FREE eBook

LEARNING symfony3

Free unaffiliated eBook created from
Stack Overflow contributors.

#symfony3

Table of Contents

About ... 1

Chapter 1: Getting started with symfony3 ... 2

Remarks ... 2

Versions ... 2

Examples ... 2

3. Windows Systems ... 2

4. Creating the Symfony Application ... 3

1. Installing the Symfony Installer ... 3

5. Basing your Project on a Specific Symfony Version ... 3

2. Linux and Mac OS X Systems ... 4

Simplest example in Symfony ... 4

Creating page ... 5

Chapter 2: Asset Management with Assetic ... 6

Introduction ... 6

Parameters ... 6

Remarks ... 6

Examples ... 6

Create relative path for asset ... 6

Create absolute path for asset ... 6

Chapter 3: Configuration ... 7

Introduction ... 7

Examples ... 7

Include all configuration files from a directory ... 7

Use fully qualified class name (FQCN) as service id ... 7

No HTTP interface needed ? ... 8

Chapter 4: Declaring Entities ... 10

Examples ... 10

Declaring a Symfony Entity as YAML ... 10

Declaring a Symfony Entity with annotations ... 12

Chapter 5: Dynamic Forms ... 14

Examples ... 14
How to extend ChoiceType, EntityType and DocumentType to load choices with AJAX ... 14
Populate a select field depending on the value another ... 15
Chapter 6: Event Dispatcher ... 18
Syntax ... 18
Remarks ... 18
Examples ... 18
Event Dispatcher Quick Start ... 18
Event Subscribers ... 18
Chapter 7: Routing ... 20
Introduction ... 20
Remarks ... 20
Examples ... 20
Routing using YAML ... 20
Routing using annotations ... 21
Powerful Rest Routes ... 22
Chapter 8: Testing ... 24
Examples ... 24
Simple Testing in Symfony3 ... 24
Chapter 9: Validation ... 27
Remarks ... 27
Examples ... 27
Symfony validation using annotations ... 27
Symfony validation using YAML ... 31
Chapter 10: Working with Web Services ... 37
Examples ... 37
Rest API ... 37
Credits ... 42

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: symfony3

It is an unofficial and free symfony3 ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official symfony3.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with symfony3

Remarks

This section provides an overview of what symfony3 is, and why a developer might want to use it.

It should also mention any large subjects within symfony3, and link out to the related topics. Since the Documentation for symfony3 is new, you may need to create initial versions of those related topics.

Versions

Version	Release Date
3.0.0	2015-11-30
3.1.0	2016-05-30
3.2.0	2016-11-30
3.2.5	2017-03-09
3.2.6	2017-03-10
3.2.7	2017-04-05

Examples

3. Windows Systems

You must add php to your path environment variable. Follow theses steps :

Windows 7 :

- Right-click on a My Computer icon
- Click Properties
- Click Advanced system settings from the left nav
- Click Advanced tab
- Click Environment Variables button
- In the System Variables section, select Path (case-insensitive) and click Edit button
- Add a semi-colon (;) to the end of the string, then add the full file system path of your PHP installation (e.g. `C:\Program Files\PHP`)
- Keep clicking OK etc until all dialog boxes have disappeared
- Close your command prompt and open it again
- Sorted

Windows 8 & 10

- In Search, search for and then select: System (Control Panel)
- Click the Advanced system settings link.
- Click Environment Variables.
- In the section System Variables, find the PATH environment variable and select it. Click Edit. If the PATH environment variable does not exist, click New.
- Add the full file system path of your PHP installation (e.g. `C:\Program Files\PHP`)

After this, open your command console and execute the following command:

```
c:\> php -r "readfile('https://symfony.com/installer');" > symfony
```

Then, move the downloaded symfony file to your project's directory and execute it as follows:

```
c:\> move symfony c:\projects
c:\projects\> php symfony
```

4. Creating the Symfony Application

Once the Symfony Installer is available, create your first Symfony application with the new command:

```
# Linux, Mac OS X
$ symfony new my_project_name

# Windows
c:\> cd projects/
c:\projects\> php symfony new my_project_name
```

This command can be run from anywhere, not necessarily from the `htdocs` folder.

This command creates a new directory called `my_project_name/` that contains a fresh new project based on the most recent stable Symfony version available. In addition, the installer checks if your system meets the technical requirements to execute Symfony applications. If not, you'll see the list of changes needed to meet those requirements.

1. Installing the Symfony Installer

> The installer requires PHP 5.4 or higher. If you still use the legacy PHP 5.3 version, you cannot use the Symfony Installer. Read the Creating Symfony Applications without the Installer section to learn how to proceed. - source: http://symfony.com/doc/current/book/installation.html

5. Basing your Project on a Specific Symfony Version

In case your project needs to be based on a specific Symfony version, use the optional second argument of the new command:

```
# use the most recent version in any Symfony branch
$ symfony new my_project_name 2.8
$ symfony new my_project_name 3.1

# use a specific Symfony version
$ symfony new my_project_name 2.8.1
$ symfony new my_project_name 3.0.2

# use a beta or RC version (useful for testing new Symfony versions)
$ symfony new my_project 3.0.0-BETA1
$ symfony new my_project 3.1.0-RC1
```

The installer also supports a special version called lts which installs the most recent Symfony LTS version available:

```
$ symfony new my_project_name lts
```

Read the Symfony Release process to better understand why there are several Symfony versions and which one to use for your projects.

You can also create symfony applications without the installer, but it was not an good idea. If you want anyway, follow the original tutorial on this link:

Oficial Symfony Docs, Configuring Symfony without the installer

2. Linux and Mac OS X Systems

Open your command console and execute the following commands:

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

Simplest example in Symfony

1. Install symfony correctly as guided above.
2. Start symfony server if you are not installed in www directory.
3. Ensure http://localhost:8000 is working if symfony server is used.
4. Now it is ready to play with simplest example.
5. Add following code in a new file /src/AppBundle/Controller/MyController.php in symfony installation dir.
6. Test the example by visiting http://localhost:8000/hello
7. That's all. Next: use twig to render the response.

```
<?php
// src/AppBundle/Controller/MyController.php

namespace AppBundle\Controller;

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

```
class MyController
{
    /**
     * @Route("/hello")
     */
    public function myHelloAction()
    {
        return new Response(
            '<html><body>
                   I\'m the response for request <b>/hello</b>
             </body></html>'
        );
    }
}
```

Creating page

> Before continuing, make sure you've read the Installation chapter and can access your new Symfony app in the browser.

Suppose you want to create a page - /lucky/number - that generates a lucky (well, random) number and prints it. To do that, create a "Controller class" and a "controller" method inside of it that will be executed when someone goes to /lucky/number

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

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

class LuckyController
{
    /**
     * @Route("/lucky/number")
     */
    public function numberAction()
    {
        $number = rand(0, 100);

        return new Response(
            '<html><body>Lucky number: '.$number.'</body></html>'
        );
    }
}
```

Read Getting started with symfony3 online: https://riptutorial.com/symfony3/topic/985/getting-started-with-symfony3

Chapter 2: Asset Management with Assetic

Introduction

When using the Assetic Bundle, according to the Symfony documentation, please be aware of the following:

Starting from Symfony 2.8, Assetic is no longer included by default in the Symfony Standard Edition. Before using any of its features, install the AsseticBundle executing this console command in your project:

$ composer require symfony/assetic-bundle

There are other steps you have to take. For more information go to:
http://symfony.com/doc/current/assetic/asset_management.html

Parameters

Name	Example
Path	'static/images/logo/logo-default.png'

Remarks

The folder for the publicly accessible assets in a standard Symfony3 project is "/web". Assetic uses this folder as root folder for the assets.

Examples

Create relative path for asset

```
<img src="{{ asset('static/images/logo-default.png') }}" alt="Logo"/>

<!--Generates path for the file "/web/static/images/logo-default.png" -->
```

Create absolute path for asset

```
<img src="{{ app.request.getSchemeAndHttpHost() ~ asset('static/images/logo-default.png') }}"
alt="Logo"/>

<!--Generates path for the file "/web/static/images/logo-default.png" -->
```

Read Asset Management with Assetic online: https://riptutorial.com/symfony3/topic/6409/asset-management-with-assetic

Chapter 3: Configuration

Introduction

Examples and good practices for configuring your Symfony application that aren't in the official documentation.

Examples

Include all configuration files from a directory

After a while, you end up with many configuration items in your config.yml. It can make you configuration easier to read if you split your configuration across multiple files. You can easily include all files from a directory this way:

config.yml:

```
imports:
    - { resource: parameters.yml }
    - { resource: "includes/" }
```

In the `includes` directory you can put for example doctrine.yml, swiftmailer.yml, etc.

Use fully qualified class name (FQCN) as service id

In many examples, you will find a service id like 'acme.demo.service.id' (a string with dots). You `services.yml` will look like this:

```
services:
    acme.demo.service.id:
        class: Acme\DemoBundle\Services\DemoService
        arguments: ["@doctrine.orm.default_entity_manager", "@cache"]
```

In your controller, you can use this service:

```
$service = $this->get('acme.demo.service.id');
```

While there is no issue with this, you can use a Fully Qualified Class Name (FQCN) as service id:

```
services:
    Acme\DemoBundle\Services\DemoService:
        class: Acme\DemoBundle\Services\DemoService
        arguments: ["@doctrine.orm.default_entity_manager", "@cache"]
```

In your controller you can use it like this:

```
use Acme\DemoBundle\Services\DemoService;
// ..
$this->get(DemoService::class);
```

This makes your code better to understand. In many cases it makes no sense to have a service id that isn't just the class name.

As of Symfony 3.3, you can even remove the `class` attribute if you service id is a FQCN.

No HTTP interface needed ?

If your application does not need any HTTP interface (for example for a console only app), you will want to disable at least `Twig` and `SensioFrameworkExtra`

Just comment out those lines:

app/AppKernel.php

```
$bundles = [
//...
//    new Symfony\Bundle\TwigBundle\TwigBundle(),
//    new Sensio\Bundle\FrameworkExtraBundle\SensioFrameworkExtraBundle(),
//...
if (in_array($this->getEnvironment(), ['dev', 'test'], true)) {
//...
//    $bundles[] = new Symfony\Bundle\WebProfilerBundle\WebProfilerBundle();
```

app/config/config.yml

```
framework:
#    ...
#    router:
#        resource: '%kernel.root_dir%/config/routing.yml'
#        strict_requirements: ~
#    ...
#    templating:
#        engines: ['twig']
#...
#twig:
#    debug: '%kernel.debug%'
#    strict_variables: '%kernel.debug%'
```

app/config/config_dev.yml

```
#framework:
#    router:
#        resource: '%kernel.root_dir%/config/routing_dev.yml'
#        strict_requirements: true
#    profiler: { only_exceptions: false }

#web_profiler:
#    toolbar: true
#    intercept_redirects: false
```

You can also remove the related vendor requirements from **composer.json**:

```
"sensio/framework-extra-bundle": "x.x.x",
"twig/twig": "x.x"
```

Using Symfony at all in such case is arguable, but at least it can be temporary.

Read Configuration online: https://riptutorial.com/symfony3/topic/9175/configuration

Chapter 4: Declaring Entities

Examples

Declaring a Symfony Entity as YAML

- AppBundle/Entity/Person.php

```
<?php

namespace AppBundle\Entity;

/**
 * Person
 */
class Person
{
    /**
     * @var int
     */
    private $id;

    /**
     * @var string
     */
    private $name;

    /**
     * @var int
     */
    private $age;


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

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

        return $this;
    }
```

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

    /**
     * Set age
     *
     * @param integer $age
     *
     * @return Person
     */
    public function setAge($age)
    {
        $this->age = $age;

        return $this;
    }

    /**
     * Get age
     *
     * @return int
     */
    public function getAge()
    {
        return $this->age;
    }
}
```

- AppBundle/Resources/config/doctrine/Person.orm.yml

```
AppBundle\Entity\Person:
    type: entity
    repositoryClass: AppBundle\Repository\PersonRepository
    table: persons
    id:
        id:
            type: integer
            id: true
            generator:
                strategy: AUTO
    fields:
        name:
            type: string
            length: 20
            nullable: false
            column: Name
            unique: true
        age:
            type: integer
            nullable: false
            column: Age
            unique: false
```

```
    lifecycleCallbacks: { }
```

- Create the table

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

Or use the recommended way (assuming you already have doctrine-migrations-bundle installed in your project):

```
php bin/console doctrine:migrations:diff
php bin/console doctrine:migrations:migrate
```

- Create the entity automatically from the YAML

  php bin/console doctrine:generate:entities AppBundle

Declaring a Symfony Entity with annotations

- AppBundle/Entity/Person.php

```
<?php

namespace AppBundle\Entity;

use DoctrineORM\Mapping as ORM;
use Symfony\Bridge\Doctrine\Validator\Constraints\UniqueEntity;

/**
 * @ORM\Entity
 * @ORM\Table(name="persons")
 * @ORM\Entity(repositoryClass="AppBundle\Entity\PersonRepository")
 * @UniqueEntity("name")
 */
class Person
{
    /**
     * @ORM\Id
     * @ORM\Column(type="integer")
     * @ORM\GeneratedValue(strategy="AUTO")
     */
    protected $id;

    /**
     * @ORM\Column(type="string", name="name", length=20, nullable=false)
     */
    protected $name;

    /**
     * @ORM\Column(type="integer", name="age", nullable=false)
     */
    protected $age;

    public function getId()
    {
        return $this->id;
```

```
    }

    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;
    }
}
```

- Generate the entity

```
php bin/console doctrine:generate:entities AppBundle/Person
```

- To dump the SQL statements to the screen

```
php bin/console doctrine:schema:update --dump-sql
```

- Create the table

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

Or use the recommended way (assuming you already have doctrine-migrations-bundle installed in your project):

```
php bin/console doctrine:migrations:diff
php bin/console doctrine:migrations:migrate
```

Read Declaring Entities online: https://riptutorial.com/symfony3/topic/4443/declaring-entities

Chapter 5: Dynamic Forms

Examples

How to extend ChoiceType, EntityType and DocumentType to load choices with AJAX.

In Symfony, the built-in ChoiceType (and EntityType or DocumentType extending it), basicaly work with a constant choice list.

If you want to make it work with ajax calls, you have to change them to accept any sumitted extra choices.

- **How to start with an empty choice list ?**

  When you build your form, just set the `choices` option to an empty `array()` :

  ```
  namespace AppBundle\Form;

  use Symfony\Component\Form\AbstractType;

  use Symfony\Component\Form\FormBuilderInterface;

  use Symfony\Component\Form\Extension\Core\Type\ChoiceType;

  class FooType extends AbstractType
  {
      public function buildForm(FormBuilderInterface $builder, array $options)
      {
          $builder
              ->add('tag', ChoiceType::class, array('choices'=>array()));
      }
  }
  ```

  So you will get an empty select input, without choices. This solution works for ChoiceType and all of his children (EntityType, DocumentType, ...).

- **How to accept submitted new choices** :

  To accept the new choices, you have to make them available in the form field choicelist. You can change your form field depending on submitted data with the FormEvent::PRE_SUBMIT event.

  This example show how to do it with a basic ChoiceType :

  ```
  namespace AppBundle\Form;

  use Symfony\Component\Form\AbstractType;

  use Symfony\Component\Form\FormBuilderInterface;
  ```

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

class FooType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('tag', ChoiceType::class, array('choices'=>array()))
        ;

        $builder->addEventListener(
            FormEvents::PRE_SUBMIT,
            function(FormEvent $event){
                // Get the parent form
                $form = $event->getForm();

                // Get the data for the choice field
                $data = $event->getData()['tag'];

                // Collect the new choices
                $choices = array();

                if(is_array($data)){
                    foreach($data as $choice){
                        $choices[$choice] = $choice;
                    }
                }
                else{
                    $choices[$data] = $data;
                }

                // Add the field again, with the new choices :
                $form->add('tag', ChoiceType::class, array('choices'=>$choices));
            }
        );
    }
}
```

Your submitted choices are now allowed choices and Symfony ChoiceType built-in validation won't reject them anymore.

If you wan to do the same with a ChoiceType child (EntityType, DocumentType, ...), you have to inject the entityManager or the documentManager and to do the datatransformation when populating the new choices.

Populate a select field depending on the value another.

This is an example to show how to change the allowed choices on a subCategory select field depending on the value of the category select field. To do that you have to make your subCategory choices dynamical for both client and server side.

1. Make the form dynamic on the client side for display / user interactions

Example of client side dynamic form (using Javascript / JQuery) :

```
    $('#category').change(function(){
        switch($(this).val()){
            case '1': // If category == '1'
                var choice = {
                    'choice1_1':'1_1',
                    'choice1_2':'1_2',
                    'choice1_3':'1_3',
                };
            break;
            case '2': // If category == '2'
                var choice = {
                    'choice2_1':'2_1',
                    'choice2_2':'2_2',
                    'choice2_3':'2_3',
                };
            break;
            case '3': // If category == '3'
                var choice = {
                    'choice3_1':'3_1',
                    'choice3_2':'3_2',
                    'choice3_3':'3_3',
                };
            break;
        }

        var $subCategorySelect = $('#subCategory');

        $subCategorySelect.empty();
        $.each(choice, function(key, value) {
            $subCategorySelect.append($('<option></option>')).attr('value',value).text(key);
        });
    });
```

Of course you could get the choices from an AJAX call. That's not the purpose of this example.

2. Make the form dynamic on the server side for initialisation / validation

Example of server side dynamic form :

```
    namespace AppBundle\Form;

    use Symfony\Component\Form\AbstractType;
    use Symfony\Component\Form\FormBuilderInterface;

    use Symfony\Component\Form\Extension\Core\Type\ChoiceType;

    use Symfony\Component\Form\FormEvent;
    use Symfony\Component\Form\FormEvents;

    class MyBaseFormType extends AbstractType
    {
        /**
         * @param FormBuilderInterface $builder
         * @param array $options
         */
        public function buildForm(FormBuilderInterface $builder, array $options)
        {
            $builder
                ->add('category',ChoiceType::class,array('choices'=>array(
```

```
                    'choice1'=>'1',
                    'choice2'=>'2',
                    'choice3'=>'3',
                )))
            ;

            $addSubCategoryListener = function(FormEvent $event){
                $form = $event->getForm();
                $data = $event->getData();

                switch($data['category']){
                    case '1': // If category == '1'
                        $choices = array(
                            'choice1_1'=>'1_1',
                            'choice1_2'=>'1_2',
                            'choice1_3'=>'1_3',
                        );
                    break;
                    case '2': // If category == '2'
                        $choices = array(
                            'choice2_1'=>'2_1',
                            'choice2_2'=>'2_2',
                            'choice2_3'=>'2_3',
                        );
                    break;
                    case '3': // If category == '3'
                        $choices = array(
                            'choice3_1'=>'3_1',
                            'choice3_2'=>'3_2',
                            'choice3_3'=>'3_3',
                        );
                    break;
                }

                $form->add('subCategory',ChoiceType::class,array('choices'=>$choices));
            };

            // This listener will adapt the form with the data passed to the form during
construction :
            $builder->addEventListener(FormEvents::PRE_SET_DATA, $addSubCategoryListener);

            // This listener will adapt the form with the submitted data :
            $builder->addEventListener(FormEvents::PRE_SUBMIT, $addSubCategoryListener);
        }
    }
```

Read Dynamic Forms online: https://riptutorial.com/symfony3/topic/5855/dynamic-forms

Chapter 6: Event Dispatcher

Syntax

- $dispatcher->dispatch(string $eventName, Event $event);
- $dispatcher->addListener(string $eventName, callable $listener, int $priority = 0);
- $dispatcher->addSubscriber(EventSubscriberInterface $subscriber);

Remarks

- It is often best to use a single instance of EventDispatcher in your application that you inject into the objects that need to fire events.
- It is best practice to have a single location where you manage the configuration of, and add event listeners to, your EventDispatcher. The Symfony framework uses the Dependency Injection Container.
- These patterns will allow you to easily change your event listeners without needing to change the code of any module that is dispatching events.
- The decoupling of event dispatch from event listener configuration is what makes the Symfony EventDispatcher so powerful
- The EventDispatcher helps you satisfy the Open/Closed Principle.

Examples

Event Dispatcher Quick Start

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

// you may store this in a dependency injection container for use as a service
$dispatcher = new EventDispatcher();

// you can attach listeners to specific events directly with any callable
$dispatcher->addListener('an.event.occurred', function(Event $event) {
    // process $event
});

// somewhere in your system, an event happens
$data = // some important object
$event = new GenericEvent($data, ['more' => 'event information']);

// dispatch the event
// our listener on "an.event.occurred" above will be called with $event
// we could attach many more listeners to this event, and they too would be called
$dispatcher->dispatch('an.event.occurred', $event);
```

Event Subscribers

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

$dispatcher = new EventDispatcher();

// you can attach event subscribers, which allow a single object to subscribe
// to many events at once
$dispatcher->addSubscriber(new class implements EventSubscriberInterface {
    public static function getSubscribedEvents()
    {
        // here we subscribe our class methods to listen to various events
        return [
            // when anything fires a "an.event.occurred" call "onEventOccurred"
            'an.event.occurred' => 'onEventOccurred',
            // an array of listeners subscribes multiple methods to one event
            'another.event.happened' => ['whenAnotherHappened', 'sendEmail'],
        ];
    }

    function onEventOccurred(Event $event) {
        // process $event
    }

    function whenAnotherHappened(Event $event) {
        // process $event
    }

    function sendEmail(Event $event) {
        // process $event
    }
});
```

Read Event Dispatcher online: https://riptutorial.com/symfony3/topic/5609/event-dispatcher

Chapter 7: Routing

Introduction

A route is like mapping a URL to an action (function) in a Controller class. The following topic will focus on creating routes, passing parameters to the Controller class via a route either using YAML or annotation.

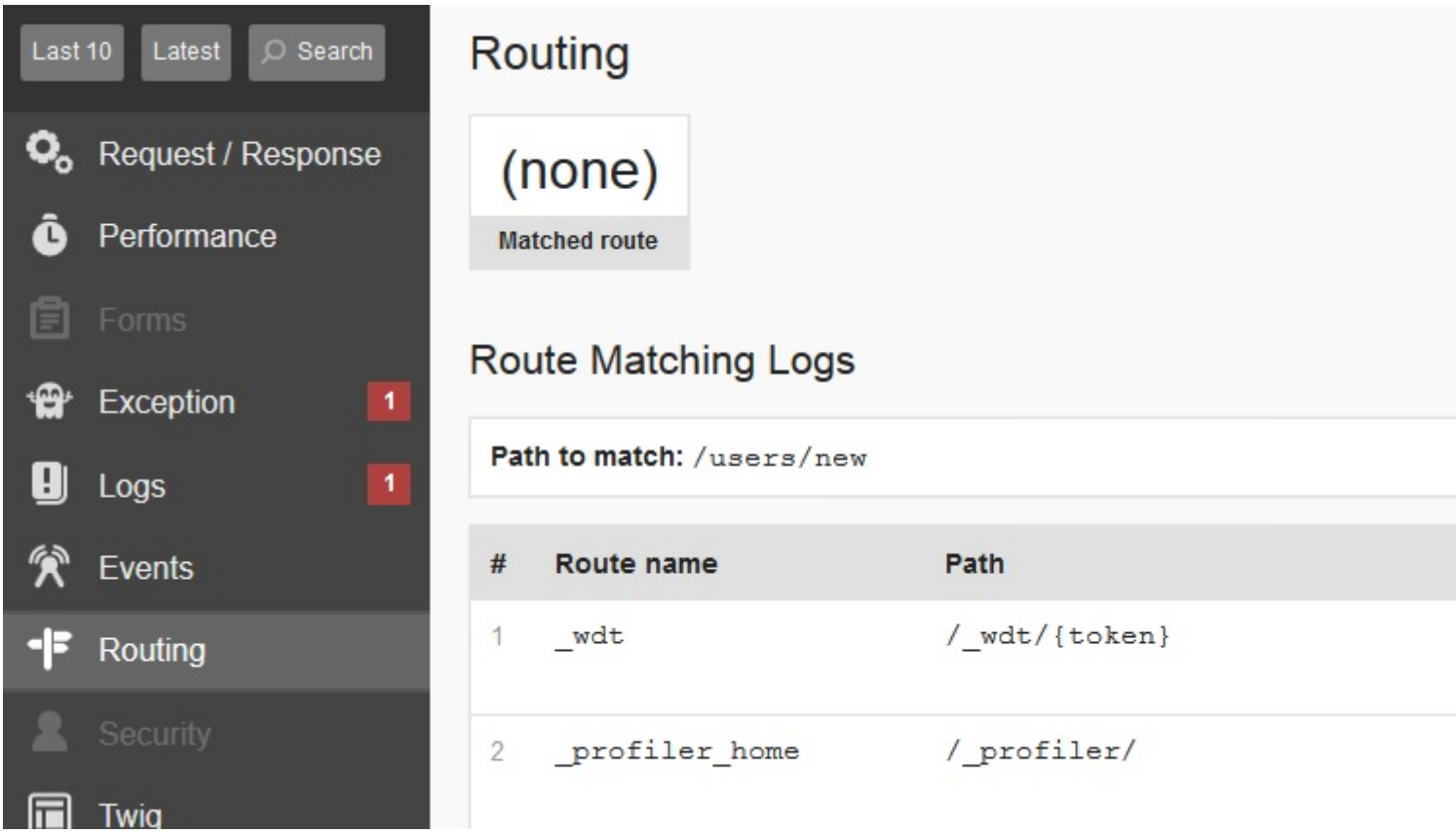
Remarks

It's useful to see what is generated by Symfony framework, this one provide tools to watch all routes of an specific application.

From the Symfony Doc, use (in a shell) :

```
php bin/console debug:router
```

As well, you can watch all the relevant routes informations in the Framework profiler, in the routing menu :



Examples

Routing using YAML

The routing configuration is included in your `app/config/config.yml` file, by default the `app/config/routing.yml` file.

From there you can link to your own routing configuration in a bundle

```
# app/config/routing.yml

app:
    resource: "@AppBundle/Resources/config/routing.yml"
```

It might also contain several application global routes.

In your own bundle you can configure, routes which serve two purposes:

- Matching against a request, such that the correct action is called for the request.
- Generating an URL from the name and route parameters.

Below is an example YAML route configuration:

```
# src/AppBundle/Resources/config/routing.yml

my_page:
    path: /application/content/page/{parameter}
    defaults:
        _controller: AppBundle:Default:myPage
        parameter: 42
    requirements:
        parameter: '\d+'
    methods: [ GET, PUT ]
    condition: "request.headers.get('User-Agent') matches '/firefox/i'"
```

The route is named `my_page`, and calls the `myPageAction` of the `DefaultController` in the `AppBundle` when requested. It has a parameter, named `parameter` with a default value. The value is only valid when it matches the regex `\d+`. For this route, only the HTTP methods `GET` and `PUT` are accepted. The `condition` is an expression in the example the route won't match unless the User-Agent header matches firefox. You can do any complex logic you need in the expression by leveraging two variables that are passed into the expression: `context` (RequestContext) and `request` (Symfony Request).

A generated route with the parameter value of 10 might look like `/application/content/page/10`.

Routing using annotations

The routing configuration is included in your `app/config/config.yml` file, by default the `app/config/routing.yml` file.

From there you can link to the controllers that have annotated routing configuration:

```
# app/config/routing.yml

app:
    resource: "@AppBundle/Controller"
```

```
    type:     annotation
```

In your own bundle you can configure, routes which serve two purposes:

- Matching against a request, such that the correct action is called for the request.
- Generating an URL from the name and route parameters.

Below is an example annotated route configuration:

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

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

/**
 * @Route("/application")
 */
class DefaultController extends Controller {
    /**
     * @Route("/content/page/{parameter}",
     *        name="my_page",
     *        requirements={"parameter" = "\d+"},
     *        defaults={"parameter" = 42})
     * @Method({"GET", "PUT"})
     */
    public function myPageAction($parameter)
    {
        // ...
    }
}
```

The controller is annotated with a *prefix* route, such that any configured route in this controller will be prepended using the prefix.

The configured route is named `my_page`, and calls the `myPageAction` function when requested. It has a parameter, named `parameter` with a default value. The value is only valid when it matches the regex `\d+`. For this route, only the HTTP methods `GET` and `PUT` are accepted.

Notice that the parameter is injected into the action as a function parameter.

A generated route with the parameter value of 10 might look like `/application/content/page/10`.

Powerful Rest Routes

Traditionally, you can use routing to map an the request with the Routing Component who handled request and response parameters by HttpFoundation Component.

Additionally, a custom route parameter can be created by using the FOSRestBundle to extends the default functionalities of the Routing Component.

This is useful for creating REST routes, an really useful to specify how to tranfer structured data like XML or json file in a request (and a response).

See FOSRestBundle Doc for more information, and especially this annotation :

```
use FOS\RestBundle\Controller\Annotations\FileParam;

/**
 * @FileParam(
 *   name="",
 *   key=null,
 *   requirements={},
 *   default=null,
 *   description="",
 *   strict=true,
 *   nullable=false,
 *   image=false
 * )
 */
```

Read Routing online: https://riptutorial.com/symfony3/topic/2287/routing

Chapter 8: Testing

Examples

Simple Testing in Symfony3

Unit Test

Unit tests are used to ensure that your code has no syntax error and to test the logic of your code to work as what you expected. Quick example:

src/AppBundle/Calculator/BillCalculator.php

```
<?php

namespace AppBundle\Calculator;

use AppBundle\Calculator\TaxCalculator;

class BillCalculator
{
    private $taxCalculator;

    public function __construct(TaxCalculator $taxCalculator)
    {
        $this->taxCalculator = $taxCalculator;
    }

    public function calculate($products)
    {
        $totalPrice = 0;
        foreach ($products as $product) {
            $totalPrice += $product['price'];
        }
        $tax = $this->taxCalculator->calculate($totalPrice);

        return $totalPrice + $tax;
    }
}
```

src/AppBundle/Calculator/TaxCalculator.php

```
<?php

namespace AppBundle\Calculator;

class TaxCalculator
{
    public function calculate($price)
    {
        return $price * 0.1; // for example the tax is 10%
    }
}
```

tests/AppBundle/Calculator/BillCalculatorTest.php

```
<?php

namespace Tests\AppBundle\Calculator;

class BillCalculatorTest extends \PHPUnit_Framework_TestCase
{
    public function testCalculate()
    {
        $products = [
            [
                'name' => 'A',
                'price' => 100,
            ],
            [
                'name' => 'B',
                'price' => 200,
            ],
        ];
        $taxCalculator = $this->getMock(\AppBundle\Calculator\TaxCalculator::class);

        // I expect my BillCalculator to call $taxCalculator->calculate once
        // with 300 as the parameter
        $taxCalculator->expects($this->once())->method('calculate')->with(300)-
>willReturn(30);

        $billCalculator = new BillCalculator($taxCalculator);
        $price = $billCalculator->calculate($products);

        $this->assertEquals(330, $price);
    }
}
```

I tested my BillCalculator class so I can ensure that my BillCalculator will return total products price + 10% tax. In unit test, we create our own test case. In this test, I provide 2 products (the prices are 100 and 200), so the tax will be 10% = 30. I expect the TaxCalculator to return 30, so that the total price will be 300 + 30 = 330.

Functional Test

Functional tests are used to test the input and output. With the given input, I expected some output without testing the process to create the output. (this is different with unit test because in unit test, we test the code flow). Quick example:

```
namespace Tests\AppBundle;

use Symfony\Bundle\FrameworkBundle\Test\WebTestCase;

class ApplicationAvailabilityFunctionalTest extends WebTestCase
{
    /**
     * @dataProvider urlProvider
     */
    public function testPageIsSuccessful($url)
    {
        $client = self::createClient();
```

```
        $client->request('GET', $url);

        $this->assertTrue($client->getResponse()->isSuccessful());
    }

    public function urlProvider()
    {
        return array(
            array('/'),
            array('/posts'),
            array('/post/fixture-post-1'),
            array('/blog/category/fixture-category'),
            array('/archives'),
            // ...
        );
    }
}
```

I tested my controller so i can ensure that my controller will return 200 response instead of 400 (Not Found) or 500 (Internal Server Error) with the given url.

References:

- http://symfony.com/doc/current/best_practices/tests.html
- http://symfony.com/doc/current/book/testing.html

Read Testing online: https://riptutorial.com/symfony3/topic/2881/testing

Chapter 9: Validation

Remarks

In fact, form validation is based from a component, named "**Validator Component**".

You can often use the dedicated service if you did't have to show a form in a template. Like APIs. You may validate datas in the same way, like this :

For example, *based on symfony doc* :

```
$validator = $this->get('validator');
$errors = $validator->validate($author);

if (count($errors) > 0) {
    /*
     * Uses a __toString method on the $errors variable which is a
     * ConstraintViolationList object. This gives us a nice string
     * for debugging.
     */
    $errorsString = (string) $errors;
}
```

Examples

Symfony validation using annotations

- Enable validation using annotations in `app/config/config.yml` file

```
framework:
    validation: { enable_annotations: true }
```

- Create an Entity in `AppBundle/Entity` directory. The validations are made with `@Assert` annotations.

```
<?php
# AppBundle/Entity/Car.php


namespace AppBundle\Entity;


use Doctrine\ORM\Mapping as ORM;
use Symfony\Component\Validator\Constraints as Assert;

/**
 * Car
 *
 * @ORM\Table(name="cars")
 * @ORM\Entity(repositoryClass="AppBundle\Repository\CarRepository")
 */
```

```
class Car
{
    /**
     * @var int
     *
     * @ORM\Column(name="id", type="integer")
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="AUTO")
     */
    private $id;

    /**
     * @var string
     *
     * @ORM\Column(name="name", type="string", length=50)
     * @Assert\NotBlank(message="Please provide a name")
     * @Assert\Length(
     *     min=3,
     *     max=50,
     *     minMessage="The name must be at least 3 characters long",
     *     maxMessage="The name cannot be longer than 50 characters"
     * )
     * @Assert\Regex(
     *     pattern="/^[A-Za-z]+$/",
     *     message="Only letters allowed"
     * )
     */
    private $name;

    /**
     * @var string
     *
     * @ORM\Column(name="number", type="integer")
     * @Assert\NotBlank(message="Please provide a number")
     * @Assert\Length(
     *     min=1,
     *     max=3,
     *     minMessage="The number field must contain at least one number",
     *     maxMessage="The number field must contain maximum 3 numbers"
     * )
     * @Assert\Regex(
     *     pattern="/^[0-9]+$/",
     *     message="Only numbers allowed"
     * )
     */
    private $number;


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

    /**
     * Set name
     *
```

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

        return $this;
    }

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

    /**
     * Set number
     *
     * @param integer $number
     *
     * @return Car
     */
    public function setNumber($number)
    {
        $this->number = $number;

        return $this;
    }

    /**
     * Get number
     *
     * @return integer
     */
    public function getNumber()
    {
        return $this->number;
    }
}
```

- Create a new form in `AppBundle/Form` directory.

```
<?php
# AppBundle/Form/CarType.php

namespace AppBundle\Form;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolver;
use Symfony\Component\OptionsResolver\OptionsResolverInterface;
use Symfony\Component\Form\Extension\Core\Type\TextType;
use Symfony\Component\Form\Extension\Core\Type\IntegerType;
```

```
class CarType extends AbstractType
{
    /**
     * @param FormBuilderInterface $builder
     * @param array $options
     */
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('name', TextType::class, ['label'=>'Name'])
            ->add('number', IntegerType::class, ['label'=>'Number'])
        ;
    }

    /**
     * @param OptionsResolver $resolver
     */
    public function configureOptions(OptionsResolver $resolver)
    {
        $resolver->setDefaults(array(
            'data_class' => 'AppBundle\Entity\Car'
        ));
    }

    public function setDefaultOptions(OptionsResolverInterface $resolver)
    {
        // TODO: Implement setDefaultOptions() method.
    }

    public function getName()
    {
        return 'car_form';
    }
}
```

- Create a new route and a new action method in `AppBundle/Controller/DefaultController.php`. The route will be declared with annotations too, so make sure you've imported this route in the main route file (`app/config/routing.yml`).

```
<?php

namespace AppBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\Routing\Annotation\Route;
use AppBundle\Entity\Car;
use AppBundle\Form\CarType;

class DefaultController extends Controller
{
    /**
     * @Route("/car", name="app_car")
     */
    public function carAction(Request $request)
    {
        $car = new Car();

        $form = $this->createForm(
```

```
            CarType::class,
            $car,
            [
                'action' => $this->generateUrl('app_car'),
                'method'=>'POST',
                'attr'=>[
                    'id'=>'form_car',
                    'class'=>'car_form'
                ]
            ]
        );

        $form->handleRequest($request);

        return $this->render(
            'AppBundle:Default:car.html.twig',[
                'form'=>$form->createView()
            ]
        );
    }
}
```

- Create the view in `AppBundle/Resources/views/Default/car.html.twig`.

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

{% block body %}
    {{ form_start(form, {'attr': {'novalidate':'novalidate'}}) }}
    {{ form_row(form.name) }}
    {{ form_row(form.number) }}
    <button type="submit">Go</button>
    {{ form_end(form) }}
{% endblock %}
```

- Start Symfony's built-in server (`php bin/console server:run`) and access `127.0.0.1:8000/car` route in your browser. There should be a form consisting of two input boxes and a submit button. If you press the submit button without entering any data into the input boxes, then the error messages will be displayed.

Symfony validation using YAML

- Create an Entity in `AppBundle/Entity` directory. You can do this manually, or by using Symfony's command `php bin/console doctrine:generate:entity` and filling in the required information in each step. You must specify `yml` option at `Configuration format (yml, xml, php or annotation)` step.

```
<?php
# AppBundle/Entity/Person.php

namespace AppBundle\Entity;

/**
 * Person
 */
class Person
```

```
{
    /**
     * @var int
     */
    private $id;

    /**
     * @var string
     */
    private $name;

    /**
     * @var int
     */
    private $age;


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

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

        return $this;
    }

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

    /**
     * Set age
     *
     * @param integer $age
     *
     * @return Person
     */
    public function setAge($age)
    {
        $this->age = $age;
```

```
        return $this;
    }

    /**
     * Get age
     *
     * @return int
     */
    public function getAge()
    {
        return $this->age;
    }
}
```

- Create the Entity mapping information for the Entity class. If you are using Symfony's command `php bin/console doctrine:generate:entity`, then the following code will be auto-generated. Otherwise, if you don't use the command, you can create the following code by hand.

```
# AppBundle/Resources/config/doctrine/Person.orm.yml

AppBundle\Entity\Person:
type: entity
table: persons
repositoryClass: AppBundle\Repository\PersonRepository
id:
    id:
        type: integer
        id: true
        generator:
            strategy: AUTO
fields:
    name:
        type: string
        length: '50'
    age:
        type: integer
lifecycleCallbacks: {  }
```

- Create the validation for the Entity class.

```
# AppBundle/Resources/config/validation/person.yml

AppBundle\Entity\Person:
    properties:
        name:
            - NotBlank:
                message: "Name is required"
            - Length:
                min: 3
                max: 50
                minMessage: "Please use at least 3 chars"
                maxMessage: "Please use max 50 chars"
            - Regex:
                pattern: "/^[A-Za-z]+$/"
                message: "Please use only letters"
        age:
```

```
            - NotBlank:
                message: "Age is required"
            - Length:
                min: 1
                max: 3
                minMessage: "The age must have at least 1 number in length"
                maxMessage: "The age must have max 3 numbers in length"
            - Regex:
                pattern: "/^[0-9]+$/"
                message: "Please use only numbers"
```

- Create a new form in `AppBundle/Form` directory.

```
<?php
# AppBundle/Form/PersonType.php

namespace AppBundle\Form;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolver;
use Symfony\Component\OptionsResolver\OptionsResolverInterface;
use Symfony\Component\Form\Extension\Core\Type\TextType;
use Symfony\Component\Form\Extension\Core\Type\IntegerType;

class PersonType extends AbstractType
{
    /**
     * @param FormBuilderInterface $builder
     * @param array $options
     */
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('name', TextType::class, ['label'=>'Name'])
            ->add('age', IntegerType::class, ['label'=>'Age'])
        ;
    }

    /**
     * @param OptionsResolver $resolver
     */
    public function configureOptions(OptionsResolver $resolver)
    {
        $resolver->setDefaults(array(
            'data_class' => 'AppBundle\Entity\Person'
        ));
    }

    public function setDefaultOptions(OptionsResolverInterface $resolver)
    {
        // TODO: Implement setDefaultOptions() method.
    }

    public function getName()
    {
        return 'person_form';
    }
}
```

- Create a new route in `AppBundle/Resources/config/routing.yml`

```
app_person:
    path: /person
    defaults: { _controller: AppBundle:Default:person }
```

- Now create a new action method for that route.

```
<?php
# AppBundle/Controller/DefaultController.php

namespace AppBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Request;
use AppBundle\Entity\Person;
use AppBundle\Form\PersonType;

class DefaultController extends Controller
{
    public function personAction(Request $request)
    {
        $person = new Person();

        $form = $this->createForm(
            PersonType::class,
            $person,
            [
                'action' => $this->generateUrl('app_person'),
                'method'=>'POST',
                'attr'=>[
                    'id'=>'form_person',
                    'class'=>'person_form'
                ]
            ]
        );

        $form->handleRequest($request);

        return $this->render(
            'AppBundle:Default:person.html.twig', [
                'form'=>$form->createView()
            ]
        );
    }
}
```

- Create the view in `AppBundle/Resources/views/Default/person.html.twig`

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

{% block body %}
    {{ form_start(form, {'attr': {'novalidate':'novalidate'}}) }}
        {{ form_row(form.name) }}
        {{ form_row(form.age) }}
        <button type="submit">Go</button>
    {{ form_end(form) }}
{% endblock %}
```

- Start Symfony's built-in server (`php bin/console server:run`) and access `127.0.0.1:8000/person` route in your browser. There should be a form consisting of two input boxes and a submit button. If you press the submit button without entering any data into the input boxes, then the error messages will be displayed.

Read Validation online: https://riptutorial.com/symfony3/topic/6457/validation

Chapter 10: Working with Web Services

Examples

Rest API

I have previously written documentation on this site in order to describe how to make web services on Symfony

I will write again a tutorial for the symfony >= 3 version.

We think that we have a installed web-server on a configured version of Symfony Framework. You must have composer (php packages manager) installed too.

To made it simple, if you have composer installed, type this in a terminal / command prompt :

```
composer create-project symfony/framework-standard-edition example "3.1.*"
```

This will create a new directory called "example" in the current directory, with a standard installation of symfony framework.

You must install this 2 Bundles : JMSSerializer Bundle (extends framework component serializer) and FOSRest Bundle (extends framework component routing and controllers...)

You can do this like this (in the example directory) :

```
composer require jms/serializer-bundle "~0.13"
composer require friendsofsymfony/rest-bundle
```

Don't forget to activate them in AppKernel !

Here you can't use :

```
composer create-project gimler/symfony-rest-edition --stability=dev example
```

Because it's based on Symfony 2.8 version.

First, create your own ("Example") Bundle (in Symfony directory) :

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

Imagine that we want to make CRUD (Create / Read / Update / Delete) of this StackOverFlower Entity :

```
# src/ExampleBundle/Resources/config/doctrine/StackOverFlower.orm.yml
ExampleBundle\Entity\StackOverFlower:
```

```
    type: entity
    table: stackoverflower
    id:
        id:
            type: integer
            generator: { strategy: AUTO }
    fields:
        name:
            type: string
            length: 100
```

Configure your Bundle :

```
#app/config/config.yml
fos_rest:
    format_listener:
        rules:
            - { path: '^/stackoverflower', priorities: ['xml', 'json'], fallback_format: xml,
prefer_extension: true }
            - { path: '^/', priorities: [ 'text/html', '*/*'], fallback_format: html,
prefer_extension: true }
```

Generate this entity :

```
php bin/console doctrine:generate:entity StackOverFlower
php bin/console doctrine:schema:update --force
```

Make a Controller :

```
#src/ExampleBundle/Controller/StackOverFlowerController.php

namespace ExampleBundle\Controller;

use FOS\RestBundle\Controller\FOSRestController;
use Symfony\Component\HttpFoundation\Request;

use FOS\RestBundle\Controller\Annotations\Get;
use FOS\RestBundle\Controller\Annotations\Post;
use FOS\RestBundle\Controller\Annotations\Delete;

use ExampleBundle\Entity\StackOverFlower;

class StackOverFlowerController extends FOSRestController
{
    /**
     * findStackOverFlowerByRequest
     *
     * @param Request $request
     * @return StackOverFlower
     * @throws NotFoundException
     */
    private function findStackOverFlowerByRequest(Request $request) {

        $id = $request->get('id');
        $user = $this->getDoctrine()->getManager()-
>getRepository("ExampleBundle:StackOverFlower")->findOneBy(array('id' => $id));
```

```
        return $user;
    }

    /**
     * validateAndPersistEntity
     *
     * @param StackOverFlower $user
     * @param Boolean $delete
     * @return View the view
     */
    private function validateAndPersistEntity(StackOverFlower $user, $delete = false) {

        $template = "ExampleBundle:StackOverFlower:example.html.twig";

        $validator = $this->get('validator');
        $errors_list = $validator->validate($user);

        if (0 === count($errors_list)) {

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

            if ($delete === true) {
                $em->remove($user);
            } else {
                $em->persist($user);
            }

            $em->flush();

            $view = $this->view($user)
                        ->setTemplateVar('user')
                        ->setTemplate($template);
        } else {

            $errors = "";
            foreach ($errors_list as $error) {
                $errors .= (string) $error->getMessage();
            }

            $view = $this->view($errors)
                        ->setTemplateVar('errors')
                        ->setTemplate($template);

        }

        return $view;
    }

    /**
     * newStackOverFlowerAction
     *
     * @Get("/stackoverflower/new/{name}")
     *
     * @param Request $request
     * @return String
     */
    public function newStackOverFlowerAction(Request $request)
    {
        $user = new StackOverFlower();
        $user->setName($request->get('name'));
```

```
        $view = $this->validateAndPersistEntity($user);

        return $this->handleView($view);
    }

    /**
     * editStackOverFlowerAction
     *
     * @Get("/stackoverflower/edit/{id}/{name}")
     *
     * @param Request $request
     * @return type
     */
    public function editStackOverFlowerAction(Request $request) {

        $user = $this->findStackOverFlowerByRequest($request);

        if (! $user) {
            $view = $this->view("No StackOverFlower found for this id:". $request->get('id'),
404);
            return $this->handleView($view);
        }

        $user->setName($request->get('name'));

        $view = $this->validateAndPersistEntity($user);

        return $this->handleView($view);
    }

    /**
     * deleteStackOverFlowerAction
     *
     * @Get("/stackoverflower/delete/{id}")
     *
     * @param Request $request
     * @return type
     */
    public function deleteStackOverFlowerAction(Request $request) {

        $user = $this->findStackOverFlowerByRequest($request);

        if (! $user) {
            $view = $this->view("No StackOverFlower found for this id:". $request->get('id'),
404);
            return $this->handleView();
        }

        $view = $this->validateAndPersistEntity($user, true);

        return $this->handleView($view);
    }

    /**
     * getStackOverFlowerAction
     *
     * @Get("/stackoverflowers")
     *
     * @param Request $request
     * @return type
     */
```

```
    public function getStackOverFlowerAction(Request $request) {

        $template = "ExampleBundle:StackOverFlower:example.html.twig";

        $users = $this->getDoctrine()->getManager()-
>getRepository("ExampleBundle:StackOverFlower")->findAll();

        if (0 === count($users)) {
            $view = $this->view("No StackOverFlower found.", 404);
            return $this->handleView();
        }

        $view = $this->view($users)
                    ->setTemplateVar('users')
                    ->setTemplate($template);

        return $this->handleView($view);
    }
}
```

Don't tell me that is a fat controller, it's for the example !!!

Create your template :

```
#src/ExampleBundle/Resources/views/StackOverFlower.html.twig
{% if errors is defined %}
  {{ errors }}
{% else %}
  {% if users is defined %}
    {{ users | serialize }}
  {% else %}
    {{ user | serialize }}
  {% endif %}
{% endif %}
```

You have just made your first RESTFul API !!!

You can test it on : http://your-server-name/your-symfony-path/app_dev.php/stackoverflower/new/test.

As you can see in the database, a new user has been created with the name : "test".

You can view a full working example of this code on my GitHub Account, one branch with more real routes...

This is a very basic example, don't let that in production environnement, you must protect your api with apikey !!!

A future example, may be ?

Read Working with Web Services online: https://riptutorial.com/symfony3/topic/6972/working-with-web-services

Credits

S. No	Chapters	Contributors
1	Getting started with symfony3	Community, insertusernamehere, Paweł Kolanowski, Raphael Schubert, Tartare2240, TRiNE, uddhab, YoannFleuryDev
2	Asset Management with Assetic	Aaron Belchamber, Orlando
3	Configuration	Pierre de LESPINAY, Stephan Vierkant
4	Declaring Entities	Dan Costinel, janek1, Nicodemuz, rubenj
5	Dynamic Forms	Alsatian
6	Event Dispatcher	Chris Tickner
7	Routing	Alvin Bunk, Anjana Silva, Hidde, Mathieu Dormeval, Matteo, Renato Mefi
8	Testing	Hendra Huang, Pierre de LESPINAY
9	Validation	Dan Costinel, Mathieu Dormeval
10	Working with Web Services	Mathieu Dormeval, Pascal, Pierre de LESPINAY