', EmailType::class, array(
    'data' => 'abcdef',
));
```

PasswordType

The PasswordType field renders an input password text box. Its syntax is as follows:

```
use Symfony\Component\Form\Extension\Core\Type>PasswordType;
$builder->add('password', PasswordType::class);
```

RangeType

The RangeType field is a slider that is rendered using the HTML5 range tag. Its syntax is as follows:

```
use Symfony\Component\Form\Extension\Core\Type\RangeType;
// ...

$builder->add('name', RangeType::class, array(
    'attr' => array(
        'min' => 100,
        'max' => 200
    )
));
```

PercentType

The PercentType renders an input text field and specializes in handling percentage data. Its syntax is as follows:

```
use Symfony\Component\Form\Extension\Core\Type\PercentType;
// ...
$builder->add('token', PercentType::class, array(
    'data' => 'abcdef',
));
```

DateType

Renders a date format. Its syntax is as follows:

```
use Symfony\Component\Form\Extension\Core\Type\DateType;
// ...
$builder->add('joined', DateType::class, array(
    'widget' => 'choice',
));
```

Here, Widget is the basic way to render a field.

It performs the following function.

- **choice:** Renders three select inputs. The order of the selects is defined in the format option.
- **text:** Renders a three field input of type text (month, day, year).
- **single_text:** Renders a single input of type date. The user's input is validated based on the format option.

CheckboxType

Creates a single input checkbox. This should always be used for a field that has a boolean value. Its syntax is as follows:

```
use Symfony\Component\Form\Extension\Core\Type\CheckboxType;
// ...

$builder->add('sports', CheckboxType::class, array(
    'label'      => 'Are you interested in sports?',
    'required'   => false,
));
```

RadioType

Creates a single radio button. If the radio button is selected, the field will be set to the specified value. Its syntax is as follows:

```
use Symfony\Component\Form\Extension\Core\Type\RadioType;
// ...

$builder->add('token', RadioType::class, array(
    'data' => 'abcdef',
));
```

Note that, Radio buttons cannot be unchecked, the value only changes when another radio button with the same name gets checked.

RepeatedType

This is a special field “group”, that creates two identical fields whose values must match. Its syntax is as follows:

```
use Symfony\Component\Form\Extension\Core\Type\RepeatedType;
use Symfony\Component\Form\Extension\Core\Type\PasswordType;
// ...

$builder->add('password', RepeatedType::class, array(
    'type' => PasswordType::class,
    'invalid_message' => 'The password fields must match.',
    'options' => array('attr' => array('class' => 'password-field')),
    'required' => true,
    'first_options' => array('label' => 'Password'),
    'second_options' => array('label' => 'Repeat Password'),
));
```

This is mostly used to check the user’s password or email.

ButtonType

A simple clickable button. Its syntax is as follows:

```
use Symfony\Component\Form\Extension\Core\Type\ButtonType;
// ...

$builder->add('save', ButtonType::class, array(
```



```
'attr' => array('class' => 'save'),
));
```

ResetType

A button that resets all fields to its initial values. Its syntax is as follows:

```
use Symfony\Component\Form\Extension\Core\Type\ResetType;
// ...

$builder->add('save', ResetType::class, array(
    'attr' => array('class' => 'save'),
));
```

ChoiceType

A multi-purpose field is used to allow the user to “choose” one or more options. It can be rendered as a select tag, radio buttons, or checkboxes. Its syntax is as follows:

```
use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
// ...

$builder->add('gender', ChoiceType::class, array(
    'choices' => array(
        'Male' => true,
        'Female' => false,
    ),
));
```

SubmitType

A submit button is used to submit form-data. Its syntax is as follows:

```
use Symfony\Component\Form\Extension\Core\Type\SubmitType;
// ...

$builder->add('save', SubmitType::class, array(
    'attr' => array('class' => 'save'),
));
```

Form Helper Function

Form helper functions are twig functions used to create forms easily in templates.

form_start

Returns an HTML form tag that points to a valid action, route, or URL. Its syntax is as follows:

```
{{ form_start(form, {'attr': {'id': 'form_person_edit'}}) }}
```

form_end

Closes the HTML form tag created using *form_start*. Its syntax is as follows:

```
{{ form_end(form) }}
```

textarea

Returns a textarea tag, optionally wrapped with an inline rich-text JavaScript editor.

checkbox

Returns an XHTML compliant input tag with type="checkbox". Its syntax is as follows:

```
echo checkbox_tag('choice[]', 1);  
echo checkbox_tag('choice[]', 2);  
echo checkbox_tag('choice[]', 3);  
echo checkbox_tag('choice[]', 4);
```

input_password_tag

Returns an XHTML compliant input tag with type="password". Its syntax is as follows:

```
echo input_password_tag('password');  
echo input_password_tag('password_confirm');
```

input_tag

Returns an XHTML compliant input tag with type="text". Its syntax is as follows:

```
echo input_tag('name');
```

label

Returns a label tag with the specified parameter.

radiobutton

Returns an XHTML compliant input tag with type="radio". Its syntax is as follows:

```
echo ' Yes '.radiobutton_tag('true', 1);  
echo ' No '.radiobutton_tag('false', 0);
```

reset_tag

Returns an XHTML compliant input tag with type="reset". Its syntax is as follows:

```
echo reset_tag('Start Over');
```

select

Returns a select tag populated with all the countries in the world. Its syntax is as follows:

```
echo select_tag('url', options_for_select($url_list), array('onChange' =>  
'Javascript:this.form.submit();'));
```

submit

Returns an XHTML compliant input tag with type="submit". Its syntax is as follows:

```
echo submit_tag('Update Record');
```

In the next section, we will learn how to create a form using form fields.

Student Form Application

Let's create a simple Student details form using Symfony Form fields. To do this, we should adhere to the following steps.:

Step 1: Create a Symfony Application

Create a Symfony application, **formsample**, using the following command.

```
symfony new formsample
```

Entities are usually created under the "src/AppBundle/Entity/" directory.

Step 2: Create an Entity

Create the file "StudentForm.php" under the "src/AppBundle/Entity/" directory. Add the following changes in the file.

StudentForm.php

```
<?php
namespace AppBundle\Entity;

class StudentForm
{
    private $studentName;
    private $studentId;
    public $password;
    private $address;
    public $joined;
    public $gender;
    private $email;
    private $marks;
    public $sports;

    public function getStudentName()
    {
        return $this->studentName;
    }

    public function setStudentName($studentName)
    {
        $this->studentName = $studentName;
    }

    public function getStudentId()
    {
        return $this->studentId;
    }

    public function setStudentId($studentid)
    {
        $this->studentid = $studentid;
    }
}
```

```
public function getAddress()
{
    return $this->address;
}

public function setAddress($address)
{
    $this->address = $address;
}

public function getEmail()
{
    return $this->email;
}

public function setEmail($email)
{
    $this->email = $email;
}

public function getMarks()
{
    return $this->marks;
}

public function setMarks($marks)
{
    $this->marks = $marks;
}
}
```

Step 3: Add a StudentController

Move to the directory "src/AppBundle/Controller", create "StudentController.php" file, and add the following code in it.

StudentController.php

```
<?php

namespace AppBundle\Controller;

use AppBundle\Entity\StudentForm;
use AppBundle\Form\FormValidationType;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Request;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Form\Extension\Core\Type\TextType;
use Symfony\Component\Form\Extension\Core\Type\DateType;
use Symfony\Component\Form\Extension\Core\Type\SubmitType;
use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
use Symfony\Component\Form\Extension\Core\Type>PasswordType;
use Symfony\Component\Form\Extension\Core\Type\RangeType;
use Symfony\Component\Form\Extension\Core\Type\EmailType;
use Symfony\Component\Form\Extension\Core\Type\CheckboxType;
use Symfony\Component\Form\Extension\Core\Type\ButtonType;
use Symfony\Component\Form\Extension\Core\Type\TextareaType;
use Symfony\Component\Form\Extension\Core\Type\PercentType;
use Symfony\Component\Form\Extension\Core\Type\RepeatedType;

class StudentController extends Controller
{
    /**
     * @Route("/student/new")
     */
    public function newAction(Request $request)
    {

        $stud = new StudentForm();
        $form = $this->createFormBuilder($stud)
            ->add('studentName', TextType::class)
```

```

->add('studentId', TextType::class)
->add('password', RepeatedType::class, array(
    'type' => PasswordType::class,
    'invalid_message' => 'The password fields must match.',
    'options' => array('attr' => array('class' => 'password-field')),
    'required' => true,
    'first_options' => array('label' => 'Password'),
    'second_options' => array('label' => 'Re-enter'),
))
->add('address', TextareaType::class)
->add('joined', DateType::class, array(
    'widget' => 'choice',
))
->add('gender', ChoiceType::class, array(
    'choices' => array(
        'Male' => true,
        'Female' => false,
    ),
))
->add('email', EmailType::class)
->add('marks', PercentType::class)
->add('sports', CheckboxType::class, array(
    'label' => 'Are you interested in sports?',
    'required' => false,
))
->add('save', SubmitType::class, array('label' => 'Submit'))
->getForm();

return $this->render('student/new.html.twig', array(
    'form' => $form->createView(),
));
}
}

```

Step 4: Render the View

Move to the directory "app/Resources/views/student/", create "new.html.twig" file and add the following changes in it.

```
{% extends 'base.html.twig' %}
{% block stylesheets %}
    <style>
#simpleform
{
    width:600px;
    border:2px solid grey;
    padding:14px;
}

#simpleform label
{
    font-size:14px;
    float:left;
    width:300px;
    text-align:right;
    display:block;
}

#simpleform span
{
    font-size:11px;
    color:grey;
    width:100px;
    text-align:right;
    display:block;
}

#simpleform input
{
    border:1px solid grey;
    font-family:verdana;
    font-size:14px;
```



```
        color:light blue;
        height:24px;
        width:250px;
        margin: 0 0 10px 10px;
    }

    #simpleform textarea
    {
        border:1px solid grey;
        font-family:verdana;
        font-size:14px;
        color:light blue;
        height:120px;
        width:250px;
        margin: 0 0 20px 10px;
    }

    #simpleform select
    {
        margin: 0 0 20px 10px;
    }

    #simpleform button
    {
        clear:both;
        margin-left:250px;
        background: grey;
        color:#FFFFFF;
        border:solid 1px #666666;
        font-size:16px;
    }

    </style>
{% endblock %}

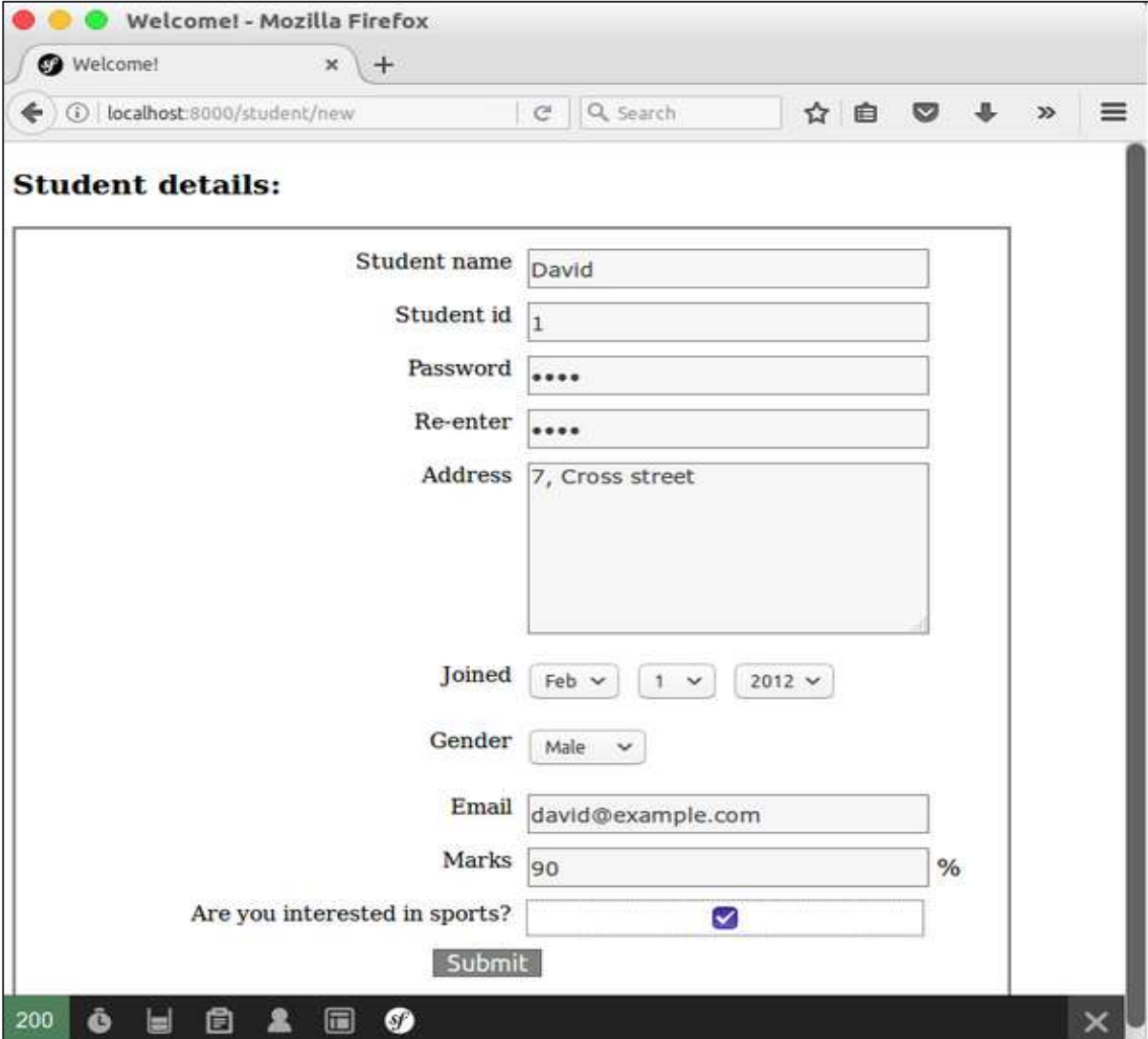
{% block body %}
    <h3>Student details:</h3>
```

```
<div id="simpleform">
    {{ form_start(form) }}
    {{ form_widget(form) }}
    {{ form_end(form) }}

</div>
{% endblock %}
```

Now request the url, "http://localhost:8000/student/new" and it produces the following result.

Result



The screenshot shows a Mozilla Firefox browser window with the title 'Welcome! - Mozilla Firefox'. The address bar displays 'localhost:8000/student/new'. The page content is titled 'Student details:' and contains a registration form with the following fields and values:

- Student name: David
- Student id: 1
- Password:
- Re-enter:
- Address: 7, Cross street
- Joined: Feb 1 2012
- Gender: Male
- Email: david@example.com
- Marks: 90 %
- Are you interested in sports? ☒
- Submit button

The browser's status bar at the bottom shows a green bar with the number '200' and several icons, including a clock, a folder, a person, and a search icon.

15. Symfony – Validation

Validation is the most important aspect while designing an application. It validates the incoming data. This chapter explains about form validation in detail.

Validation Constraints

The validator is designed to validate objects against constraints. If you validate an object, simply map one or more constraints to its class and then pass it to the validator service. By default, when validating an object all constraints of the corresponding class will be checked to see whether or not they actually pass. Symfony supports the following notable validation constraints.

NotBlank

Validates that a property is not blank. Its syntax is as follows:

```
namespace AppBundle\Entity;
use Symfony\Component\Validator\Constraints as Assert;
class Student
{
    /**
     * @Assert\NotBlank()
     */
    protected $studentName;
}
```

This NotBlank constraint ensures that studentName property should not blank.

NotNull

Validates that a value is not strictly equal to null. Its syntax is as follows:

```
namespace AppBundle\Entity;
use Symfony\Component\Validator\Constraints as Assert;
class Student
{
    /**
     * @Assert\NotNull()
     */
    protected $studentName;
}
```

Email

Validates that a value is a valid email address. Its syntax is as follows:

```
namespace AppBundle\Entity;
use Symfony\Component\Validator\Constraints as Assert;
class Student
{
    /**
     * @Assert\Email(
     *     message = "The email '{{ value }}' is not a valid email.",
     *     checkMX = true
     * )
     */
    protected $email;
}
```

IsNull

Validates that a value is exactly equal to null. Its syntax is as follows:

```
namespace AppBundle\Entity;
use Symfony\Component\Validator\Constraints as Assert;
class Student
{
    /**
     * @Assert\IsNull()
     */
    protected $studentName;
}
```

Length

Validates that a given string length is between some minimum and maximum value. Its syntax is as follows:

```
namespace AppBundle\Entity;
use Symfony\Component\Validator\Constraints as Assert;
class Student
{
    /**
```

```

    * @Assert\Length(
    *   min = 5,
    *   max = 25,
    *   minMessage = "Your first name must be at least {{ limit }} characters long",
    *   maxMessage = "Your first name cannot be longer than {{ limit }} characters"
    * )
    */
    protected $studentName;
}

```

Range

Validates that a given number is between some minimum and maximum number. Its syntax is as follows:

```

namespace AppBundle\Entity;
use Symfony\Component\Validator\Constraints as Assert;
class Student
{
    /**
     * @Assert\Range(
     *     min = 40,
     *     max = 100,
     *     minMessage = "You must be at least {{ limit }} marks",
     *     maxMessage = "Your maximum {{ limit }} marks"
     * )
     */
    protected $marks;
}

```

Date

Validates that a value is a valid date. It follows a valid YYYY-MM-DD format. Its syntax is as follows:

```

namespace AppBundle\Entity;
use Symfony\Component\Validator\Constraints as Assert;
class Student
{

```

```

    /**
     * @Assert\Date()
     */
    protected $joinedAt;
}

```

Choice

This constraint is used to ensure that the given value is one of a given set of valid choices. It can also be used to validate that each item in an array of items is one of those valid choices. Its syntax is as follows:

```

namespace AppBundle\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Student
{
    /**
     * @Assert\Choice(choices = {"male", "female"}, message = "Choose a valid gender.")
     */
    protected $gender;
}

```

UserPassword

This validates that an input value is equal to the current authenticated user's password. This is useful in a form where users can change their password, but need to enter their old password for security. Its syntax is as follows:

```

namespace AppBundle\Form\Model;

use Symfony\Component\Security\Core\Validator\Constraints as SecurityAssert;

class ChangePassword
{
    /**
     * @SecurityAssert\UserPassword(
     *     message = "Wrong value for your current password"
     * )
     */
    protected $oldPassword;
}

```

```
}
```

This constraint validates that the old password matches the user's current password.

Validation Example

Let us write a simple application example to understand the validation concept.

Step 1: Create a validation application.

Create a Symfony application, **validationssample**, using the following command.

```
symfony new validationssample
```

Step 2: Create an entity named, **FormValidation** in file "**FormValidation.php**" under the "**src/AppBundle/Entity/**" directory. Add the following changes in the file.

FormValidation.php

```
<?php
namespace AppBundle\Entity;
use Symfony\Component\Validator\Constraints as Assert;
class FormValidation
{
    /**
     * @Assert\NotBlank()
     */
    protected $name;

    /**
     * @Assert\NotBlank()
     */
    protected $id;

    protected $age;

    /**
     * @Assert\NotBlank()
     */
    protected $address;

    public $password;
```

```
/**
 * @Assert\Email(
 *     message = "The email '{{ value }}' is not a valid email.",
 *     checkMX = true
 * )
 */
protected $email;

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

public function setName($name)
{
    $this->name = $name;
}

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

public function getAge()
{
    return $this->age;
}

public function setAge($age)
{
    $this->age = $age;
}
```



```

    }

    public function getAddress()
    {
        return $this->address;
    }

    public function setAddress($address)
    {
        $this->address = $address;
    }

    public function getEmail()
    {
        return $this->email;
    }

    public function setEmail($email)
    {
        $this->email = $email;
    }
}

```

Step 3: Create a **validateAction** method in StudentController. Move to the directory "**src/AppBundle/Controller**", create "**studentController.php**" file, and add the following code in it.

StudentController.php

```

use AppBundle\Entity\FormValidation;
/**
 * @Route("/student/validate")
 */
public function validateAction(Request $request)
{
    $validate = new FormValidation();
    $form = $this->createFormBuilder($validate)
        ->add('name', TextType::class)

```

```

        ->add('id', TextType::class)
        ->add('age', TextType::class)
        ->add('address', TextType::class)
        ->add('email', TextType::class)
        ->add('save', SubmitType::class, array('label' => 'Submit'))
        ->getForm();

    $form->handleRequest($request);

    if ($form->isSubmitted() && $form->isValid()) {
        $validate = $form->getData();
        return new Response('Form is validated.');
    }

    return $this->render('student/validate.html.twig', array(
        'form' => $form->createView(),
    ));
}

```

Here, we have created the form using Form classes and then handled the form. If the form is submitted and is valid, a form validated message is shown. Otherwise, the default form is shown.

Step 4: Create a view for the above created action in StudentController. Move to the directory "**app/Resources/views/student/**". Create "**validate.html.twig**" file and add the following code in it.

```

{% extends 'base.html.twig' %}
{% block stylesheets %}
    <style>
#simpleform
{
    width:600px;
    border:2px solid grey;
    padding:14px;
}

#simpleform label
{

```

```
        font-size:14px;
        float:left;
        width:300px;
        text-align:right;
        display:block;
    }

    #simpleform span
    {
        font-size:11px;
        color:grey;
        width:100px;
        text-align:right;
        display:block;
    }

    #simpleform input
    {
        border:1px solid grey;
        font-family:verdana;
        font-size:14px;
        color:light blue;
        height:24px;
        width:250px;
        margin: 0 0 10px 10px;
    }

    #simpleform textarea
    {
        border:1px solid grey;
        font-family:verdana;
        font-size:14px;
        color:light blue;
        height:120px;
        width:250px;
        margin: 0 0 20px 10px;
```

```

    }

    #simpleform select
    {
        margin: 0 0 20px 10px;
    }

    #simpleform button
    {
        clear:both;
        margin-left:250px;
        background: grey;
        color:#FFFFFF;
        border:solid 1px #666666;
        font-size:16px;
    }

    </style>
{% endblock %}


{% block body %}
    <h3>Student form validation:</h3>
    <div id="simpleform">
        {{ form_start(form) }}
        {{ form_widget(form) }}
        {{ form_end(form) }}
    </div>
{% endblock %}

```

Here, we have used form tags to create the form.

Step 5: Finally, run the application, **<http://localhost:8000/student/validate>**.

Result: Initial Page



A screenshot of a Mozilla Firefox browser window. The title bar says "Welcome! - Mozilla Firefox". The address bar shows "localhost:8000/student/validat". The page content is titled "Student form validation:". Below the title is a form with the following fields: Name (Peter), Id (1), Age (20), Address (test), and Email (test@test.com). A "Submit" button is at the bottom of the form. The browser's status bar at the bottom shows a green bar with the number "200" and a Symfony logo.

Welcome! - Mozilla Firefox

Welcome!

localhost:8000/student/validat

Search

Student form validation:

Name Peter

Id 1

Age 20

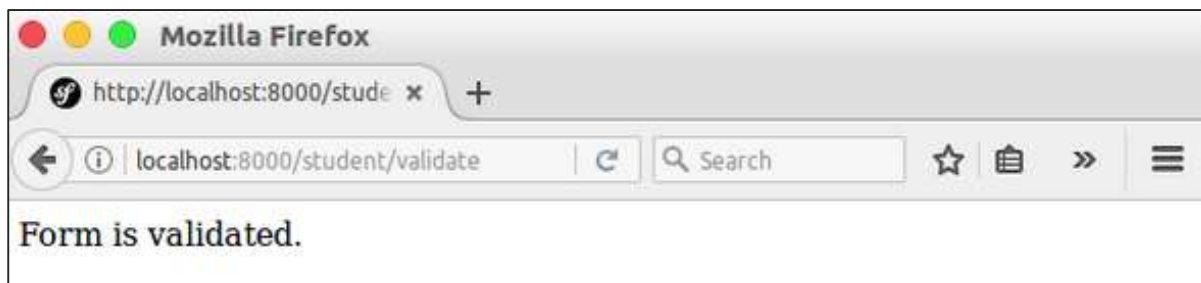
Address test

Email test@test.com

Submit

200

Result: Final Page



A screenshot of a Mozilla Firefox browser window. The title bar says "Mozilla Firefox". The address bar shows "http://localhost:8000/stude". The page content displays the message "Form is validated.".

Mozilla Firefox

http://localhost:8000/stude

localhost:8000/student/validate

Search

Form is validated.

16. Symfony – File Uploading

Symfony Form component provides **FileType** class to handle file input element. It enables easy uploading of images, documents, etc. Let us learn how to create a simple application using FileType feature.

Step 1: Create a new application, **fileuploadsample** using the following command.

```
symfony new fileuploadsample
```

Step 2: Create an entity, **Student**, having name, age and photo as shown in the following code.

src/AppBundle/Entity/Student.php

```
<?php
namespace AppBundle\Entity;
use Symfony\Component\Validator\Constraints as Assert;

class Student
{
    /**
     * @Assert\NotBlank()
     */
    private $name;

    /**
     * @Assert\NotBlank()
     */
    private $age;

    /**
     * @Assert\NotBlank(message="Please, upload the photo.")
     * @Assert\File(mimeTypes={ "image/png", "image/jpeg" })
     */
    private $photo;
    public function getName()
    {
        return $this->name;
    }
}
```

```

    public function setName($name)
    {
        $this->name = $name;
        return $this;
    }
    public function getAge()
    {
        return $this->age;
    }
    public function setAge($age)
    {
        $this->age = $age;
        return $this;
    }
    public function getPhoto()
    {
        return $this->photo;
    }
    public function setPhoto($photo)
    {
        $this->photo = $photo;
        return $this;
    }
}

```

Here, we have specified File for photo property.

Step 3: Create student controller, StudentController and a new method, addAction as shown in the following code.

```

<?php
namespace AppBundle\Controller;
use AppBundle\Entity\Student;
use AppBundle\Form\FormValidationType;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;

```

```

use Symfony\Component\Form\Extension\Core\Type\TextType;
use Symfony\Component\Form\Extension\Core\Type\FileType;
use Symfony\Component\Form\Extension\Core\Type\SubmitType;

class StudentController extends Controller
{
    /**
     * @Route("/student/new")
     */
    public function newAction(Request $request)
    {
        $student = new Student();
        $form = $this->createFormBuilder($student)
            ->add('name', TextType::class)
            ->add('age', TextType::class)
            ->add('photo', FileType::class, array('label' => 'Photo (png, jpeg)'))
            ->add('save', SubmitType::class, array('label' => 'Submit'))
            ->getForm();

        $form->handleRequest($request);
        if ($form->isSubmitted() && $form->isValid()) {
            $file = $student->getPhoto();
            $fileName = md5(uniqid()).'.'.$file->guessExtension();
            $file->move(
                $this->getParameter('photos_directory'),
                $fileName
            );
            $student->setPhoto($fileName);
            return new Response("User photo is successfully uploaded.");
        } else {
            return $this->render('student/new.html.twig', array(
                'form' => $form->createView(),
            ));
        }
    }
}

```


Here, we have created the form for student entity and handled the request. When the form is submitted by the user and it is valid, then we have moved the uploaded file to our upload directory using parameter, **photos_directory**.

Step 4: Create the view, **new.html.twig**, using the following form tags.

```
{% extends 'base.html.twig' %}
{% block javascripts %}
    <script language="javascript"
        src="https://code.jquery.com/jquery-2.2.4.min.js"></script>
{% endblock %}
{% block stylesheets %}
    <style>
#simpleform
{
    width:600px;
    border:2px solid grey;
    padding:14px;
}

#simpleform label
{
    font-size:12px;
    float:left;
    width:300px;
    text-align:right;
    display:block;
}
#simpleform span
{
    font-size:11px;
    color:grey;
    width:100px;
    text-align:right;
    display:block;
}

#simpleform input
```

```

{
    border:1px solid grey;
    font-family:verdana;
    font-size:14px;
    color:grey;
    height:24px;
    width:250px;
    margin: 0 0 20px 10px;
}

#simpleform button
{
    clear:both;
    margin-left:250px;
    background:grey;
    color:#FFFFFF;
    border:solid 1px #666666;
    font-size:16px;
}

</style>
{% endblock %}
{% block body %}
    <h3>Student form</h3>
    <div id="simpleform">
        {{ form_start(form) }}
        {{ form_widget(form) }}
        {{ form_end(form) }}
    </div>
{% endblock %}

```

Step 5: Set the parameter, **photos_directory** in the parameter config file as follows.

app/config/config.xml

```
parameters:
    photos_directory: '%kernel.root_dir%/../web/uploads/photos'
```

Step 6: Now, run the application and open <http://localhost:8000/student/new> and upload a photo. The uploaded photo will be uploaded to the photos_directory and a successful message will be shown.

Result: Initial Page

Student form

Name

Age

Photo (png, jpeg)

Result: File Upload Page

User photo is successfully uploaded.

17. Symfony – Ajax Control

AJAX is a modern technology in web programming. It provides options to send and receive data in a webpage asynchronously, without refreshing the page. Let us learn Symfony AJAX programming in this chapter.

Symfony framework provides options to identify whether the request type is AJAX or not. Request class of Symfony HttpFoundation component has a method, `isXmlHttpRequest()` for this purpose. If an AJAX request is made, the current request object's `isXmlHttpRequest()` method returns true, otherwise false.

This method is used to handle an AJAX request properly in the server side.

```
if ($request->isXmlHttpRequest()) {  
    // Ajax request  
} else {  
    // Normal request  
}
```

Symfony also provides a JSON based Response class, `JsonResponse` to create the response in JSON format. We can combine these two methods to create a simple and clean AJAX based web application.

AJAX – Working Example

Let us add a new page, **student/ajax** in student application and try to fetch the student information asynchronously.

Step 1: Add **ajaxAction** method in `StudentController` (`src/AppBundle/Controller/StudentController.php`).

```
/**  
 * @Route("/student/ajax")  
 */  
public function ajaxAction(Request $request) {  
    $students = $this->getDoctrine()  
        ->getRepository('AppBundle:Student')  
        ->findAll();  
  
    if ($request->isXmlHttpRequest() || $request->query->get('showJson') == 1) {  
        $jsonData = array();  
        $idx = 0;  
        foreach($students as $student) {  
            $temp = array(  

```

```

        'name' => $student->getName(),
        'address' => $student->getAddress(),
    );
    $jsonData[$idx++] = $temp;
}
return new JsonResponse($jsonData);
}
else
{
    return $this->render('student/ajax.html.twig');
}
}

```

Here, if the request is AJAX, we fetch student information, encode it as JSON and return it using **JsonResponse** object. Otherwise, we just render the corresponding view.

Step 2: Create a view file **ajax.html.twig** in Student views directory, **app/Resources/views/student/** and add the following code.

```

{% extends 'base.html.twig' %}
{% block javascripts %}
    <script language="javascript"
        src="https://code.jquery.com/jquery-2.2.4.min.js"></script>
    <script language = "javascript">
$(document).ready(function(){
    $("#loadstudent").on("click", function(event){
        $.ajax({
            url:          '/student/ajax',
            type:          'POST',
            dataType:      'json',
            async:          true,
            success: function(data, status) {
                var e = $('<tr><th>Name</th><th>Address</th></tr>');
                $('#student').html('');
                $('#student').append(e);
                for(i = 0; i < data.length; i++) {
                    student = data[i];
                    var e = $('<tr><td id = "name"></td><td id =
"address"></td></tr>');

```

```

        $('#name', e).html(student['name']);
        $('#address', e).html(student['address']);
        $('#student').append(e);
    }
},
error : function(xhr, textStatus, errorThrown) {
    alert('Ajax request failed.');
```

Here, we have created an anchor tag (id: loadstudent) to load the student information using AJAX call. The AJAX call is done using JQuery. Event attached to loadstudent tag

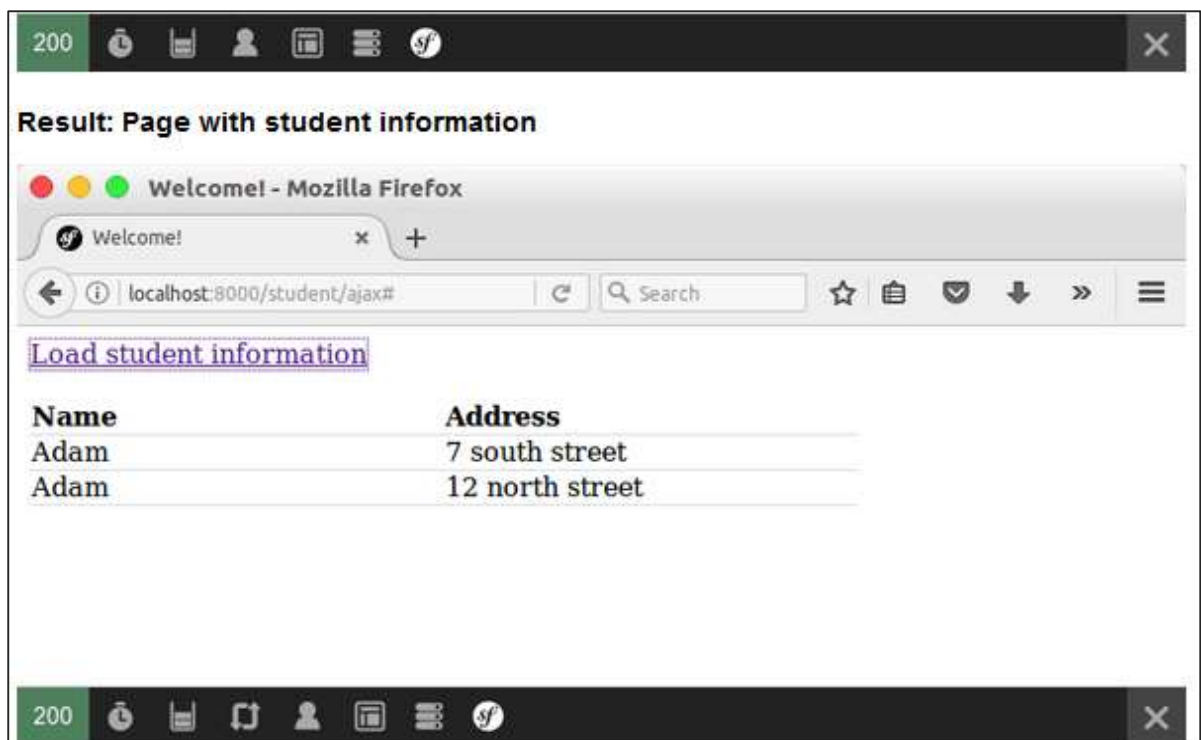
activates when a user clicks it. Then, it will fetch the student information using AJAX call and generate the required HTML code dynamically.

Step 3: Finally, run the application, **http://localhost:8000/student/ajax** and click the Load student information anchor tab.

Result: Initial Page



Result: Page with Student Information



18. Symfony – Cookies & Session Management

Symfony HttpFoundation component provides cookie and session management in an object-oriented manner. **Cookie** provides client-side data storage and it only supports a small amount of data. Usually, it is 2KB per domain and it depends on the browser. **Session** provides server-side data storage and it supports a large amount of data. Let us see how to create a cookie and session in a Symfony web application.

Cookie

Symfony provides Cookie class to create a cookie item. Let us create a cookie **color**, which expires in 24 hours with value **blue**. The constructor parameter of the cookie class is as follows.

- name (type: string) - cookie name
- value (type: string) - cookie value
- expire (type: integer / string / datetime) - expiry information
- path (type: string) - the server path in which the cookie is available
- domain (type: string) – the domain address in which the cookie is available
- secure (type: boolean) - whether the cookie needs to be transmitted in HTTPS connection
- httpOnly (type: boolean) - whether the cookie is available only in HTTP protocol

```
use Symfony\Component\HttpFoundation\Cookie;  
  
$cookie = new Cookie('color', 'green', strtotime('tomorrow'), '/',  
    'somedomain.com', true, true);
```

Symfony also provides the following string-based cookie creation option.

```
$cookie = Cookie::fromString('color=green; expires=Web, 4-May-2017 18:00:00  
+0100; path=/; domain=somedomain.com; secure; httponly');
```

Now, the created cookie needs to be attached to the http response object's header as follows.

```
$response->headers->setCookie($cookie);
```

To get the cookie, we can use Request object as follows.

```
$cookie = $request->cookie->get('color');
```


Here, **request->cookie** is of type **PropertyBag** and we can manipulate it using PropertyBag methods.

Session

Symfony provides a Session class implementing SessionInterface interface. The important session API are as follows,

start - Starts the session.

```
Session $session = new Session();  
$session->start();
```

invalidate - Clears all session data and regenerates the session ID.

set - Stores data in the session using a key.

```
$session->set('key', 'value');
```

We can use any data in the session value, be in simple integer to complex objects.

get - Gets data from the session using the key.

```
$val = $session->get('key');
```

remove - Removes a key from the session.

clear - Removes a session data.

FlashBag

Session provides another useful feature called **FlashBag**. It is a special container inside the session holding the data only during page redirection. It is useful in http redirects. Before redirecting to a page, data can be saved in FlashBag instead of a normal session container and the saved data will be available in the next request (the redirected page). Then, the data will be invalidated automatically.

```
$session->getFlashBag()->add('key', 'value');  
  
$session->getFlashBag()->get('key');
```

19. Symfony – Internationalization

Internationalization (i18n) and **Localization (l10n)** help to increase the customer coverage of a web application. Symfony provides an excellent Translation component for this purpose. Let us learn how to use the Translation component in this chapter.

Enable Translation

By default, Symfony web framework disables Translation component. To enable it, add the translator section in the configuration file, `app/config/config.yml`.

```
framework:
    translator: { fallbacks: [en] }
```

Translation File

Translation component translates the text using the translation resource file. The resource file may be written in PHP, XML, and YAML. The default location of the resource file is **app/Resources/translations**. It needs one resource file per language. Let us write a resource file, **messages.fr.yml** for *French* language.

```
I love Symfony: J'aime Symfony
I love %name%: J'aime %name%
```

The left side text is in English and the right side text is in French. The second line shows the use of a placeholder. The placeholder information can be added dynamically while using translation.

Usage

By default, the default locale of the user's system will be set by the Symfony web framework. If the default locale is not configured in the web application, it will fallback to English. The locale can be set in the URL of the web page as well.

```
http://www.somedomain.com/en/index
http://www.somedomain.com/fr/index
```

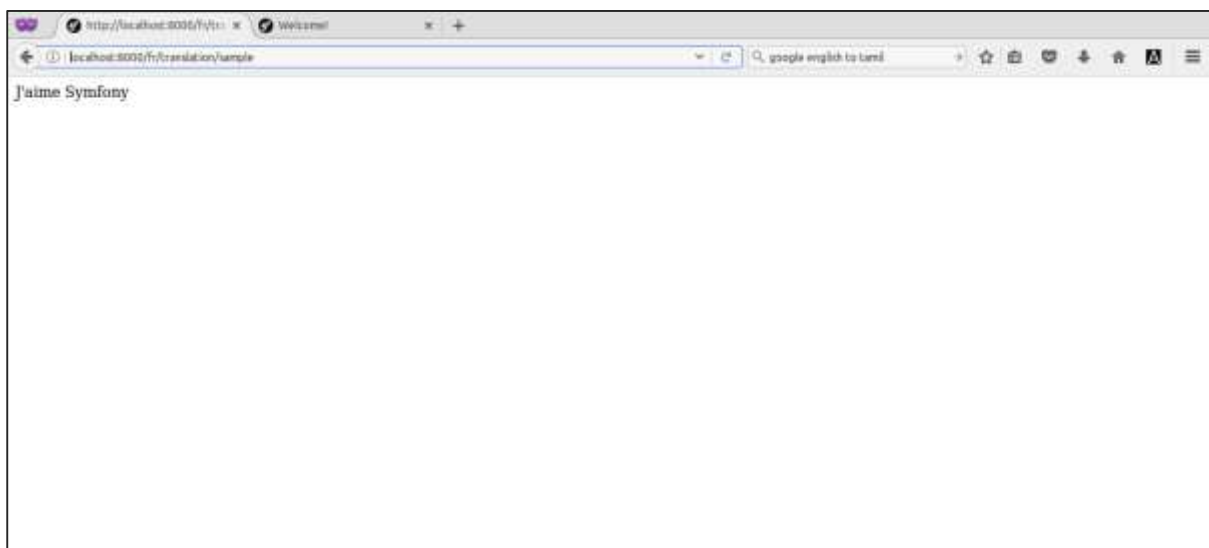
Let us use URL-based locale in our example to easily understand the translation concept. Create a new function, **translationSample** with route **/_{_locale}/translation/sample** in **DefaultController** (`src/AppBundle/Controller/DefaultController.php`). `{_locale}` is a special keyword in Symfony to specify the default locale.

```
/**
 * @Route("/{_locale}/translation/sample", name="translation_sample")
 */
public function translationSample()
```

```
{
    $translated = $this->get('translator')->trans('I love Symfony');
    return new Response($translated);
}
```

Here, we have used translation method, **trans**, which translates the content to the current locale. In this case, the current locale is the first part of the URL. Now, run the application and load the page, **http://localhost:8000/en/translation/sample** in the browser.

The result will be "I love Symfony" in English language. Now, load the page **http://localhost:8000/fr/translation/sample** in the browser. Now, the text will be translated to French as follows.



Similarly, twig template has **{% trans %}** block to enable translation feature in views as well. To check it, add a new function, **translationTwigSample** and the corresponding view at **app/Resources/views/translate/index.html.twig**.

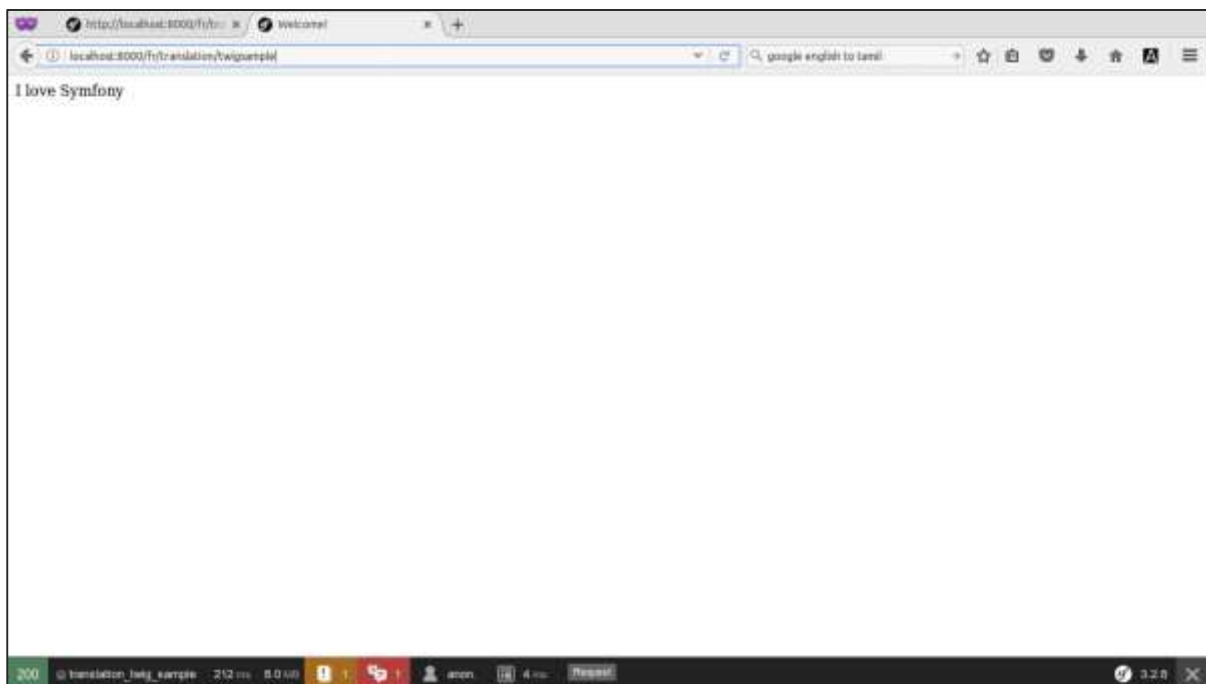
```
/**
 * @Route("/{_locale}/translation/twigsample", name="translation twig sample")
 */
public function translationTwigSample()
{
    return $this->render('translate/index.html.twig');
}
```

View

```
{% extends 'base.html.twig' %}

{% block body %}
    {% trans with {'%name%': 'Symfony'} from "app" into "fr" %}I love %name% {%
endtrans %}
{% endblock %}
```

Here, the trans block specifies the placeholder as well. The page result is as follows.



20. Symfony – Logging

Logging is very important for a web application. Web applications are used by hundreds to thousands of users at a time. To get sneak preview of happenings around a web application, Logging should be enabled. Without logging, the developer will not be able to find the status of the application. Let us consider that an end customer reports an issue or a project stakeholder reports performance issue, then the first tool for the developer is Logging. By checking the log information, one can get some idea about the possible reason of the issue.

Symfony provides an excellent logging feature by integrating **Monolog** logging framework. Monolog is a de-facto standard for logging in PHP environment. Logging is enabled in every Symfony web application and it is provided as a Service. Simply get the logger object using base controller as follows.

```
$logger = $this->get('logger');
```

Once the logger object is fetched, we can log information, warning, and error using it.

```
$logger->info('Hi, It is just a information. Nothing to worry.');
```

```
$logger->warn('Hi, Something is fishy. Please check it.');
```

```
$logger->error('Hi, Some error occurred. Check it now.');
```

```
$logger->critical('Hi, Something catastrophic occurred. Hurry up!');
```

Symfony web application configuration file **app/config/config.yml** has a separate section for the logger framework. It can be used to update the working of the logger framework.

21. Symfony – Email Management

Email functionality is the most requested feature in a web framework. Even a simple application will have a contact form and the details will be sent to the system administration through email. Symfony integrates **SwiftMailer**, the best PHP email module available in the market. SwiftMailer is an excellent email library providing an option to send email using old-school sendmail to the latest cloud-based mailer application.

Let us understand the concept of mailing in Symfony by sending a simple email. Before writing the mailer functionality, set the mailer configuration details in **app/config/parameters.yml**. Then, create a new function, **MailerSample** in **DefaultController** and add the following code.

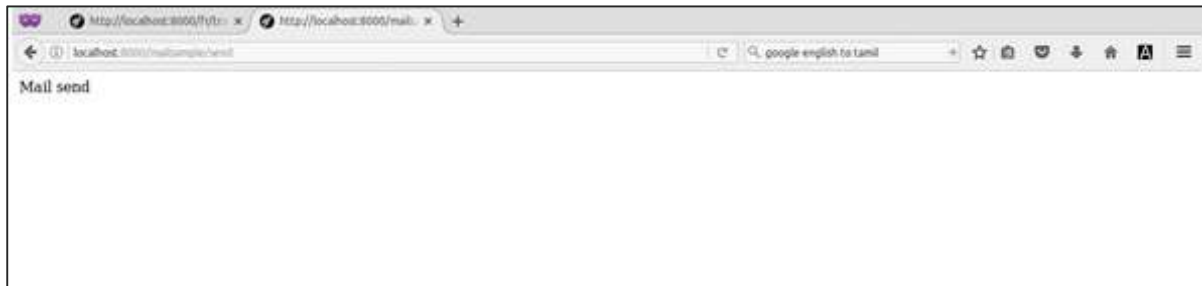
```
/**
 * @Route("/mailsample/send", name="mail_sample_send")
 */
public function MailerSample()
{
    $message = \Swift_Message::newInstance()
        ->setSubject('Hello Email')
        ->setFrom('someone@gmail.com')
        ->setTo('anotherone@gmail.com')
        ->setBody(
            $this->renderView(
                'Emails/sample.html.twig'
            ),
            'text/html'
        );

    $this->get('mailer')->send($message);

    return new Response("Mail send");
}
```

Here, we have simply created a message using **SwiftMailer** component and rendered the body of the message using **Twig** template. Then, we fetched the mailer component from the controller's **get** method with the key 'mailer'. Finally, we sent the message using **send** method and printed the **Mail send** message.

Now, run the page, **<http://localhost:8000/mailexample/send>** and the result would be as follows.



22. Symfony – Unit Testing

Unit test is essential for ongoing development in large projects. Unit tests will automatically test your application's components and alert you when something is not working. Unit testing can be done manually but is often automated.

PHPUnit

Symfony framework integrates with the PHPUnit unit testing framework. To write a unit test for the Symfony framework, we need to set up the PHPUnit. If PHPUnit is not installed, then download and install it. If it is installed properly, then you will see the following response.

```
phpunit
PHPUnit 5.1.3 by Sebastian Bergmann and contributors.
```

Unit test

A unit test is a test against a single PHP class, also called as a unit.

Create a class Student in the Libs/ directory of the AppBundle. It is located at "**src/AppBundle/Libs/Student.php**".

Student.php

```
namespace AppBundle\Libs;
class Student
{
    public function show($name)
    {
        return $name. " , Student name is tested!";
    }
}
```

Now, create a StudentTest file in the "tests/AppBundle/Libs" directory.

StudentTest.php

```
namespace Tests\AppBundle\Libs;
use AppBundle\Libs\Student;

class StudentTest extends \PHPUnit_Framework_TestCase
{
```



```
public function testShow()  
{  
    $stud = new Student();  
    $assign = $stud->show('stud1');  
    $check = "stud1 , Student name is tested!";  
    $this->assertEquals($check, $assign);  
}  
}
```

Run test

To run the test in the directory, use the following command.

```
$ phpunit
```

After executing the above command, you will see the following response.

```
PHPUnit 5.1.3 by Sebastian Bergmann and contributors.  
  
Usage: phpunit [options] UnitTest [UnitTest.php]  
       phpunit [options] <directory>  
  
Code Coverage Options:  
  
--coverage-clover <file>  Generate code coverage report in Clover XML format.  
--coverage-crap4j <file>  Generate code coverage report in Crap4J XML format.  
--coverage-html <dir>     Generate code coverage report in HTML format.
```

Now, run the tests in the Libs directory as follows.

```
$ phpunit tests/AppBundle/Libs
```

Result

```
Time: 26 ms, Memory: 4.00Mb  
OK (1 test, 1 assertion)
```

23. Symfony – Advanced Concepts

In this chapter, we will learn about some advanced concepts in Symfony framework.

HTTP Cache

Caching in a web application improves performance. For example, hot products in a shopping cart web application can be cached for a limited time, so that it can be presented to the customer in a speedy manner without hitting the database. Following are some basic components of Cache.

Cache Item

Cache Item is a single unit of information stored as a key/value pair. The **key** should be string and **value** can be any PHP object. PHP objects are stored as string by serialization and converted back into objects while reading the items.

Cache Adapter

Cache Adapter is the actual mechanism to store the item in a store. The store may be a memory, file system, database, redis, etc. Cache component provides an **AdapterInterface** through which an adapter can store cache item in a back-end store. There are lot of built-in cache adapters available. Few of them are as follows:

- Array Cache adapter - Cache items are stored in PHP array.
- Filesystem Cache adapter - Cache items are stored in files.
- PHP Files Cache Adapter - Cache items are stored as php files.
- APCu Cache Adapter - Cache items are stored in shared memory using PHP APCu extension.
- Redis Cache Adapter - Cache items are stored in Redis server.
- PDO and Doctrine DBAL Cache Adapter - Cache items are stored in the database.
- Chain Cache Adapter - Combines multiple cache adapters for replication purpose.
- Proxy Cache Adapter - Cache items are stored using third party adapter, which implements CacheItemPoolInterface.

Cache Pool

Cache Pool is a logical repository of cache items. Cache pools are implemented by cache adapters.

Simple Application

Let us create a simple application to understand the cache concept.

Step 1: Create a new application, **cache-example**

```
cd /path/to/app
mkdir cache-example
cd cache-example
```

Step 2: Install cache component.

```
composer require symfony/cache
```

Step 3: Create a file system adapter.

```
require __DIR__ . '/vendor/autoload.php';

use Symfony\Component\Cache\Adapter\FilesystemAdapter;

$cache = new FilesystemAdapter();
```

Step 4: Create a cache item using **getItem** and **set** method of adapter. **getItem** fetches the cache item using its key. if the key is not present, it creates a new item. **set** method stores the actual data.

```
$usercache = $cache->getItem('item.users');
$usercache->set(['jon', 'peter']);
$cache->save($usercache);
```

Step 5: Access the cache item using **getItem**, **isHit** and **get** method. **isHit** informs the availability of the cache item and **get** method provides the actual data.

```
$userCache = $cache->getItem('item.users');
if(!$userCache->isHit()) {
    echo "item.users is not available";
} else {
    $users = $userCache->get();
    var_dump($users);
}
```

Step 6: Delete the cache item using **deleteItem** method.

```
$cache->deleteItem('item.users');
```

The complete code listing is as follows.

```
<?php

require __DIR__ . '/vendor/autoload.php';
use Symfony\Component\Cache\Adapter\FilesystemAdapter;

$cache = new FilesystemAdapter();

$usercache = $cache->getItem('item.users');
$usercache->set(['jon', 'peter']);
$cache->save($usercache);

$userCache = $cache->getItem('item.users');
if(!$userCache->isHit()) {
    echo "item.users is not available";
} else {
    $users = $userCache->get();
    var_dump($users);
}

$cache->deleteItem('item.users');

?>
```

Result

```
array(2) {
    [0]=>
    string(3) "jon"
    [1]=>
    string(5) "peter"
}
```

Debug

Debugging is one of the most frequent activity while developing an application. Symfony provides a separate component to ease the process of debugging. We can enable Symfony debugging tools by just calling the **enable** method of Debug class.

```
use Symfony\Component\Debug\Debug;

Debug::enable();
```

Symfony provides two classes, **ErrorHandler** and **ExceptionHandler** for debugging purpose. While ErrorHandler catches PHP errors and converts them into exceptions, RuntimeException or FatalErrorException, ExceptionHandler catches uncaught PHP exceptions and converts them into useful PHP response. ErrorHandler and ExceptionHandler are disabled by default. We can enable it by using the register method.

```
use Symfony\Component\Debug\ErrorHandler;
use Symfony\Component\Debug\ExceptionHandler;

ErrorHandler::register();
ExceptionHandler::register();
```

In a Symfony web application, the debug environment is provided by **DebugBundle**. Register the bundle in AppKernel's **registerBundles** method to enable it.

```
if (in_array($this->getEnvironment(), ['dev', 'test'], true)) {
    $bundles[] = new Symfony\Bundle\DebugBundle\DebugBundle();
}
```

Profiler

Development of an application needs a world-class profiling tool. The profiling tool collects all the run-time information about an application such as execution time, execution time of individual modules, time taken by a database activity, memory usage, etc. A web application needs much more information such as the time of request, time taken to create a response, etc. in addition to the above metrics.

Symfony enables all such information in a web application by default. Symfony provides a separate bundle for web profiling called **WebProfilerBundle**. Web profiler bundle can be enabled in a web application by registering the bundle in the AppKernel's registerBundles method.

```
if (in_array($this->getEnvironment(), ['dev', 'test'], true)) {
    $bundles[] = new Symfony\Bundle\WebProfilerBundle\WebProfilerBundle();
}
```

The web profile component can be configured under **web_profile section** of the application configuration file, **app/config/config.xml**

```
web_profiler:
    toolbar:      false
    position:     bottom
```

Symfony application shows the profiled data at the bottom of the page as a distinct section.



Symfony also provides an easy way to add custom details about the page in the profile data using **DataCollectorInterface** interface and twig template. In short, Symfony enables a web developer to create a world-class application by providing a great profiling framework with relative ease.

Security

As discussed earlier, Symfony provides a robust security framework through its security component. The security component is divided into four sub-components as follows.

- symfony/security-core - Core security functionality.
- symfony/security-http - Integrated security feature in HTTP protocol.
- symfony/security-csrf - Protection against cross-site request forgery in a web application.
- symfony/security-acl - Advanced access control list based security framework.

Simple Authentication and Authorization

Let us learn the concept of authentication and authorization using a simple demo application.

Step 1: Create a new web application **securitydemo** using the following command.

```
symfony new securitydemo
```

Step 2: Enable the security feature in the application using the security configuration file. The security related configuration are placed in a separate file, **security.yml**. The default configuration is as follows.

```
security:
    providers:
        in_memory:
            memory: ~

    firewalls:
        dev:
            pattern: ^/(_(profiler|wdt)|css|images|js)/
            security: false

    main:
        anonymous: ~
        #http_basic: ~
        #form_login: ~
```

The default configuration enables memory-based security provider and anonymous access to all pages. The firewall section excludes the files matching the pattern, **^/(_(profiler|wdt)|css|images|js)/** from the security framework. The default pattern includes stylesheets, images, and JavaScripts (plus dev tools like profiler).

Step 3: Enable HTTP based security authenticate system by adding http_basic option in main section as follows.

```
security:
    # ...

    firewalls:
        # ...

    main:
        anonymous: ~
        http_basic: ~
        #form_login: ~
```

Step 4: Add some users in the memory provider section. Also, add roles for the users.

```
security:
    providers:
        in_memory:
            memory:
                users:
                    myuser:
                        password: user
                        roles: 'ROLE_USER'
                    myadmin:
                        password: admin
                        roles: 'ROLE_ADMIN'
```

We have added two users, *user* in role `ROLE_USER` and *admin* in role `ROLE_ADMIN`.

Step 5: Add the encoder to get complete details of the current logged-in user. The purpose of the encoder is to get complete details of the current user object from the web request.

```
security:
    # ...
    encoders:
        Symfony\Component\Security\Core\User\User: bcrypt
    # ...
```

Symfony provides an interface, **UserInterface** to get user details such as username, roles, password, etc. We need to implement the interface to our requirement and configure it in the encoder section.

For example, let us consider that the user details are in the database. Then, we need to create a new User class and implement UserInterface methods to get the user details from the database. Once the data is available, then the security system uses it to allow/deny the user. Symfony provides a default User implementation for Memory provider. Algorithm is used to decrypt the user password.

Step 6: Encrypt the user password using **bcrypt** algorithm and place it in the configuration file. Since we used **bcrypt** algorithm, User object tries to decrypt the password specified in configuration file and then tries to match with the password entered by the user. Symfony console application provides a simple command to encrypt the password.

```
php bin/console security:encode-password admin
Symfony Password Encoder Utility
=====
```

```
-----
```


Key	Value
Encoder used	Symfony\Component\Security\Core\Encoder\BCryptPasswordEncoder
Encoded password	\$2y\$12\$0Hy6/.MNxWdFcCRDdstHU.hT5j3Mg1tqBunMLIUYkz6..IucpaPNO
! [NOTE] Bcrypt encoder used: the encoder generated its own built-in salt.	
[OK] Password encoding succeeded	

Step 7: Use the command to generate the encrypted password and update it in the configuration file.

```
# To get started with security, check out the documentation:
# http://symfony.com/doc/current/security.html
security:

    # http://symfony.com/doc/current/security.html#b-configuring-how-users-are-loaded
    providers:

        in_memory:
            memory:
                users:
                    user:
                        password:
$2y$13$WsGWNufreEnVK1InBXL2cO/U7WftvfNvHVb/IJBH6JiYoDwVN4zoi
                        roles: 'ROLE_USER'
                    admin:
                        password:
$2y$13$jQNdIeoNV1BKVbpnBuhKRuOL01NeMKF7nEqEi/MqlzgtS0njK3toy
                        roles: 'ROLE_ADMIN'

    encoders:

        Symfony\Component\Security\Core\User\User: bcrypt

    firewalls:

        # disables authentication for assets and the profiler, adapt it
        according to your needs
        dev:
            pattern: ^/(_(profiler|wdt)|css|images|js)/
```

```

        security: false

    main:
        anonymous: ~
        # activate different ways to authenticate

        # http://symfony.com/doc/current/security.html#a-configuring-how-
        your-users-will-authenticate
        http_basic: ~

        #
        http://symfony.com/doc/current/cookbook/security/form_login_setup.html
        #form_login: ~

```

Step 8: Now, apply the security to some section of the application. For example, restrict admin section to the users in role, ROLE_ADMIN.

```

security:
    # ...
    firewalls:
        # ...
        default:
            # ...

    access_control:
        # require ROLE_ADMIN for /admin*
        - { path: ^/admin, roles: 'ROLE_ADMIN' }

```

Step 9: Add an admin page in DefaultController as follows.

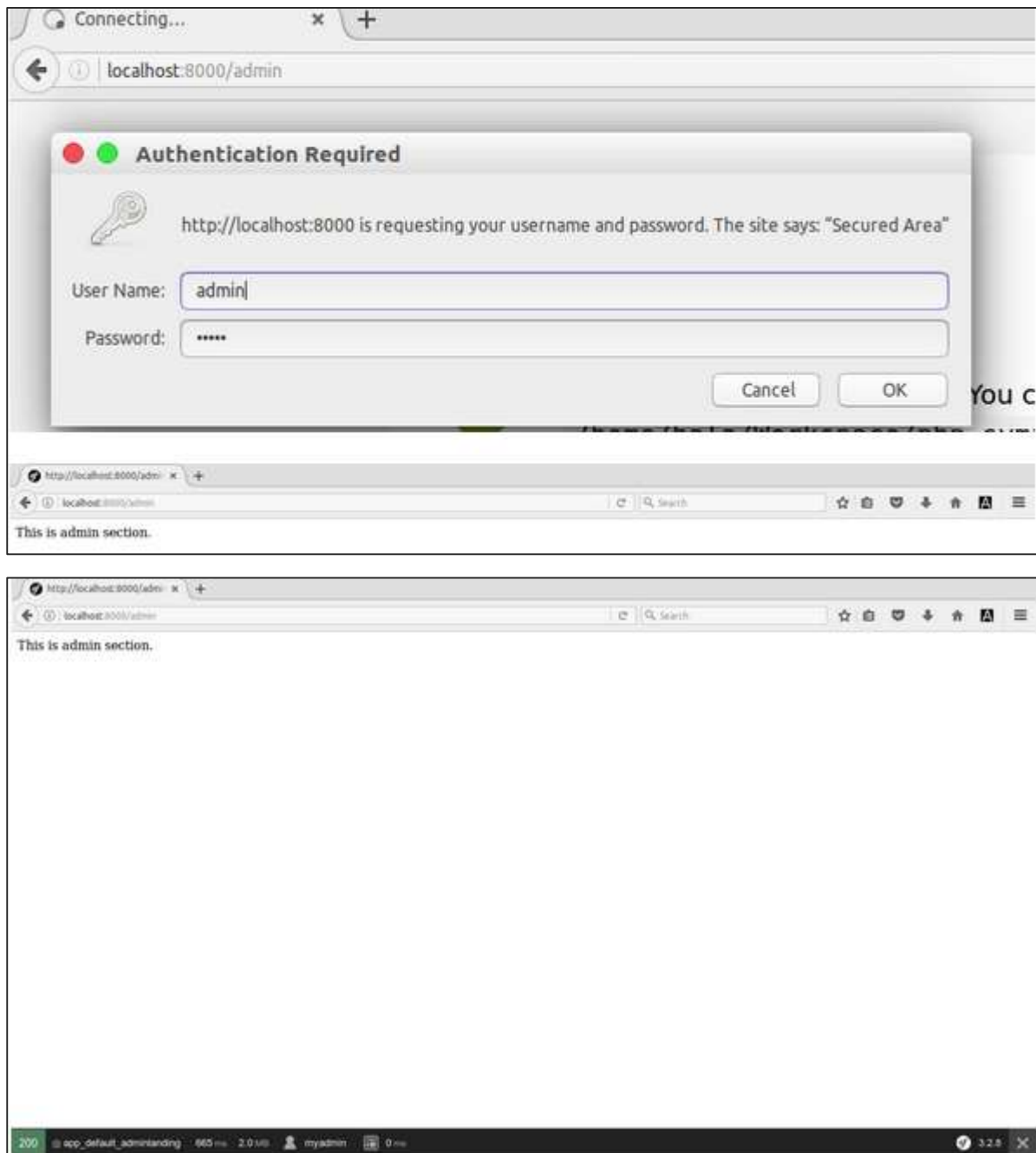
```

/**
 * @Route("/admin")
 */
public function adminLandingAction()
{
    return new Response('<html><body>This is admin
section.</body></html>');
}

```

Step 10: Finally, access the admin page to check the security configuration in a browser. The browser will ask for the username and password and only allow configured users.

Result



Workflow

Workflow is an advanced concept having usage in many enterprise applications. In an ecommerce application, the product delivery process is a workflow. The product is first billed (order creation), procured from the store and packaged (packaging/ready to dispatch), and dispatched to the user. If there is any issue, the product returns from the user and the order is reverted. Order of the flow of action is very important. For example, we can't deliver a product without billing.

Symfony component provides an object-oriented way to define and manage a workflow. Each step in a process is called **place** and the action required to move from one place to another is called **transition**. The collection of places and transition to create a workflow is called a **Workflow definition**.

Let us understand the concept of workflow by creating a simple application for leave management.

Step 1: Create a new application, **workflow-example**.

```
cd /path/to/dev
mkdir workflow-example
cd workflow-example
composer require symfony/workflow
```

Step 2: Create a new class, **Leave** having **applied_by**, **leave_on** and **status** attributes.

```
class Leave
{
    public $applied_by;
    public $leave_on;

    public $status;
}
```

Here, **applied_by** refers to the employees who want leave. **leave_on** refers to the date of leave. **status** refers to the leave status.

Step 3: Leave management has four places, **applied**, **in_process** and **approved / rejected**.

```
use Symfony\Component\Workflow\DefinitionBuilder;
use Symfony\Component\Workflow\Transition;
use Symfony\Component\Workflow\Workflow;
use Symfony\Component\Workflow\MarkingStore\SingleStateMarkingStore;
use Symfony\Component\Workflow\Registry;
use Symfony\Component\Workflow\Dumper\GraphvizDumper;
```

```
$builder = new DefinitionBuilder();
$builder->addPlaces(['applied', 'in_process', 'approved', 'rejected']);
```

Here, we have created a new definition using **DefinitionBuilder** and added places using **addPlaces** method.

Step 4: Define the actions required to move from one place to another place.

```
$builder->addTransition(new Transition('to_process', 'applied',
'in_process'));
$builder->addTransition(new Transition('approve', 'in_process', 'approved'));
$builder->addTransition(new Transition('reject', 'in_process', 'rejected'));
```

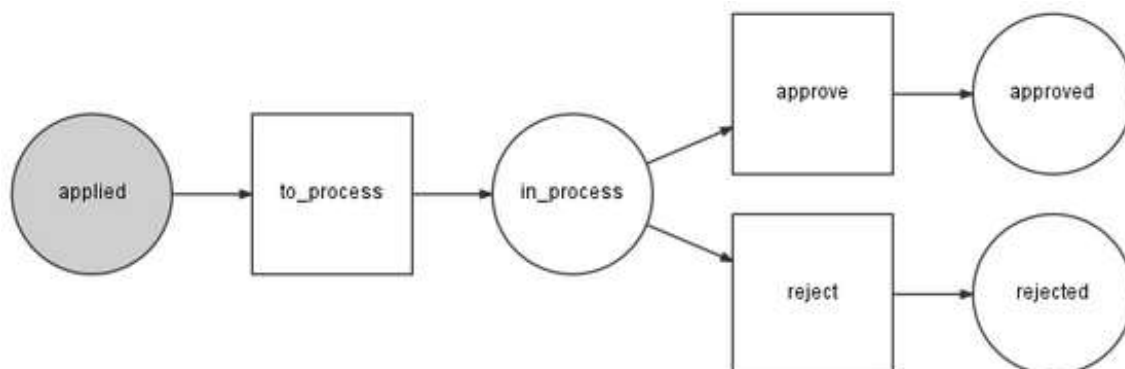
Here, we have three transitions, **to_process**, **approve** and **reject**. **to_process** transition accepts the leave application and moves the place from applied to in_process. **approve** transition approves the leave application and moves the place to approved. Similarly, **reject** transition rejects the leave application and moves the place to rejected. We have created all transitions using **addTransition** method.

Step 5: Build the definition using **build** method.

```
$definition = $builder->build();
```

Step 6: Optionally, the definition can be dumped as graphviz dot format, which can be converted to image file for reference purpose.

```
$dumper = new GraphvizDumper();
echo $dumper->dump($definition);
```



Step 7: Create a marking store, which is used to store the current places/status of the object.

```
$marking = new SingleStateMarkingStore('status');
```

Here, we have used **SingleStateMarkingStore** class to create the mark and it marks the current status into the status property of the object. In our example, the object is Leave object.

Step 8: Create the workflow using definition and marking.

```
$leaveWorkflow = new Workflow($definition, $marking);
```

Here, we have used **Workflow** class to create the workflow.

Step 9: Add the workflow into the registry of the workflow framework using **Registry** class.

```
$registry = new Registry();
$registry->add($leaveWorkflow, Leave::class);
```

Step 10: Finally, use the workflow to find whether a given transition is applied using **can** method and if so, apply the transition using **apply** method. When a transition is applied, the status of the object moves from one place to another.

```
$workflow = $registry->get($leave);
echo "Can we approve the leave now? " . $workflow->can($leave, 'approve') .
"\r\n";
echo "Can we approve the start process now? " . $workflow->can($leave,
'to_process') . "\r\n";
$workflow->apply($leave, 'to_process');
echo "Can we approve the leave now? " . $workflow->can($leave, 'approve') .
"\r\n";
echo $leave->status . "\r\n";
$workflow->apply($leave, 'approve');
echo $leave->status . "\r\n";
```

The complete coding is as follows:

```
<?php

require __DIR__ . '/vendor/autoload.php';

use Symfony\Component\Workflow\DefinitionBuilder;
use Symfony\Component\Workflow\Transition;
use Symfony\Component\Workflow\Workflow;
use Symfony\Component\Workflow\MarkingStore\SingleStateMarkingStore;
use Symfony\Component\Workflow\Registry;
use Symfony\Component\Workflow\Dumper\GraphvizDumper;
```

```

class Leave
{
    public $applied_by;
    public $leave_on;

    public $status;
}

$builder = new DefinitionBuilder();
$builder->addPlaces(['applied', 'in_process', 'approved', 'rejected']);
$builder->addTransition(new Transition('to_process', 'applied', 'in_process'));
$builder->addTransition(new Transition('approve', 'in_process', 'approved'));
$builder->addTransition(new Transition('reject', 'in_process', 'rejected'));
$definition = $builder->build();

// $dumper = new GraphvizDumper();
// echo $dumper->dump($definition);

$marking = new SingleStateMarkingStore('status');
$leaveWorkflow = new Workflow($definition, $marking);

$registry = new Registry();
$registry->add($leaveWorkflow, Leave::class);

$leave = new Leave();
$leave->applied_by = "Jon";
$leave->leave_on = "1998-12-12";
$leave->status = 'applied';

$workflow = $registry->get($leave);
echo "Can we approve the leave now? " . $workflow->can($leave, 'approve') .
"\r\n";
echo "Can we approve the start process now? " . $workflow->can($leave,
'to_process') . "\r\n";
$workflow->apply($leave, 'to_process');

```

```
echo "Can we approve the leave now? " . $workflow->can($leave, 'approve') .
"\r\n";
echo $leave->status . "\r\n";
$workflow->apply($leave, 'approve');
echo $leave->status . "\r\n";

?>
```

Result

```
Can we approve the leave now?
Can we approve the start process now? 1
Can we approve the leave now? 1
in_process
approved
```


24. Symfony – Symfony REST Edition

In any modern application, REST service is one of the core fundamental building blocks. Be it a web-based application or a slick mobile application, the front-end is usually a well designed interface for the back-end REST services. Symfony REST edition provides a readymade template to kick-start our REST based web application.

Let us learn how to install a template REST application using Symfony REST edition.

Step 1: Download Symfony REST edition using the following command.

```
composer create-project gimler/symfony-rest-edition --stability=dev  
path/to/install
```

This will download the Symfony REST edition.

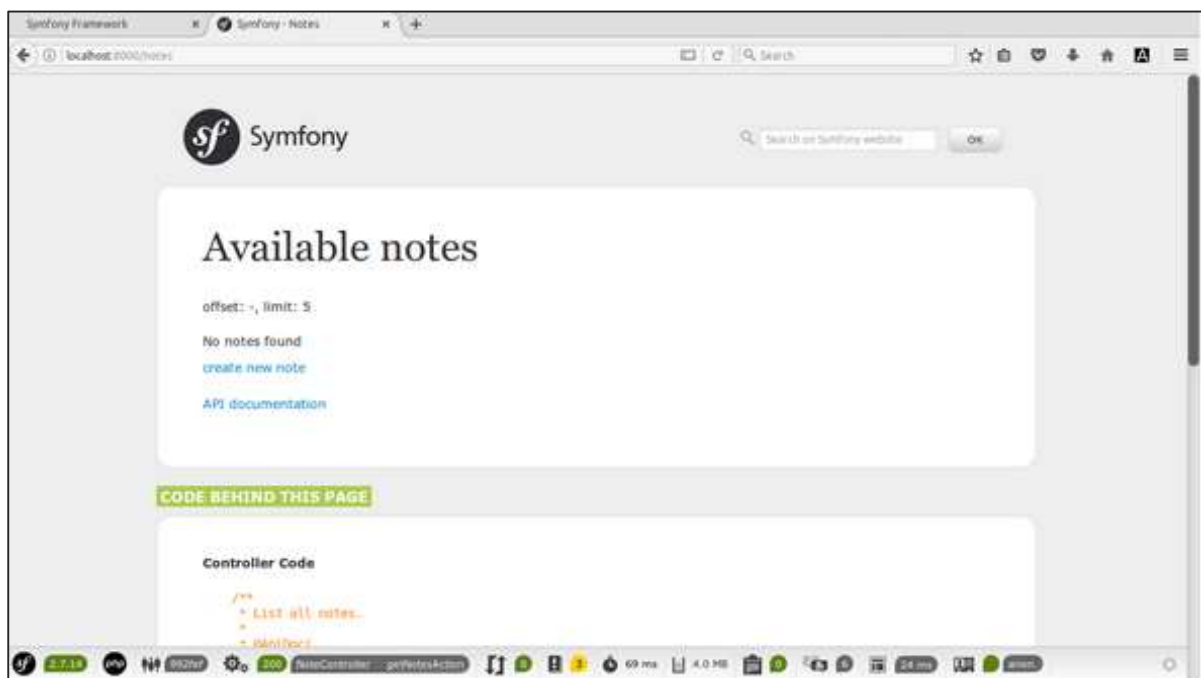
Step 2: Try to configure it by asking some questions. For all the questions, select the default answer except database. For database, select pdo_sqlite. You may need to enable PHP's sqlite extension, if it is not already installed.

Step 3: Now, run the application using the following command.

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

Step 4: Finally, open the application in the browser using <http://localhost:8000/>

It will produce the following result:



25. Symfony – Symfony CMF Edition

Content management system is one of the largest market in the web application scenario. There are a lot of frameworks available for content management system, in virtually all languages under the sun. Most of the frameworks are easy to work as an end customer but very hard to work with as a developer and vice-versa.

Symfony provides a simple and easy framework for a developer to start with. It has all the basic features expected by an end customer as well. In short, it is the responsibility of the developer to provide a great experience for the end customer.

Let us see how to install a CMS application template using Symfony CMF edition.

Step 1: Download Symfony CMF sandbox using the following command.

```
composer create-project symfony-cmf/sandbox cmf-sandbox
```

This will download the Symfony CMF.

Step 2: Try to configure it by asking some questions. For all the questions, select the default answer except database. For database, select pdo_sqlite. You may need to enable PHP's sqlite extension, if it is not already installed.

Step 3: Create demo database using the console application as follows.

```
php app/console doctrine:database:create
```

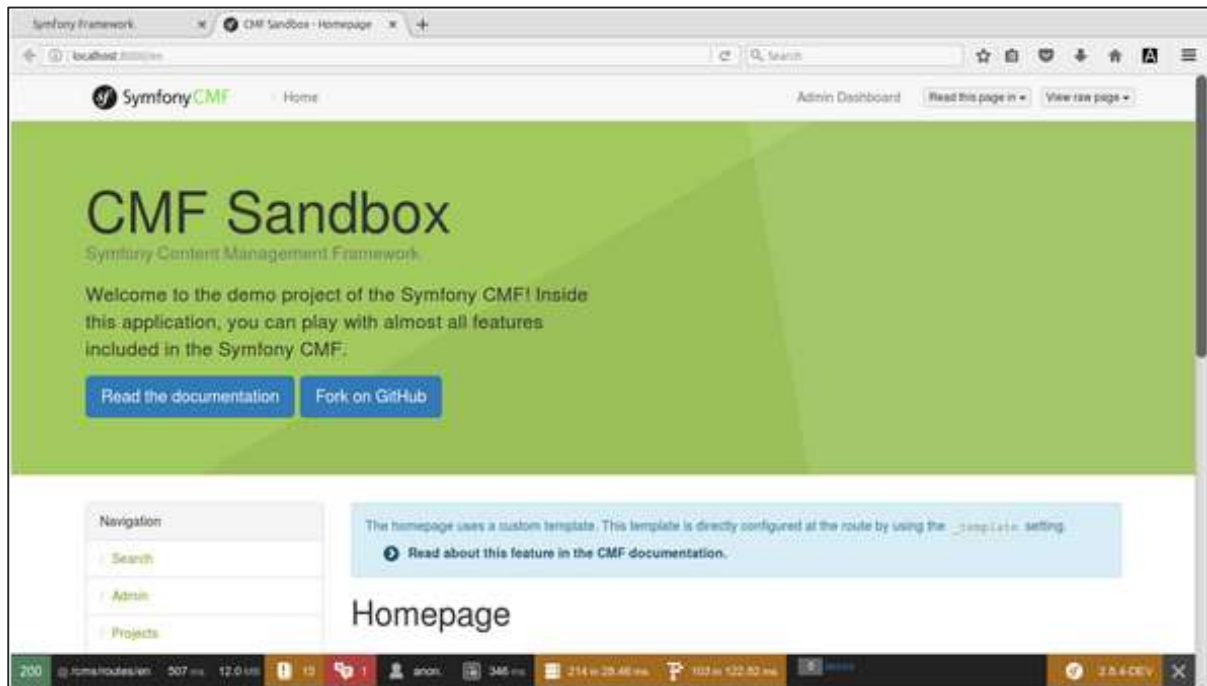
Step 4: Load the demo data into the database using the following command.

```
php app/console doctrine:phpcr:init:dbal --force
php app/console doctrine:phpcr:repository:init
php app/console doctrine:phpcr:fixtures:load -n
```

Step 5: Now, run the application using the following command.

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

Step 6: Finally, open the application in the browser using `http://localhost:8000/`
It will produce the following output:



26. Symfony – Complete Working Example

In this chapter, we will learn how to create a complete MVC based **BookStore Application** in Symfony Framework. Following are the steps.

Step 1: Create a Project

Let's create a new project named "BookStore" in Symfony using the following command.

```
symfony new BookStore
```

Step 2: Create a Controller and Route

Create a BooksController in "src/AppBundle/Controller" directory. It is defined as follows.

BooksController.php

```
<?php

namespace AppBundle\Controller;

use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Response;

class BooksController
{
    /**
     * @Route("/books/author")
     */
    public function authorAction()
    {
        return new Response('Book store application!');
    }
}
```

Now, we have created a BooksController, next create a view to render the action.

Step 3: Create a View

Let's create a new folder named "Books" in "app/Resources/views/" directory. Inside the folder, create a file "author.html.twig" and add the following changes.

author.html.twig

```
<h3> Simple book store application </h3>
```

Now, render the view in BooksController class. It is defined as follows.

BooksController.php

```
<?php

namespace AppBundle\Controller;

use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Response;

class BooksController extends Controller
{
    /**
     * @Route("/books/author")
     */
    public function authorAction()
    {
        return $this->render('books/author.html.twig');
    }
}
```

As of now, we have created a basic BooksController and the result is rendered. You can check the result in the browser using the URL "http://localhost:8000/books/author".

Step 4: Database Configuration

Configure the database in "app/config/parameters.yml" file.

Open the file and add the following changes.

parameter.yml

```
# This file is auto-generated during the composer install

parameters:
    database_driver: pdo_mysql
    database_host: localhost
    database_port: 3306
    database_name: booksdb
    database_user: <database_username>
    database_password: <database_password>
    mailer_transport: smtp
    mailer_host: 127.0.0.1
    mailer_user: null
    mailer_password: null
    secret: 0ad4b6d0676f446900a4cb11d96cf0502029620d
    doctrine:
        dbal:
            driver: pdo_mysql
            host: '%database_host%'
            dbname: '%database_name%'
            user: '%database_user%'
            password: '%database_password%'
            charset: utf8mb4
```

Now, Doctrine can connect to your database “booksdb”.

Step 5: Create a Database

Issue the following command to generate “booksdb” database. This step is used to bind the database in Doctrine.

```
php bin/console doctrine:database:create
```

After executing the command, it automatically generates an empty “booksdb” database. You can see the following response on your screen.

It will produce the following result:

```
Created database `booksdb` for connection named default
```

Step 6: Mapping Information

Create a Book entity class inside the Entity directory which is located at "src/AppBundle/Entity".

You can directly pass Book class using annotations. It is defined as follows.

Book.php

Add the following code in the file.

```
<?php
namespace AppBundle\Entity;
use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity
 * @ORM\Table(name="Books")
 */

class Book
{
    /**
     * @ORM\Column(type="integer")
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="AUTO")
     */
    private $id;

    /**
     * @ORM\Column(type="string", length=50)
     */
    private $name;

    /**
     * @ORM\Column(type="string", length=50)
     */

    private $author;
```

```

    /**
     * @ORM\Column(type="decimal", scale=2)
     */
    private $price;
}

```

Here, the table name is optional.

If the table name is not specified, then it will be determined automatically based on the name of the entity class.

Step 7: Bind an Entity

Doctrine creates simple entity classes for you. It helps you build any entity.

Issue the following command to generate an entity.

```
php bin/console doctrine:generate:entities AppBundle\Entity/Book
```

Then you will see the following result and the entity will be updated.

```

Generating entity "AppBundle\Entity\Book"
  > backing up Book.php to Book.php~
  > generating AppBundle\Entity\Book

```

Book.php

```

<?php

namespace AppBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity
 * @ORM\Table(name="Books")
 */
class Book
{
    /**
     * @ORM\Column(type="integer")
     * @ORM\Id

```



```
* @ORM\GeneratedValue(strategy="AUTO")
*/
private $id;

/**
 * @ORM\Column(type="string", length=50)
 */
private $name;

/**
 * @ORM\Column(type="string", length=50)
 */
private $author;

/**
 * @ORM\Column(type="decimal", scale=2)
 */
private $price;

/**
 * Get id
 *
 * @return integer
 */
public function getId()
{
    return $this->id;
}

/**
 * Set name
 *
 * @param string $name
 *
 * @return Book
 */
```

```
public function setName($name)
{
    $this->name = $name;
    return $this;
}

/**
 * Get name
 *
 * @return string
 */
public function getName()
{
    return $this->name;
}

/**
 * Set author
 *
 * @param string $author
 *
 * @return Book
 */
public function setAuthor($author)
{
    $this->author = $author;
    return $this;
}

/**
 * Get author
 *
 * @return string
 */
public function getAuthor()
{

```

```
        return $this->author;
    }

    /**
     * Set price
     *
     * @param string $price
     *
     * @return Book
     */
    public function setPrice($price)
    {
        $this->price = $price;
        return $this;
    }

    /**
     * Get price
     *
     * @return string
     */
    public function getPrice()
    {
        return $this->price;
    }
}
```

Step 8: Mapping Validation

After creating entities, you should validate the mappings using the following command.

```
php bin/console doctrine:schema:validate
```

It will produce the following result:

```
[Mapping] OK - The mapping files are correct.
```

```
[Database] FAIL - The database schema is not in sync with the current mapping file.
```

Since we have not created the Books table, the entity is out of sync. Let us create the Books table using Symfony command in the next step.

Step 9: Creating Schema

Doctrine can automatically create all the database tables needed for Book entity. This can be done using the following command.

```
php bin/console doctrine:schema:update --force
```

After executing the command, you will see the following response.

```
Updating database schema...  
Database schema updated successfully! "1" query was executed
```

Now, again validate the schema using the following command.

```
php bin/console doctrine:schema:validate
```

It will produce the following result:

```
[Mapping] OK - The mapping files are correct.  
[Database] OK - The database schema is in sync with the mapping files.
```

Step 10: Getter and Setter

As seen in the Bind an Entity section, the following command generates all the getters and setters for the Book class.

```
$ php bin/console doctrine:generate:entities AppBundle/Entity/Book
```

Step 11: Fetching Objects from the Database

Create a method in BooksController that will display the books' details.

BooksController.php

```
/**  
 * @Route("/books/display", name="app_book_display")  
 */  
public function displayAction()  
{  
    $bk = $this->getDoctrine()
```

```

        ->getRepository('AppBundle:Book')
        ->findAll();
    return $this->render('books/display.html.twig', array('data' => $bk));
}

```

Step 12: Create a View

Let's create a view that points to display action. Move to the views directory and create file "display.html.twig". Add the following changes in the file.

display.html.twig

```

{% extends 'base.html.twig' %}
{% block stylesheets %}
<style>
.table { border-collapse: collapse; }
.table th, td {
    border-bottom: 1px solid #ddd;
    width: 250px;
    text-align: left;
    align: left;
}
</style>
{% endblock %}

{% block body %}
<h2>Books database application!</h2>

<table class="table">
    <tr>
        <th>Name</th>
        <th>Author</th>
        <th>Price</th>
    </tr>
{% for x in data %}
    <tr>
        <td>{{ x.Name }}</td>
        <td>{{ x.Author }}</td>

```

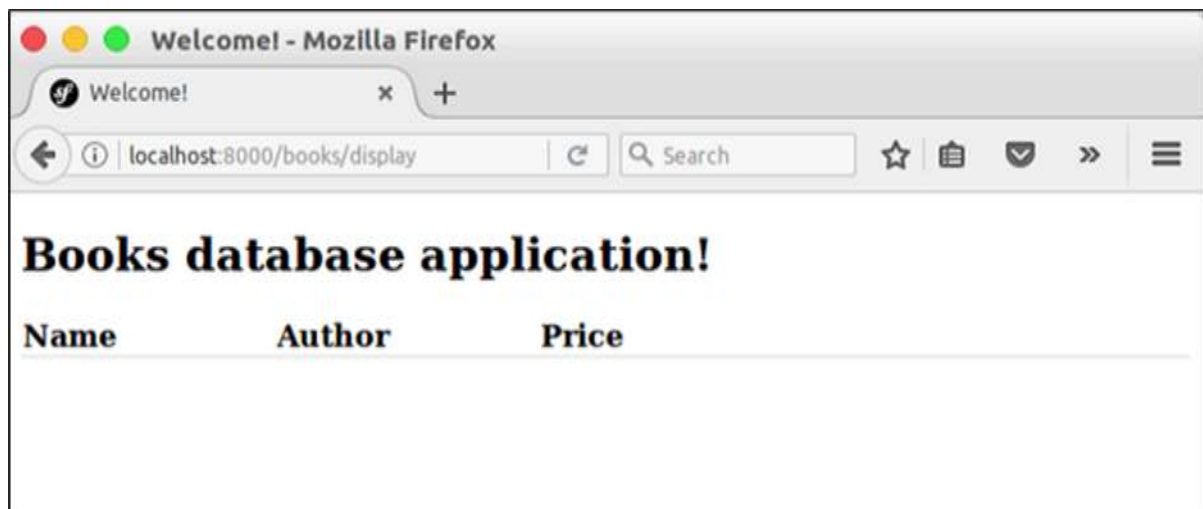
```

        <td>{{ x.Price }}</td>
    </tr>
{% endfor %}
</table>
{% endblock %}

```

You can obtain the result by requesting the URL "http://localhost:8000/books/display" in the browser.

Result



Step 13: Add a Book Form

Let's create a functionality to add a book into the system. Create a new page, newAction method in the BooksController as follows.

```

// use section
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\Form\Extension\Core\Type\TextType;
use Symfony\Component\Form\Extension\Core\Type\SubmitType;

// methods section
/**
 * @Route("/books/new")
 */
public function newAction(Request $request)
{
    $stud = new StudentForm();
}

```

```

    $form = $this->createFormBuilder($stud)
        ->add('name', TextType::class)
        ->add('author', TextType::class)
        ->add('price', TextType::class)
        ->add('save', SubmitType::class, array('label' => 'Submit'))
        ->getForm();

    return $this->render('books/new.html.twig', array(
        'form' => $form->createView(),
    ));
}

```

Step 14: Create a View For Book Form

Let's create a view that points to a new action. Move to the views directory and create a file "new.html.twig". Add the following changes in the file.

```

{% extends 'base.html.twig' %}
{% block stylesheets %}
    <style>
#simpleform
{
    width:600px;
    border:2px solid grey;
    padding:14px;
}
#simpleform label
{
    font-size:14px;
    float:left;
    width:300px;
    text-align:right;
    display:block;
}
#simpleform span
{
    font-size:11px;
    color:grey;
}

```

```
        width:100px;
        text-align:right;
        display:block;
    }

    #simpleform input
    {
        border:1px solid grey;
        font-family:verdana;
        font-size:14px;
        color:light blue;
        height:24px;
        width:250px;
        margin: 0 0 10px 10px;
    }

    #simpleform textarea
    {
        border:1px solid grey;
        font-family:verdana;
        font-size:14px;
        color:light blue;
        height:120px;
        width:250px;
        margin: 0 0 20px 10px;
    }

    #simpleform select
    {
        margin: 0 0 20px 10px;
    }

    #simpleform button
    {
        clear:both;
        margin-left:250px;
        background: grey;
```



```

        color:#FFFFFF;
        border:solid 1px #666666;
        font-size:16px;
    }

    </style>
{% endblock %}

{% block body %}
    <h3>Book details:</h3>
    <div id="simpleform">
        {{ form_start(form) }}
        {{ form_widget(form) }}
        {{ form_end(form) }}
    </div>
{% endblock %}

```

It will produce the following screen as output:

The screenshot shows a Mozilla Firefox browser window with the title 'Welcome! - Mozilla Firefox'. The address bar shows 'localhost:8000/books/new'. The page content displays 'Book details:' followed by a form. The form is enclosed in a box and contains three text input fields labeled 'Name', 'Author', and 'Price'. Below these fields is a 'Submit' button.

Step 15: Collect Book Information and Store It

Let's change the newAction method and include the code to handle form submission. Also, store the book information into the database.

```
/**
```

```

* @Route("/books/new", name="app_book_new")
*/
public function newAction(Request $request)
{
    $book = new Book();
    $form = $this->createFormBuilder($book)
        ->add('name', TextType::class)
        ->add('author', TextType::class)
        ->add('price', TextType::class)
        ->add('save', SubmitType::class, array('label' => 'Submit'))
        ->getForm();

    $form->handleRequest($request);

    if ($form->isSubmitted() && $form->isValid()) {
        $book = $form->getData();
        $doct = $this->getDoctrine()->getManager();

        // tells Doctrine you want to save the Product
        $doct->persist($book);

        //executes the queries (i.e. the INSERT query)
        $doct->flush();

        return $this->redirectToRoute('app_book_display');
    } else {
        return $this->render('books/new.html.twig', array(
            'form' => $form->createView(),
        ));
    }
}

```

Once the book is stored into the database, redirect to the book display page.

Step 16: Updating the Book

To update the book, create an action, `updateAction`, and add the following changes.

```

/**
 * @Route("/books/update/{id}", name="app_book_update" )
 */
public function updateAction($id, Request $request)
{
    $doct = $this->getDoctrine()->getManager();
    $bk = $doct->getRepository('AppBundle:Book')->find($id);

    if (!$bk) {
        throw $this->createNotFoundException(
            'No book found for id '.$id
        );
    }

    $form = $this->createFormBuilder($bk)
        ->add('name', TextType::class)
        ->add('author', TextType::class)
        ->add('price', TextType::class)
        ->add('save', SubmitType::class, array('label' => 'Submit'))
        ->getForm();

    $form->handleRequest($request);

    if ($form->isSubmitted() && $form->isValid()) {
        $book = $form->getData();
        $doct = $this->getDoctrine()->getManager();

        // tells Doctrine you want to save the Product
        $doct->persist($book);

        //executes the queries (i.e. the INSERT query)
        $doct->flush();
        return $this->redirectToRoute('app_book_display');
    } else {

        return $this->render('books/new.html.twig', array(

```

```

        'form' => $form->createView(),
    ));
}
}

```

Here, we are processing two functionalities. If the request only contains **id**, then we fetch it from the database and show it in the book form. And, if the request contains full book information, then we update the details in the database and redirect to the book display page.

Step 17: Deleting an Object

Deleting an object requires a call to the `remove()` method of the entity (doctrine) manager. This can be done using the following code.

```

/**
 * @Route("/books/delete/{id}", name="app_book_delete")
 */
public function deleteAction($id)
{
    $doct = $this->getDoctrine()->getManager();
    $bk = $doct->getRepository('AppBundle:Book')->find($id);
    if (!$bk) {
        throw $this->createNotFoundException(
            'No book found for id '.$id
        );
    }
    $doct->remove($bk);
    $doct->flush();
    return $this->redirectToRoute('app_book_display');
}

```

Here, we deleted the book and redirected to book display page.

Step 18: Include Add / Edit / Delete Functionality in Display Page

Now, update the body block in display view and include include add / edit / delete links as follows.

```

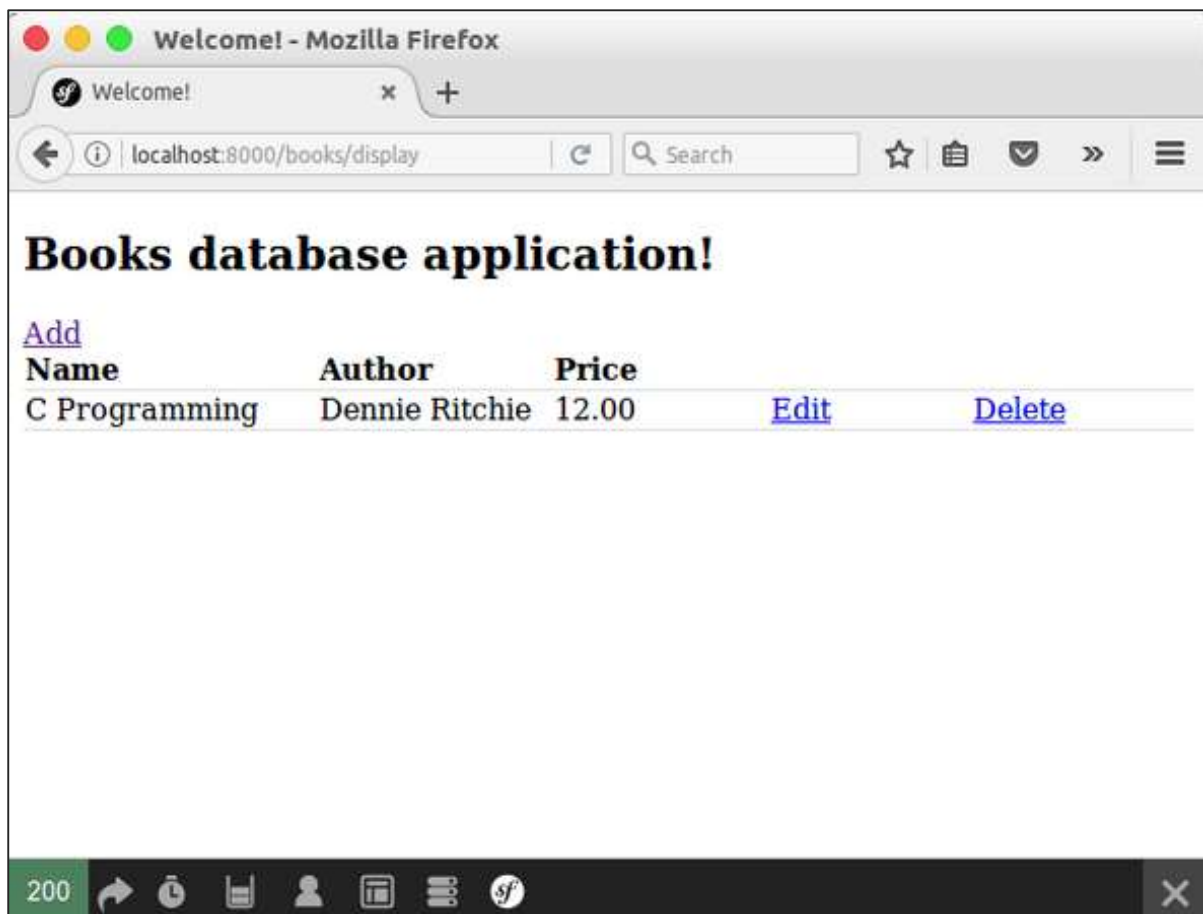
{% block body %}
<h2>Books database application!</h2>
<div>
    <a href="{{ path('app_book_new') }}">Add</a>
</div>
<table class="table">
    <tr>
        <th>Name</th>
        <th>Author</th>
        <th>Price</th>
        <th> </th>
        <th> </th>
    </tr>
{% for x in data %}
    <tr>
        <td>{{ x.Name }}</td>
        <td>{{ x.Author }}</td>
        <td>{{ x.Price }}</td>

        <td><a href="{{ path('app_book_update', { 'id' : x.Id }) }}">Edit</a></td>
        <td><a href="{{ path('app_book_delete', { 'id' : x.Id }) }}">Delete</a></td>
    </tr>
{% endfor %}
</table>

{% endblock %}

```

It will produce the following screen as output:



Symfony comprises of a set of PHP components, an application framework, a community and a philosophy. Symfony is extremely flexible and capable of meeting all the requirements of advanced users, professionals, and an ideal choice for all the beginners with PHP.