**MANUAL DE INTRODUCCIÓN A SYMFONY**

1. **Descargar e Instalar composer**

https://getcomposer.org/download/

**Link php composer.phar to composer**

mv composer.phar /usr/local/bin/composer

1. **CREAR UN NUEVO PROYECTO DE SYMFONY (http://symfony.com/doc/current/setup.html)**

composer create-project symfony/framework-standard-edition NOMBRE\_PROYECTO

1. **CORRER LA APP**

php bin/console server:run

php bin/console server:start

1. **PROBLEMAS DE VULNERABILIDAD**

php bin/console security:check

1. **ACTUALIZAR EL PROYECTO Y DEPENDENCIAS**

composer update

1. **AÑADIR NUEVAS DEPENDENCIAS AL PROYECTO**

composer require doctrine/doctrine-fixtures-bundle

1. **SUBIR EL PROYECTO A GITHUB**

Crear el repositorio en github.com

git init

git remote add origin REPO\_URL

git add -A

git commit -m "inicio proyecto"

git config --global push.default simple

git push --set-upstream origin master

1. **CLONAR E INSTALAR EL PROYECTO**

En el servidor remoto:

git clone REPO\_URL

cd my\_symfony

composer install

1. **CREAR LA PRIMERA PÁGINA (LUCKY NUMBER)**

Crear ruta y controlador

// 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 = mt\_rand(0, 100);

return new Response(

'<html><body>Lucky number: '.$number.'</body></html>'

);

}

}

1. **CORRER LA APP E IR A http://localhost:8000/app\_dev.php/lucky/number**

SI SE USA EL WEB SERVER DE SYMFONY SE OMITE EL app-Dev http://localhost:8000/lucky/number

1. **BORRAR CACHE**

php bin/console cache:clear [--no-warmup] [--no-optional-warmers]

1. **THE WEB DEBUG TOOLBAR**

**CREAR EL TEMPLATE**

Añadir estas lineas al controlador

// --> add this new use statement

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

La Clase debe extends Controller

Crear a new app/Resources/views/lucky directory

con el archivo number.html.twig:

{# app/Resources/views/lucky/number.html.twig #}

<h1>Your lucky number is {{ number }}</h1>

**CHECKING OUT THE PROJECT STRUCTURE**

app/

Contains things like configuration and templates. Basically, anything that is not PHP code goes here.

src/

Your PHP code lives here.

99% of the time, you'll be working in src/ (PHP files) or app/ (everything else). As you keep reading, you'll learn what can be done inside each of these.

So what about the other directories in the project?

bin/

The famous bin/console file lives here (and other, less important executable files).

tests/

The automated tests (e.g. Unit tests) for your application live here.

var/

This is where automatically-created files are stored, like cache files (var/cache/), logs (var/logs/) and sessions (var/sessions/).

vendor/

Third-party (i.e. "vendor") libraries live here! These are downloaded via the Composer package manager.

web/

This is the document root for your project: put any publicly accessible files here (e.g. CSS, JS and images).

Bundles & Configuration[¶](http://symfony.com/doc/current/page_creation.html#bundles-configuration)

* Your Symfony application comes pre-installed with a collection of *bundles*, like FrameworkBundleand TwigBundle. Bundles are similar to the idea of a *plugin*, but with one important difference: *all* functionality in a Symfony application comes from a bundle.
* Bundles are registered in your app/AppKernel.php file (a rare PHP file in the app/ directory) and each gives you more *tools*, sometimes called *services*:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | class AppKernel extends Kernel  {  public function registerBundles()  {  $bundles = array(  new Symfony\Bundle\FrameworkBundle\FrameworkBundle(),  new Symfony\Bundle\TwigBundle\TwigBundle(),  *// ...*  );  *// ...*  return $bundles;  }  *// ...*  } |

* For example, TwigBundle is responsible for adding the Twig tool to your app!

## Configuration: config.yml

Para saber que podemos configurar:

php bin/console config:dump-reference twig

## The imports Key

Similar al include de PHP

The parameters Key: Parameters (Variables)[¶](http://symfony.com/doc/current/configuration.html#the-parameters-key-parameters-variables)

* Another special key is called parameters: it's used to define *variables* that can be referenced in *any* other configuration file.
* For example, in config.yml, a locale parameter is defined and then referenced below under the framework key:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | *# app/config/config.yml*  *# ...*  parameters:  locale: en  framework:  *# ...*  *# any string surrounded by two % is replaced by that parameter value*  default\_locale: "%locale%"  *# ...* |

### **The Special parameters.yml File**

* Estos parametros son usados dentro del config.yml
* Because of that, **parameters.yml is not committed to your version control**

Environments & the Other Config Files[¶](http://symfony.com/doc/current/configuration.html#environments-the-other-config-files)

You have just *one* app, but whether you realize it or not, you need it to behave *differently* at different times:

* While **developing**, you want your app to log everything and expose nice debugging tools;
* After deploying to **production**, you want that *same* app to be optimized for speed and only log errors.

How can you make *one* application behave in two different ways? With *environments*.

# **Routing**

<http://symfony.com/doc/current/routing.html>

The Symfony router lets you define creative URLs that you map to different areas of your application. By the end of this chapter, you'll be able to:

* Create complex routes that map to controllers
* Generate URLs inside templates and controllers
* Load routing resources from bundles (or anywhere else)
* Debug your routes

<?php

// src/AppBundle/Controller/LuckyController.php

namespace AppBundle\Controller;

use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;

use Symfony\Component\HttpFoundation\Response;

// --> add this new use statement

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class CommentsController extends Controller

{

/\*\*

\* @Route("/comentarios", name="comments\_login")

\*/

public function loginAction()

{

return $this->render('comments/login.html.twig');

}

/\*\*

\* @Route("/comentarios/lista", name="comments\_list")

\*/

public function listAction()

{

}

/\*\*

\* @Route("/comentarios/nuevo", name="comments\_new")

\*/

public function newAction()

{

}

/\*\*

\* @Route("/comentarios/editar/{id}", name="comments\_edit", requirements={"page": "\d+"})

\*/

public function editAction($id)

{

}

}

## Adding {wildcard} Requirements

*/\*\**

*\* @Route("/blog/{page}", name="blog\_list", requirements={"page": "\d+"})*

*\*/*

## Giving {placeholders} a Default Value

public function listAction($page = 1)

{

*// ...*

}

}

### **Advanced Routing Example**

*// src/AppBundle/Controller/ArticleController.php*

*// ...*

class ArticleController extends Controller

{

*/\*\**

*\* @Route(*

*\* "/articles/{\_locale}/{year}/{slug}.{\_format}",*

*\* defaults={"\_format": "html"},*

*\* requirements={*

*\* "\_locale": "en|fr",*

*\* "\_format": "html|rss",*

*\* "year": "\d+"*

*\* }*

*\* )*

*\*/*

public function showAction($\_locale, $year, $slug)

{

}

}

* /articles/en/2010/my-post
* /articles/fr/2010/my-post.rss
* /articles/en/2013/my-latest-post.html

## Generating URLs

class MainController extends Controller

{

public function showAction($slug)

{

*// ...*

*// /blog/my-blog-post*

$url = $this->generateUrl(

'blog\_show',

array('slug' => 'my-blog-post')

);

}

}

### **Generating URLs with Query Strings**

The generate() method takes an array of wildcard values to generate the URI. But if you pass extra ones, they will be added to the URI as a query string:

$this->get('router')->generate('blog', array(

'page' => 2,

'category' => 'Symfony'

));

*// /blog/2?category=Symfony*

### **Generating URLs from a Template**

<http://symfony.com/doc/current/templating.html#templating-pages>

Simple

<a href="{{ path('welcome') }}">Home</a>

Con parámetros

*{# app/Resources/views/article/recent\_list.html.twig #}*

{% for article in articles %}

<a href="{{ path('article\_show', {'slug': article.slug}) }}">

{{ article.title }}

</a>

{% endfor %}

### **Linking to Assets**[**¶**](http://symfony.com/doc/current/templating.html#linking-to-assets)

Templates also commonly refer to images, JavaScript, stylesheets and other assets. Of course you could hard-code the path to these assets (e.g. /images/logo.png), but Symfony provides a more dynamic option via the asset() Twig function:

* [Twig](http://symfony.com/doc/current/templating.html)

|  |  |
| --- | --- |
| 1  2  3 | <img src="{{ asset('images/logo.png') }}" alt="Symfony!" />  <link href="{{ asset('css/blog.css') }}" rel="stylesheet" /> |

* [PHP](http://symfony.com/doc/current/templating.html)

The asset() function's main purpose is to make your application more portable. If your application lives at the root of your host (e.g. http://example.com), then the rendered paths should be /images/logo.png. But if your application lives in a subdirectory (e.g. http://example.com/my\_app), each asset path should render with the subdirectory (e.g. /my\_app/images/logo.png). The asset() function takes care of this by determining how your application is being used and generating the correct paths accordingly.

Additionally, if you use the asset() function, Symfony can automatically append a query string to your asset, in order to guarantee that updated static assets won't be loaded from cache after being deployed. For example, /images/logo.png might look like /images/logo.png?v2. For more information, see the [version](http://symfony.com/doc/current/reference/configuration/framework.html#reference-framework-assets-version) configuration option.

If you need absolute URLs for assets, use the absolute\_url() Twig function as follows:

|  |  |
| --- | --- |
| 1 | <img src="{{ absolute\_url(asset('images/logo.png')) }}" alt="Symfony!" /> |

## Including Stylesheets and JavaScripts in Twig

Start by adding two blocks to your base template that will hold your assets: one called stylesheets inside the head tag and another called javascripts just above the closing bodytag.

*{# app/Resources/views/base.html.twig #}*

<html>

<head>

*{# ... #}*

{% block stylesheets %}

<link href="{{ asset('css/main.css') }}" rel="stylesheet" />

{% endblock %}

</head>

<body>

*{# ... #}*

{% block javascripts %}

<script src="{{ asset('js/main.js') }}"></script>

{% endblock %}

</body>

</html>

## Include an extra stylesheet or JavaScript from a child template?

For example, suppose you have a contact page and you need to include a contact.cssstylesheet *just* on that page.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | *{# app/Resources/views/contact/contact.html.twig #}*  {% extends 'base.html.twig' %}  {% block stylesheets %}  {{ parent() }}  <link href="{{ asset('css/contact.css') }}" rel="stylesheet" />  {% endblock %}  *{# ... #}* |

Output Escaping[¶](http://symfony.com/doc/current/templating.html#output-escaping)

Twig performs automatic "output escaping" when rendering any content in order to protect you from Cross Site Scripting (XSS) attacks.

Suppose description equals I <3 this product:

|  |  |
| --- | --- |
| 1  2  3  4  5 | <!-- output escaping is on automatically -->  {{ description }} <!-- I &lt;3 this product -->  <!-- disable output escaping with the raw filter -->  {{ description|raw }} <!-- I <3 this product --> |

PHP templates do not automatically escape content.

# **Controller**

<http://symfony.com/doc/current/controller.html>

### **Redirecting**

public function indexAction()

{

*// redirect to the "homepage" route*

return $this->redirectToRoute('homepage');

*// do a permanent - 301 redirect*

return $this->redirectToRoute('homepage', array(), 301);

*// redirect to a route with parameters*

return $this->redirectToRoute('blog\_show', array('slug' => 'my-page'));

*// redirect externally*

return $this->redirect('http://symfony.com/doc');

}

## Managing Errors and 404 Pages

## http://symfony.com/doc/current/controller.html#managing-errors-and-404-pages

if (!$product) {

throw $this->createNotFoundException('The product does not exist');

}

## Example 404 Error Template

<http://symfony.com/doc/current/controller/error_pages.html>

*{# app/Resources/TwigBundle/views/Exception/error404.html.twig #}*

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

{% block body %}

<h1>Page not found</h1>

{% if is\_granted('IS\_AUTHENTICATED\_FULLY') %}

*{# ... #}*

{% endif %}

<p>

The requested page couldn't be located. Checkout for any URL

misspelling or <a href="{{ path('homepage') }}">return to the homepage</a>.

</p>

{% endblock %}

### **Testing Error Pages during Development**

*# app/config/routing\_dev.yml*

\_errors:

resource: "@TwigBundle/Resources/config/routing/errors.xml"

prefix: /\_error

## Executing an Application in different Environments

<https://symfony.com/doc/current/configuration/environments.html>

app.php

app\_dev.php

## The Request object as a Controller Argument

if you need to read query parameters, grab a request header or get access to an uploaded file? All of that information is stored in Symfony's Request object.

use Symfony\Component\HttpFoundation\Request;

public function indexAction(Request $request, $firstName, $lastName)

{

$page = $request->query->get('page', 1);

*// ...*

}

## Managing the Session

<http://symfony.com/doc/current/controller.html#managing-the-session>

use Symfony\Component\HttpFoundation\Request;

public function indexAction(Request $request)

{

$session = $request->getSession();

*// store an attribute for reuse during a later user request*

$session->set('foo', 'bar');

*// get the attribute set by another controller in another request*

$foobar = $session->get('foobar');

*// use a default value if the attribute doesn't exist*

$filters = $session->get('filters', array());

}

### **Flash Messages**

flash messages are meant to be used exactly once

use Symfony\Component\HttpFoundation\Request;

public function updateAction(Request $request)

{

*// ...*

if ($form->isSubmitted() && $form->isValid()) {

*// do some sort of processing*

$this->addFlash(

'notice',

'Your changes were saved!'

);

*// $this->addFlash() is equivalent to $request->getSession()->getFlashBag()->add()*

return $this->redirectToRoute(...);

}

return $this->render(...);

}

In the template of the next page (or even better, in your base layout template), read any flash messages from the session:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | *{# app/Resources/views/base.html.twig #}*  {% for flash\_message in app.session.flashBag.get('notice') %}  <div class="flash-notice">  {{ flash\_message }}  </div>  {% endfor %} |

It's common to use notice, warning and error as the keys of the different types of flash messages, but you can use any key that fits your needs.

## The Request and Response Object

<http://symfony.com/doc/current/controller.html#the-request-and-response-object>

use Symfony\Component\HttpFoundation\Request;

public function indexAction(Request $request)

{

$request->isXmlHttpRequest(); *// is it an Ajax request?*

$request->getPreferredLanguage(array('en', 'fr'));

*// retrieve GET and POST variables respectively*

$request->query->get('page');

$request->request->get('page');

*// retrieve SERVER variables*

$request->server->get('HTTP\_HOST');

*// retrieves an instance of UploadedFile identified by foo*

$request->files->get('foo');

*// retrieve a COOKIE value*

$request->cookies->get('PHPSESSID');

*// retrieve an HTTP request header, with normalized, lowercase keys*

$request->headers->get('host');

$request->headers->get('content\_type');

}

### **File helper**

You can use the [file()](http://api.symfony.com/3.2/Symfony/Bundle/FrameworkBundle/Controller/Controller.html#method_file) helper to serve a file from inside a controller:

|  |  |
| --- | --- |
| 1  2  3  4  5 | public function fileAction()  {  *// send the file contents and force the browser to download it*  return $this->file('/path/to/some\_file.pdf');  } |

The file() helper provides some arguments to configure its behavior:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | use Symfony\Component\HttpFoundation\File\File;  use Symfony\Component\HttpFoundation\ResponseHeaderBag;  public function fileAction()  {  *// load the file from the filesystem*  $file = new File('/path/to/some\_file.pdf');  return $this->file($file);  *// rename the downloaded file*  return $this->file($file, 'custom\_name.pdf');  *// display the file contents in the browser instead of downloading it*  return $this->file('invoice\_3241.pdf', 'my\_invoice.pdf', ResponseHeaderBag::DISPOSITION\_INLINE);  } |

# **Creating and Using Templates**

<http://symfony.com/doc/current/templating.html>

Twig defines three types of special syntax:

{{ ... }}

"Says something": prints a variable or the result of an expression to the template.

{% ... %}

"Does something": a **tag** that controls the logic of the template; it is used to execute statements such as for-loops for example.

{# ... #}

"Comment something": it's the equivalent of the PHP /\* comment \*/ syntax. It's used to add single or multi-line comments. The content of the comments isn't included in the rendered pages.

Twig also contains **filters**, which modify content before being rendered. The following makes the title variable all uppercase before rendering it:

|  |  |
| --- | --- |
| 1 | {{ title|upper }} |

TAGS

<http://twig.sensiolabs.org/doc/tags/index.html>

FILTERS

<http://twig.sensiolabs.org/doc/filters/index.html>

FUNCTIONS

<http://twig.sensiolabs.org/doc/functions/index.html>

Ejemplo print div with odd and even classes

{% for i in 1..10 %}

<div class="{{ cycle(['even', 'odd'], i) }}">

*<!-- some HTML here -->*

</div>

{% endfor %}

Ejemplo for - else

<ul>

{% for user in users if user.active %}

<li>{{ user.username }}</li>

{% else %}

<li>No users found</li>

{% endfor %}

</ul>

## Template Inheritance and Layouts

## The one that comes for default

## <!DOCTYPE html>

## <html>

## <head>

## <meta charset="UTF-8" />

## <title>{% block title %}Welcome!{% endblock %}</title>

## {% block stylesheets %}{% endblock %}

## <link rel="icon" type="image/x-icon" href="{{ asset('favicon.ico') }}" />

## </head>

## <body>

## {% block body %}{% endblock %}

## {% block javascripts %}{% endblock %}

## </body>

## </html>

## Another example with sidebar

BASE TEMPLATE

|  |
| --- |
| *{# app/Resources/views/base.html.twig #}*  <!DOCTYPE html>  <html>  <head>  <meta charset="UTF-8">  <title>{% block title %}Test Application{% endblock %}</title>  </head>  <body>  <div id="sidebar">  {% block sidebar %}  <ul>  <li><a href="/">Home</a></li>  <li><a href="/blog">Blog</a></li>  </ul>  {% endblock %}  </div>  <div id="content">  {% block body %}{% endblock %}  </div>  </body>  </html> |

If you need to get the content of a block from the parent template, you can use the {{ parent() }} function. This is useful **if you want to add to the contents of a parent block instead of completely overriding it:**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | {% block sidebar %}  <h3>Table of Contents</h3>  *{# ... #}*  {{ parent() }}  {% endblock %} |

### **Including other Templates**

*{# app/Resources/views/article/article\_details.html.twig #}*

<h2>{{ article.title }}</h2>

<h3 class="byline">by {{ article.authorName }}</h3>

<p>

{{ article.body }}

</p>

*{# app/Resources/views/article/list.html.twig #}*

{% extends 'layout.html.twig' %}

{% block body %}

<h1>Recent Articles<h1>

{% for article in articles %}

{{ include('article/article\_details.html.twig', { 'article': article }) }}

{% endfor %}

{% endblock %}

# **How to Access the User, Request, Session & more in Twig via the app Variable**

<http://symfony.com/doc/current/templating/app_variable.html>

<p>Username: {{ app.user.username }}</p>

{% if app.debug %}

<p>Request method: {{ app.request.method }}</p>

<p>Application Environment: {{ app.environment }}</p>

{% endif %}

# **Databases and the Doctrine ORM**

<http://symfony.com/doc/current/doctrine.html>

parameters.yml contains the config ifo fot the DB

php bin/console doctrine:database:create

*Setting up the Database to be UTF8*

*# app/config/config.yml*

doctrine:

dbal:

charset: utf8mb4

default\_table\_options:

charset: utf8mb4

collate: utf8mb4\_unicode\_ci

php bin/console doctrine:database:drop --force

php bin/console doctrine:database:create

### **Creating an Entity Class**

*// src/AppBundle/Entity/Product.php*

namespace AppBundle\Entity;

class Product

{

private $name;

private $price;

private $description;

}

**Command to create entities**

php bin/console doctrine:generate:entity

**validate the mappings with the following command:**

|  |
| --- |
| php bin/console doctrine:schema:validate |

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

### **Generating Getters and Setters**

|  |
| --- |
| php bin/console doctrine:generate:entities AppBundle/Entity/Product |

**Generate all entities in AppBundle**

php bin/console doctrine:generate:entities AppBundle

### **Persisting Objects to the Database**

*// src/AppBundle/Controller/DefaultController.php*

*// ...*

use AppBundle\Entity\Product;

use Symfony\Component\HttpFoundation\Response;

*// ...*

public function createAction()

{

$product = new Product();

$product->setName('Keyboard');

$product->setPrice(19.99);

$product->setDescription('Ergonomic and stylish!');

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

*// tells Doctrine you want to (eventually) save the Product (no queries yet)*

$em->persist($product);

*// actually executes the queries (i.e. the INSERT query)*

$em->flush();

return new Response('Saved new product with id '.$product->getId());

}

### **Fetching Objects from the Database**

public function showAction($productId)

{

$product = $this->getDoctrine()

->getRepository('AppBundle:Product')

->find($productId);

if (!$product) {

throw $this->createNotFoundException(

'No product found for id '.$productId

);

}

*// ... do something, like pass the $product object into a template*

}

**Once you have a repository object, you can access all sorts of helpful methods:**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | $repository = $this->getDoctrine()->getRepository('AppBundle:Product');  *// query for a single product by its primary key (usually "id")*  $product = $repository->find($productId);  *// dynamic method names to find a single product based on a column value*  $product = $repository->findOneById($productId);  $product = $repository->findOneByName('Keyboard');  *// dynamic method names to find a group of products based on a column value*  $products = $repository->findByPrice(19.99);  *// find \*all\* products*  $products = $repository->findAll(); |

You can also take advantage of the useful findBy() and findOneBy() methods to easily fetch objects based on multiple conditions:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | $repository = $this->getDoctrine()->getRepository('AppBundle:Product');  *// query for a single product matching the given name and price*  $product = $repository->findOneBy(  array('name' => 'Keyboard', 'price' => 19.99)  );  *// query for multiple products matching the given name, ordered by price*  $products = $repository->findBy(  array('name' => 'Keyboard'),  array('price' => 'ASC')  ); |

### **Updating an Object**

public function updateAction($productId)

{

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

$product = $em->getRepository('AppBundle:Product')->find($productId);

if (!$product) {

throw $this->createNotFoundException(

'No product found for id '.$productId

);

}

$product->setName('New product name!');

$em->flush();

return $this->redirectToRoute('homepage');

}

### **Deleting an Object**

$em->remove($product);

$em->flush();

### **Querying for Objects with DQL**

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

$query = $em->createQuery(

'SELECT p

FROM AppBundle:Product p

WHERE p.price > :price

ORDER BY p.price ASC'

)->setParameter('price', 19.99);

$products = $query->getResult();

GET ONE OR NULL

$product = $query->setMaxResults(1)->getOneOrNullResult();

### **Querying for Objects Using Doctrine's Query Builder**

$repository = $this->getDoctrine()

->getRepository('AppBundle:Product');

*// createQueryBuilder() automatically selects FROM AppBundle:Product*

*// and aliases it to "p"*

$query = $repository->createQueryBuilder('p')

->where('p.price > :price')

->setParameter('price', '19.99')

->orderBy('p.price', 'ASC')

->getQuery();

$products = $query->getResult();

*// to get just one result:*

*// $product = $query->setMaxResults(1)->getOneOrNullResult();*

## Relationship Mapping Metadata

many products to onecategory (or equivalently, one category to many products).

To relate the Product and Category entities, simply create a category property on the Productclass, annotated as follows:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | *// src/AppBundle/Entity/Product.php*  *// ...*  class Product  {  *// ...*  */\*\**  *\* @ORM\ManyToOne(targetEntity="Category", inversedBy="products")*  *\* @ORM\JoinColumn(name="category\_id", referencedColumnName="id")*  *\*/*  private $category;  } |

*// src/AppBundle/Entity/Category.php*

*// ...*

use Doctrine\Common\Collections\ArrayCollection;

class Category

{

*// ...*

*/\*\**

*\* @ORM\OneToMany(targetEntity="Product", mappedBy="category")*

*\*/*

private $products;

public function \_\_construct()

{

$this->products = new ArrayCollection();

}

}

php bin/console doctrine:generate:entities AppBundle

Forms

<http://symfony.com/doc/current/forms.html>

php bin/console generate:doctrine:form AppBundle:Usuario

## Creating a Simple Form

This example shows you how to build your form directly in the controller

*// src/AppBundle/Controller/DefaultController.php*

namespace AppBundle\Controller;

use AppBundle\Entity\Task;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

use Symfony\Component\HttpFoundation\Request;

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

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

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

class DefaultController extends Controller

{

public function newAction(Request $request)

{

*// create a task and give it some dummy data for this example*

$task = new Task();

$task->setTask('Write a blog post');

$task->setDueDate(new \DateTime('tomorrow'));

$form = $this->createFormBuilder($task)

->add('task', TextType::class)

->add('dueDate', DateType::class)

->add('save', SubmitType::class, array('label' => 'Create Post'))

->getForm();

return $this->render('default/new.html.twig', array(

'form' => $form->createView(),

));

}

}

### **Rendering the Form**

*{# app/Resources/views/default/new.html.twig #}*

{{ form\_start(form) }}

{{ form\_widget(form) }}

{{ form\_end(form) }}

form\_start(form)

Renders the start tag of the form, including the correct enctype attribute when using file uploads.

form\_widget(form)

Renders all the fields, which includes the field element itself, a label and any validation error messages for the field.

form\_end(form)

Renders the end tag of the form and any fields that have not yet been rendered, in case you rendered each field yourself. This is useful for rendering hidden fields and taking advantage of the automatic [CSRF Protection](http://symfony.com/doc/current/form/csrf_protection.html).

*Setting the data\_class*

Every form needs to know the name of the class that holds the underlying data (e.g. AppBundle\Entity\Task). Usually, this is just guessed based off of the object passed to the second argument to createForm() (i.e. $task). Later, when you begin embedding forms, this will no longer be sufficient. So, while not always necessary, it's generally a good idea to explicitly specify the data\_class option by adding the following to your form type class:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | use AppBundle\Entity\Task;  use Symfony\Component\OptionsResolver\OptionsResolver;  *// ...*  public function configureOptions(OptionsResolver $resolver)  {  $resolver->setDefaults(array(  'data\_class' => Task::class,  ));  } |

# **How to Control the Rendering of a Form**

<https://symfony.com/doc/current/form/rendering.html>

*{# app/Resources/views/default/new.html.twig #}*

{{ form\_start(form) }}

{{ form\_errors(form) }}

{{ form\_row(form.task) }}

{{ form\_row(form.dueDate) }}

{{ form\_end(form) }}

form\_errors(form)

Renders any errors global to the whole form (field-specific errors are displayed next to each field).

form\_row(form.dueDate)

Renders the label, any errors, and the HTML form widget for the given field (e.g. dueDate) inside, by default, a div element.

You can access the current data of your form via form.vars.value:

{{ form.vars.value.task }}

{{ form\_start(form) }}

{{ form\_errors(form) }}

<div>

{{ form\_label(form.task) }}

{{ form\_errors(form.task) }}

{{ form\_widget(form.task) }}

</div>

<div>

{{ form\_label(form.dueDate) }}

{{ form\_errors(form.dueDate) }}

{{ form\_widget(form.dueDate) }}

</div>

<div>

{{ form\_widget(form.save) }}

</div>

{{ form\_end(form) }}

If the auto-generated label for a field isn't quite right, you can explicitly specify it:

{{ form\_label(form.task, 'Task Description') }}

To edit the class attribute:

{{ form\_widget(form.task, {'attr': {'class': 'task\_field'}}) }}

 you can access individual values for fields such as the id, name and label. For example to get the id:

{{ form.task.vars.id }}

To get the value used for the form field's name attribute

{{ form.task.vars.full\_name }}

# **Twig Template Form Function and Variable Reference**

<https://symfony.com/doc/current/reference/forms/twig_reference.html>

# **How to Customize Form Rendering**

<https://symfony.com/doc/current/form/form_customization.html>

## What are Form Themes?

**built-in form themes**

* [form\_div\_layout.html.twig](https://github.com/symfony/symfony/blob/master/src/Symfony/Bridge/Twig/Resources/views/Form/form_div_layout.html.twig), wraps each form field inside a <div> element.
* [form\_table\_layout.html.twig](https://github.com/symfony/symfony/blob/master/src/Symfony/Bridge/Twig/Resources/views/Form/form_table_layout.html.twig), wraps the entire form inside a <table> element and each form field inside a <tr> element.
* [bootstrap\_3\_layout.html.twig](https://github.com/symfony/symfony/blob/master/src/Symfony/Bridge/Twig/Resources/views/Form/bootstrap_3_layout.html.twig), wraps each form field inside a <div> element with the appropriate CSS classes to apply the default [Bootstrap 3 CSS framework](http://getbootstrap.com/) styles.
* [bootstrap\_3\_horizontal\_layout.html.twig](https://github.com/symfony/symfony/blob/master/src/Symfony/Bridge/Twig/Resources/views/Form/bootstrap_3_horizontal_layout.html.twig), it's similar to the previous theme, but the CSS classes applied are the ones used to display the forms horizontally (i.e. the label and the widget in the same row).
* [foundation\_5\_layout.html.twig](https://github.com/symfony/symfony/blob/master/src/Symfony/Bridge/Twig/Resources/views/Form/foundation_5_layout.html.twig), wraps each form field inside a <div> element with the appropriate CSS classes to apply the default [Foundation CSS framework](http://foundation.zurb.com/) styles.

{{ form\_widget(form.age) }}

renders:

<input type="number" id="form\_age" name="form[age]" required="required" value="33" />

The default implementation of the integer\_widget fragment looks like this:

|  |  |
| --- | --- |
| 1  2  3  4  5 | *{# form\_div\_layout.html.twig #}*  {% block integer\_widget %}  {% set type = type|default('number') %}  {{ block('form\_widget\_simple') }}  {% endblock integer\_widget %} |

As you can see, this fragment itself renders another fragment - form\_widget\_simple:

|  |  |
| --- | --- |
| 1  2  3  4  5 | *{# form\_div\_layout.html.twig #}*  {% block form\_widget\_simple %}  {% set type = type|default('text') %}  <input type="{{ type }}" {{ block('widget\_attributes') }} {% if value is not empty %}value="{{ value }}" {% endif %}/>  {% endblock form\_widget\_simple %} |

The point is, the fragments dictate the HTML output of each part of a form. **To customize the form output, you just need to identify and override the correct fragment.** A set of these form fragment customizations is known as a form "theme". When rendering a form, you can choose which form theme(s) you want to apply.

## Form Theming

| **Method** | **Pros** | **Cons** |
| --- | --- | --- |
| Inside the same template as the form | Quick and easy | Can't be reused in other templates |
| Inside a separate template | Can be reused by many templates | Requires an extra template to be created |

### **Method 1: Inside the same Template as the Form**

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

{% form\_theme form \_self %}

{% block integer\_widget %}

<div class="integer\_widget">

{% set type = type|default('number') %}

{{ block('form\_widget\_simple') }}

</div>

{% endblock %}

{% block content %}

*{# ... render the form #}*

{{ form\_row(form.age) }}

{% endblock %}

### **Method 2: Inside a separate Template**

*{# app/Resources/views/form/fields.html.twig #}*

{% block integer\_widget %}

<div class="integer\_widget">

{% set type = type|default('number') %}

{{ block('form\_widget\_simple') }}

</div>

{% endblock %}

Inside the template where you're actually rendering your form  tell Symfony to use the template via the form\_theme tag:

{% form\_theme form 'form/fields.html.twig' %}

{{ form\_widget(form.age) }}

#### **Multiple Templates**

{% form\_theme form with ['common.html.twig', 'form/fields.html.twig'] %}

*{# ... #}*

#### **Child Forms**

{% form\_theme form 'form/fields.html.twig' %}

{% form\_theme form.child 'form/fields\_child.html.twig' %}

## Making Application-wide Customizations

*# app/config/config.yml*

twig:

form\_themes:

- 'form/fields.html.twig'

*# ...*

*# app/config/config.yml*

twig:

form\_themes:

- 'form\_table\_layout.html.twig'

*# ...*

If you only want to make the change in one template, add the following line to your template file rather than adding the template as a resource:

|  |
| --- |
| {% form\_theme form 'form\_table\_layout.html.twig' %} |

## How to Customize an individual Field

{% form\_theme form \_self %}

{% block \_product\_name\_widget %}

<div class="text\_widget">

{{ block('form\_widget\_simple') }}

</div>

{% endblock %}

{{ form\_widget(form.name) }}

Here, the \_product\_name\_widget fragment defines the template to use for the field whose id is product\_name (and name is product[name]).

You can also override the markup for an entire field row using the same method:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | {% form\_theme form \_self %}  {% block \_product\_name\_row %}  <div class="name\_row">  {{ form\_label(form) }}  {{ form\_errors(form) }}  {{ form\_widget(form) }}  </div>  {% endblock %}  {{ form\_row(form.name) }} |

### **Customizing Error Output**

{{ form\_errors(form.age) }}

By default, the errors are rendered inside an unordered list:

<ul>

<li>This field is required</li>

</ul>

To override how errors are rendered for all fields, simply copy, paste and customize the form\_errors fragment.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | *{# form\_errors.html.twig #}*  {% block form\_errors %}  {% spaceless %}  {% if errors|length > 0 %}  <ul>  {% for error in errors %}  <li>{{ error.message }}</li>  {% endfor %}  </ul>  {% endif %}  {% endspaceless %}  {% endblock form\_errors %} |

To customize the general errors:

{{ form\_errors(form) }}

if the compound variable is set to true. If it is true, it means that what's being currently rendered is a collection of fields (e.g. a whole form), and not just an individual field.

*{# form\_errors.html.twig #}*

{% block form\_errors %}

{% spaceless %}

{% if errors|length > 0 %}

{% if compound %}

<ul>

{% for error in errors %}

<li>{{ error.message }}</li>

{% endfor %}

</ul>

{% else %}

*{# ... display the errors for a single field #}*

{% endif %}

{% endif %}

{% endspaceless %}

{% endblock form\_errors %}

### **Adding a "Required" Asterisk to Field Labels**[**¶**](https://symfony.com/doc/current/form/form_customization.html#adding-a-required-asterisk-to-field-labels)

If you want to denote all of your required fields with a required asterisk (\*), you can do this by customizing the form\_label fragment.

In Twig, if you're making the form customization inside the same template as your form, modify the use tag and add the following:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | {% use 'form\_div\_layout.html.twig' with form\_label as base\_form\_label %}  {% block form\_label %}  {{ block('base\_form\_label') }}  {% if required %}  <span class="required" title="This field is required">\*</span>  {% endif %}  {% endblock %} |

In Twig, if you're making the form customization inside a separate template, use the following:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | {% extends 'form\_div\_layout.html.twig' %}  {% block form\_label %}  {{ parent() }}  {% if required %}  <span class="required" title="This field is required">\*</span>  {% endif %}  {% endblock %} |

*Using CSS only*

By default, label tags of required fields are rendered with a required CSS class. Thus, you can also add an asterisk using CSS only:

|  |  |
| --- | --- |
| 1  2  3 | label.required:before {  content: "\* ";  } |

### **Adding "help" Messages**[**¶**](https://symfony.com/doc/current/form/form_customization.html#adding-help-messages)

You can also customize your form widgets to have an optional "help" message.

In Twig, if you're making the form customization inside the same template as your form, modify the use tag and add the following:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | {% use 'form\_div\_layout.html.twig' with form\_widget\_simple as base\_form\_widget\_simple %}  {% block form\_widget\_simple %}  {{ block('base\_form\_widget\_simple') }}  {% if help is defined %}  <span class="help-block">{{ help }}</span>  {% endif %}  {% endblock %} |

In Twig, if you're making the form customization inside a separate template, use the following:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | {% extends 'form\_div\_layout.html.twig' %}  {% block form\_widget\_simple %}  {{ parent() }}  {% if help is defined %}  <span class="help-block">{{ help }}</span>  {% endif %}  {% endblock %} |

### **Handling Form Submissions**

*// ...*

use Symfony\Component\HttpFoundation\Request;

public function newAction(Request $request)

{

*// just setup a fresh $task object (remove the dummy data)*

$task = new Task();

$form = $this->createFormBuilder($task)

->add('task', TextType::class)

->add('dueDate', DateType::class)

->add('save', SubmitType::class, array('label' => 'Create Task'))

->getForm();

$form->handleRequest($request);

if ($form->isSubmitted() && $form->isValid()) {

*// $form->getData() holds the submitted values*

*// but, the original `$task` variable has also been updated*

$task = $form->getData();

*// ... perform some action, such as saving the task to the database*

*// for example, if Task is a Doctrine entity, save it!*

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

*// $em->persist($task);*

*// $em->flush();*

return $this->redirectToRoute('task\_success');

}

return $this->render('default/new.html.twig', array(

'form' => $form->createView(),

));

}

Redirecting a user after a successful form submission prevents the user from being able to hit the "Refresh" button of their browser and re-post the data.

## Form Validation

*// src/AppBundle/Entity/Task.php*

namespace AppBundle\Entity;

use Symfony\Component\Validator\Constraints as Assert;

class Task

{

*/\*\**

*\* @Assert\NotBlank()*

*\*/*

public $task;

*/\*\**

*\* @Assert\NotBlank()*

*\* @Assert\Type("\DateTime")*

*\*/*

protected $dueDate;

}

*HTML5 Validation*

*{# app/Resources/views/default/new.html.twig #}*

{{ form(form, {'attr': {'novalidate': 'novalidate'}}) }}

## Constraints

## <http://symfony.com/doc/current/validation.html#constraints>

### **Basic Constraints**[**¶**](http://symfony.com/doc/current/validation.html#basic-constraints)

These are the basic constraints: use them to assert very basic things about the value of properties or the return value of methods on your object.

* [NotBlank](http://symfony.com/doc/current/reference/constraints/NotBlank.html)
* [Blank](http://symfony.com/doc/current/reference/constraints/Blank.html)
* [NotNull](http://symfony.com/doc/current/reference/constraints/NotNull.html)
* [IsNull](http://symfony.com/doc/current/reference/constraints/IsNull.html)
* [IsTrue](http://symfony.com/doc/current/reference/constraints/IsTrue.html)
* [IsFalse](http://symfony.com/doc/current/reference/constraints/IsFalse.html)
* [Type](http://symfony.com/doc/current/reference/constraints/Type.html)

### **String Constraints**[**¶**](http://symfony.com/doc/current/validation.html#string-constraints)

* [Email](http://symfony.com/doc/current/reference/constraints/Email.html)
* [Length](http://symfony.com/doc/current/reference/constraints/Length.html)
* [Url](http://symfony.com/doc/current/reference/constraints/Url.html)
* [Regex](http://symfony.com/doc/current/reference/constraints/Regex.html)
* [Ip](http://symfony.com/doc/current/reference/constraints/Ip.html)
* [Uuid](http://symfony.com/doc/current/reference/constraints/Uuid.html)

### **Number Constraints**[**¶**](http://symfony.com/doc/current/validation.html#number-constraints)

* [Range](http://symfony.com/doc/current/reference/constraints/Range.html)

### **Comparison Constraints**[**¶**](http://symfony.com/doc/current/validation.html#comparison-constraints)

* [EqualTo](http://symfony.com/doc/current/reference/constraints/EqualTo.html)
* [NotEqualTo](http://symfony.com/doc/current/reference/constraints/NotEqualTo.html)
* [IdenticalTo](http://symfony.com/doc/current/reference/constraints/IdenticalTo.html)
* [NotIdenticalTo](http://symfony.com/doc/current/reference/constraints/NotIdenticalTo.html)
* [LessThan](http://symfony.com/doc/current/reference/constraints/LessThan.html)
* [LessThanOrEqual](http://symfony.com/doc/current/reference/constraints/LessThanOrEqual.html)
* [GreaterThan](http://symfony.com/doc/current/reference/constraints/GreaterThan.html)
* [GreaterThanOrEqual](http://symfony.com/doc/current/reference/constraints/GreaterThanOrEqual.html)

### **Date Constraints**[**¶**](http://symfony.com/doc/current/validation.html#date-constraints)

* [Date](http://symfony.com/doc/current/reference/constraints/Date.html)
* [DateTime](http://symfony.com/doc/current/reference/constraints/DateTime.html)
* [Time](http://symfony.com/doc/current/reference/constraints/Time.html)

### **Collection Constraints**[**¶**](http://symfony.com/doc/current/validation.html#collection-constraints)

* [Choice](http://symfony.com/doc/current/reference/constraints/Choice.html)
* [Collection](http://symfony.com/doc/current/reference/constraints/Collection.html)
* [Count](http://symfony.com/doc/current/reference/constraints/Count.html)
* [UniqueEntity](http://symfony.com/doc/current/reference/constraints/UniqueEntity.html)
* [Language](http://symfony.com/doc/current/reference/constraints/Language.html)
* [Locale](http://symfony.com/doc/current/reference/constraints/Locale.html)
* [Country](http://symfony.com/doc/current/reference/constraints/Country.html)

### **File Constraints**[**¶**](http://symfony.com/doc/current/validation.html#file-constraints)

* [File](http://symfony.com/doc/current/reference/constraints/File.html)
* [Image](http://symfony.com/doc/current/reference/constraints/Image.html)

### **Financial and other Number Constraints**[**¶**](http://symfony.com/doc/current/validation.html#financial-and-other-number-constraints)

* [Bic](http://symfony.com/doc/current/reference/constraints/Bic.html)
* [CardScheme](http://symfony.com/doc/current/reference/constraints/CardScheme.html)
* [Currency](http://symfony.com/doc/current/reference/constraints/Currency.html)
* [Luhn](http://symfony.com/doc/current/reference/constraints/Luhn.html)
* [Iban](http://symfony.com/doc/current/reference/constraints/Iban.html)
* [Isbn](http://symfony.com/doc/current/reference/constraints/Isbn.html)
* [Issn](http://symfony.com/doc/current/reference/constraints/Issn.html)

### **Other Constraints**[**¶**](http://symfony.com/doc/current/validation.html#other-constraints)

* [Callback](http://symfony.com/doc/current/reference/constraints/Callback.html)
* [Expression](http://symfony.com/doc/current/reference/constraints/Expression.html)
* [All](http://symfony.com/doc/current/reference/constraints/All.html)
* [UserPassword](http://symfony.com/doc/current/reference/constraints/UserPassword.html)
* [Valid](http://symfony.com/doc/current/reference/constraints/Valid.html)

*// src/AppBundle/Entity/Author.php*

*// ...*

use Symfony\Component\Validator\Constraints as Assert;

class Author

{

*/\*\**

*\* @Assert\Choice(*

*\* choices = { "male", "female", "other" },*

*\* message = "Choose a valid gender."*

*\* )*

*\*/*

public $gender;

*// ...*

}

*// src/AppBundle/Entity/Author.php*

*// ...*

use Symfony\Component\Validator\Constraints as Assert;

class Author

{

*/\*\**

*\* @Assert\Choice({"male", "female", "other"})*

*\*/*

protected $gender;

*// ...*

}

### **Validating Properties**

*// src/AppBundle/Entity/Author.php*

*// ...*

use Symfony\Component\Validator\Constraints as Assert;

class Author

{

*/\*\**

*\* @Assert\NotBlank()*

*\* @Assert\Length(min=3)*

*\*/*

private $firstName;

}

### **Getters**

Symfony allows you to add a constraint to any public method whose name starts with "get", "is" or "has"

*// src/AppBundle/Entity/Author.php*

*// ...*

use Symfony\Component\Validator\Constraints as Assert;

class Author

{

*/\*\**

*\* @Assert\IsTrue(message = "The password cannot match your first name")*

*\*/*

public function isPasswordLegal()

{

return $this->firstName !== $this->password;

}

}

## Built-in Field Types

<http://symfony.com/doc/current/forms.html#built-in-field-types>

### **Field Type Options**[**¶**](http://symfony.com/doc/current/forms.html#field-type-options)

Each field type has a number of options that can be used to configure it. For example, the dueDatefield is currently being rendered as 3 select boxes. However, the [DateType](http://symfony.com/doc/current/reference/forms/types/date.html) can be configured to be rendered as a single text box (where the user would enter the date as a string in the box):

->add('dueDate', DateType::class, array('widget' => 'single\_text'))

*The required Option (only for client side)*

->add('dueDate', DateType::class, array(

'widget' => 'single\_text',

'required' => false

))

*The label Option*

->add('dueDate', DateType::class, array(

'widget' => 'single\_text',

'label' => 'Due Date',

))

disable the label

->add('dueDate', DateType::class, array(

'widget' => 'single\_text',

'label' => false,

))

## Field Type Guessing

Thanks to the class metadata symphony can guess the type of the field

public function newAction()

{

$task = new Task();

$form = $this->createFormBuilder($task)

->add('task')

->add('dueDate', null, array('widget' => 'single\_text'))

->add('save', SubmitType::class)

->getForm();

}

required

The required option can be guessed based on the validation rules (i.e. is the field NotBlank or NotNull) or the Doctrine metadata (i.e. is the field nullable). This is very useful, as your client-side validation will automatically match your validation rules.

max\_length

If the field is some sort of text field, then the max\_length option can be guessed from the validation constraints (if Length or Range is used) or from the Doctrine metadata (via the field's length).

## Creating Form Classes

*// src/AppBundle/Form/TaskType.php*

namespace AppBundle\Form;

use Symfony\Component\Form\AbstractType;

use Symfony\Component\Form\FormBuilderInterface;

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

class TaskType extends AbstractType

{

public function buildForm(FormBuilderInterface $builder, array $options)

{

$builder

->add('task')

->add('dueDate', null, array('widget' => 'single\_text'))

->add('save', SubmitType::class)

;

}

}

*// src/AppBundle/Controller/DefaultController.php*

use AppBundle\Form\TaskType;

public function newAction()

{

$task = ...;

$form = $this->createForm(TaskType::class, $task);

*// ...*

}

In cases where you need extra fields in the form (for example: a "do you agree with these terms" checkbox) that will not be mapped to the underlying object, you need to set the mapped option to false:

use Symfony\Component\Form\FormBuilderInterface;

public function buildForm(FormBuilderInterface $builder, array $options)

{

$builder

->add('task')

->add('dueDate', null, array('mapped' => false))

->add('save', SubmitType::class)

;

}

Additionally, if there are any fields on the form that aren't included in the submitted data, those fields will be explicitly set to null.

**Accessing and modifying data from the form**

The field data can be accessed in a controller with:

$form->get('dueDate')->getData();

In addition, the data of an unmapped field can also be modified directly:

$form->get('dueDate')->setData(new \DateTime());

# **Security**

<http://symfony.com/doc/current/security.html>

## 1) Initial security.yml Setup (Authentication)[¶](http://symfony.com/doc/current/security.html#initial-security-yml-setup-authentication)

*# app/config/security.yml*

security:

providers:

in\_memory:

memory: ~

firewalls:

dev:

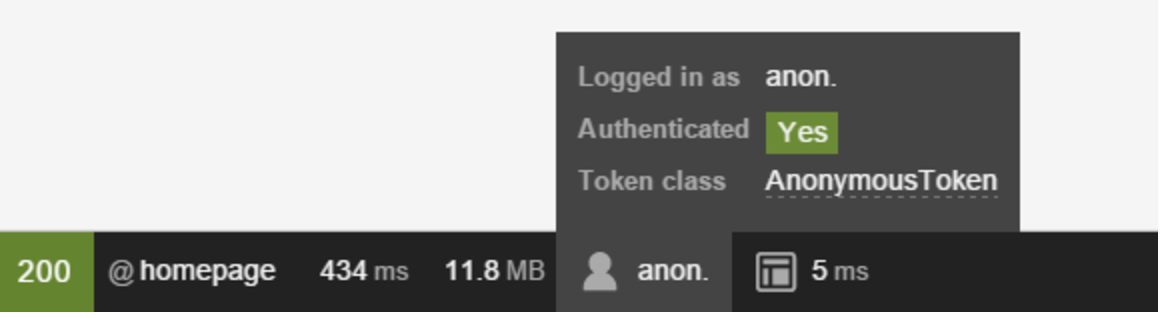
pattern: ^/(\_(profiler|wdt)|css|images|js)/

security: false

default:

anonymous: ~

The firewalls key is the heart of your security configuration



# **SecurityBundle Configuration ("security")**

[Security Configuration Reference](http://symfony.com/doc/current/reference/configuration/security.html)

<http://symfony.com/doc/current/reference/configuration/security.html>

### **A) Configuring how your Users will Authenticate**

http\_basic

*# app/config/security.yml*

security:

*# ...*

firewalls:

*# ...*

default:

anonymous: ~

http\_basic: ~

add an access\_control entry to security.yml.

Only admin can access to /admin

*# app/config/security.yml*

security:

*# ...*

firewalls:

*# ...*

default:

*# ...*

access\_control:

*# require ROLE\_ADMIN for /admin\**

- { path: ^/admin, roles: ROLE\_ADMIN }

### **B) Configuring how Users are Loaded**

configure Symfony to load hardcoded users directly from the security.yml

*# app/config/security.yml*

security:

providers:

in\_memory:

memory:

users:

ryan:

password: ryanpass

roles: 'ROLE\_USER'

admin:

password: kitten

roles: 'ROLE\_ADMIN'

*# ...*

### **Encoders**

Try to login using username admin and password kitten. You should see an error!

*No encoder has been configured for account "Symfony\Component\Security\Core\User\User"*

To fix this, add an encoders key:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | *# app/config/security.yml*  security:  *# ...*  encoders:  Symfony\Component\Security\Core\User\User: plaintext  *# ...* |

If you try to login as ryan it would be denied.

### **C) Encoding the User's Password**

*# app/config/security.yml*

security:

*# ...*

encoders:

Symfony\Component\Security\Core\User\User:

algorithm: bcrypt

cost: 12

Command to encode passwords

php bin/console security:encode-password

*# app/config/security.yml*

security:

*# ...*

providers:

in\_memory:

memory:

users:

ryan:

password: $2a$12$LCY0MefVIEc3TYPHV9SNnuzOfyr2p/AXIGoQJEDs4am4JwhNz/jli

roles: 'ROLE\_USER'

admin:

password: $2a$12$cyTWeE9kpq1PjqKFiWUZFuCRPwVyAZwm4XzMZ1qPUFl7/flCM3V0G

roles: 'ROLE\_ADMIN'

## 2) Denying Access, Roles and other Authorization

The process of authorization has two different sides:

1. The user receives a specific set of roles when logging in (e.g. ROLE\_ADMIN).
2. You add code so that a resource (e.g. URL, controller) requires a specific "attribute" (most commonly a role like ROLE\_ADMIN) in order to be accessed.

### **Roles**

All roles you assign to a user **must** begin with the ROLE\_ prefix

For example, if you need to start limiting access to the blog admin section of your website, you could protect that section using a ROLE\_BLOG\_ADMIN

Make sure every user has at least one role, or your user will look like they're not authenticated. A common convention is to give every user ROLE\_USER.

### **Add Code to Deny Access**

There are **two** ways to deny access to something:

1. [access\_control in security.yml](http://symfony.com/doc/current/security.html#security-authorization-access-control) allows you to protect URL patterns (e.g. /admin/\*). This is easy, but less flexible;
2. [in your code via the security.authorization\_checker service](http://symfony.com/doc/current/security.html#security-securing-controller).

#### **Securing URL patterns (access\_control)**

anything matching the regular expression ^/admin requires the ROLE\_ADMINrole

*# app/config/security.yml*

security:

*# ...*

firewalls:

*# ...*

default:

*# ...*

access\_control:

*# require ROLE\_ADMIN for /admin\**

- { path: ^/admin, roles: ROLE\_ADMIN }

*# app/config/security.yml*

security:

*# ...*

access\_control:

- { path: ^/admin/users, roles: ROLE\_SUPER\_ADMIN }

- { path: ^/admin, roles: ROLE\_ADMIN }

#### **How Does the Security access\_control Work?**

<http://symfony.com/doc/current/security/access_control.html>

#### **Securing Controllers and other Code**

*// ...*

public function helloAction($name)

{

*// The second parameter is used to specify on what object the role is tested.*

$this->denyAccessUnlessGranted('ROLE\_ADMIN', null, 'Unable to access this page!');

*// Old way :*

*// if (false === $this->get('security.authorization\_checker')->isGranted('ROLE\_ADMIN')) {*

*// throw $this->createAccessDeniedException('Unable to access this page!');*

*// }*

*// ...*

}

#### **SensioFrameworkExtraBundle**

<https://symfony.com/doc/current/bundles/SensioFrameworkExtraBundle/index.html>

Thanks to the SensioFrameworkExtraBundle, you can also secure your controller using annotations:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | *// ...*  use Sensio\Bundle\FrameworkExtraBundle\Configuration\Security;  */\*\**  *\* @Security("has\_role('ROLE\_ADMIN')")*  *\*/*  public function helloAction($name)  {  *// ...*  } |

#### **Access Control in Templates**[**¶**](http://symfony.com/doc/current/security.html#access-control-in-templates)

If you want to check if the current user has a role inside a template, use the built-in is\_granted()helper function:

|  |  |
| --- | --- |
| 1  2  3 | {% if is\_granted('ROLE\_ADMIN') %}  <a href="...">Delete</a>  {% endif %} |

### **Checking to see if a User is Logged In (IS\_AUTHENTICATED\_FULLY)**

only want to check if a user is logged in (you don't care about roles

*// ...*

public function helloAction($name)

{

if (!$this->get('security.authorization\_checker')->isGranted('IS\_AUTHENTICATED\_FULLY')) {

throw $this->createAccessDeniedException();

}

*// ...*

}

You can of course also use this in access\_control

*# app/config/security.yml*

security:

*# ...*

access\_control:

*#*

- { path: ^/internal, roles: IS\_AUTHENTICATED\_ANONYMOUSLY, ips: [127.0.0.1, ::1] }

- { path: ^/internal, roles: ROLE\_NO\_ACCESS }

IS\_AUTHENTICATED\_FULLY isn't a role, but it kind of acts like one, and every user that has successfully logged in will have this. In fact, there are three special attributes like this:

* IS\_AUTHENTICATED\_REMEMBERED: All logged in users have this, even if they are logged in because of a "remember me cookie". Even if you don't use the [remember me functionality](http://symfony.com/doc/current/security/remember_me.html), you can use this to check if the user is logged in.
* IS\_AUTHENTICATED\_FULLY: This is similar to IS\_AUTHENTICATED\_REMEMBERED, but stronger. Users who are logged in only because of a "remember me cookie" will have IS\_AUTHENTICATED\_REMEMBERED but will not have IS\_AUTHENTICATED\_FULLY.
* IS\_AUTHENTICATED\_ANONYMOUSLY: All users (even anonymous ones) have this - this is useful when whitelisting URLs to guarantee access - some details are in [How Does the Security access\_control Work?](http://symfony.com/doc/current/security/access_control.html).

You can also use expressions inside your templates:

{% if is\_granted(expression(

'"ROLE\_ADMIN" in roles or (user and user.isSuperAdmin())'

)) %}

<a href="...">Delete</a>

{% endif %}

### **Access Control Lists (ACLs): Securing individual Database Objects**[**¶**](http://symfony.com/doc/current/security.html#access-control-lists-acls-securing-individual-database-objects)

Imagine you are designing a blog where users can comment on your posts. You also want a user to be able to edit their own comments, but not those of other users. Also, as the admin user, you yourself want to be able to edit all comments.

To accomplish this you have 2 options:

* [Voters](http://symfony.com/doc/current/security/voters.html) allow you to write own business logic (e.g. the user can edit this post because they were the creator) to determine access. You'll probably want this option - it's flexible enough to solve the above situation.
* [ACLs](http://symfony.com/doc/current/security/acl.html) allow you to create a database structure where you can assign any arbitrary user anyaccess (e.g. EDIT, VIEW)  to any object in your system. Use this if you need an admin user to be able to grant customized access across your system via some admin interface.

# **How to Use Voters to Check User Permissions**

<http://symfony.com/doc/current/security/voters.html>

# **Retrieving the User Object**

use Symfony\Component\Security\Core\User\UserInterface;

public function indexAction(UserInterface $user)

{

if (!$this->get('security.authorization\_checker')->isGranted('IS\_AUTHENTICATED\_FULLY')) {

throw $this->createAccessDeniedException();

}

*// the above is a shortcut for this*

$user = $this->get('security.token\_storage')->getToken()->getUser();

return new Response('Well hi there '.$user->getFirstName());

}

### **Always Check if the User is Logged In**

public function indexAction(UserInterface $user = null)

{

*// $user is null when not logged-in or anon.*

}

*// yay! Use this to see if the user is logged in*

if (!$this->get('security.authorization\_checker')->isGranted('IS\_AUTHENTICATED\_FULLY')) {

throw $this->createAccessDeniedException();

}

*// boo :(. Never check for the User object to see if they're logged in*

if ($this->getUser()) {

}

### **Retrieving the User in a Template**

{% if is\_granted('IS\_AUTHENTICATED\_FULLY') %}

<p>Username: {{ app.user.username }}</p>

{% endif %}

## Logging Out need to provide a provider??

*# app/config/security.yml*

security:

*# ...*

firewalls:

secured\_area:

*# ...*

logout:

path: /logout

target: /

## Hierarchical Roles

*# app/config/security.yml*

security:

*# ...*

role\_hierarchy:

ROLE\_ADMIN: ROLE\_USER

ROLE\_SUPER\_ADMIN: [ROLE\_ADMIN, ROLE\_ALLOWED\_TO\_SWITCH]

# **How to Build a Traditional Login Form**

<http://symfony.com/doc/current/security/form_login_setup.html>

First, enable form login under your firewall:

*# app/config/security.yml*

security:

*# ...*

firewalls:

main:

anonymous: ~

form\_login:

login\_path: login

check\_path: login

The login\_path and check\_path can also be route names (but cannot have mandatory wildcards - e.g. /login/{foo}

## Form Login Configuration

<http://symfony.com/doc/current/reference/configuration/security.html#reference-security-firewall-form-login>

## Create a new SecurityController

First, create a new SecurityController inside a bundle:

*// src/AppBundle/Controller/SecurityController.php*

*// ...*

use Symfony\Component\HttpFoundation\Request;

use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;

class SecurityController extends Controller

{

*/\*\**

*\* @Route("/login", name="login")*

*\*/*

public function loginAction(Request $request)

{

$authenticationUtils = $this->get('security.authentication\_utils');

*// get the login error if there is one*

$error = $authenticationUtils->getLastAuthenticationError();

*// last username entered by the user*

$lastUsername = $authenticationUtils->getLastUsername();

return $this->render('security/login.html.twig', array(

'last\_username' => $lastUsername,

'error' => $error,

));

}

}

## Create the template

*{# app/Resources/views/security/login.html.twig #}*

*{# ... you will probably extend your base template, like base.html.twig #}*

{% if error %}

<div>{{ error.messageKey|trans(error.messageData, 'security') }}</div>

{% endif %}

<form action="{{ path('login') }}" method="post">

<label for="username">Username:</label>

<input type="text" id="username" name="\_username" value="{{ last\_username }}" />

<label for="password">Password:</label>

<input type="password" id="password" name="\_password" />

*{#*

*If you want to control the URL the user*

*is redirected to on success (more details below)*

*<input type="hidden" name="\_target\_path" value="/account" />*

*#}*

<button type="submit">login</button>

</form>

The username field has the name \_username and the password field has the name \_password

all of this can be configured under the form\_login key. See [Form Login Configuration](http://symfony.com/doc/current/reference/configuration/security.html#reference-security-firewall-form-login) for more details.

## Redirecting after Success[¶](http://symfony.com/doc/current/security/form_login_setup.html#redirecting-after-success)

If the submitted credentials are correct, the user will be redirected to the original page that was requested (e.g. /admin/foo). If the user originally went straight to the login page, they'll be redirected to the homepage. This can all be customized, allowing you to, for example, redirect the user to a specific URL.

## Avoid Common Pitfalls

### **1. Create the Correct Routes**

First, be sure that you've defined the /login route correctly and that it corresponds to the login\_path and check\_path config values

### **2. Be Sure the Login Page Isn't Secure (Redirect Loop!)**

For example, the following configuration - which requires the ROLE\_ADMIN role for all URLs (including the /login URL), will cause a redirect loop:

*# app/config/security.yml*

*# ...*

access\_control:

- { path: ^/, roles: ROLE\_ADMIN }

Adding an access control that matches /login/\* and requires no authentication fixes the problem:

*# app/config/security.yml*

*# ...*

access\_control:

- { path: ^/login, roles: IS\_AUTHENTICATED\_ANONYMOUSLY }

- { path: ^/, roles: ROLE\_ADMIN }

# **Using CSRF Protection in the Login Form**

<http://symfony.com/doc/current/security/csrf_in_login_form.html>

# **How to Manually Encode a Password**

<http://symfony.com/doc/current/security/password_encoding.html>

# **How to Load Security Users from the Database (the Entity Provider)**

<http://symfony.com/doc/current/security/entity_provider.html>

## 1) Create your User Entity

*// src/AppBundle/Entity/User.php*

namespace AppBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

use Symfony\Component\Security\Core\User\UserInterface;

*/\*\**

*\* @ORM\Table(name="app\_users")*

*\* @ORM\Entity(repositoryClass="AppBundle\Repository\UserRepository")*

*\*/*

class User implements UserInterface, \Serializable

{

*/\*\**

*\* @ORM\Column(type="integer")*

*\* @ORM\Id*

*\* @ORM\GeneratedValue(strategy="AUTO")*

*\*/*

private $id;

*/\*\**

*\* @ORM\Column(type="string", length=25, unique=true)*

*\*/*

private $username;

*/\*\**

*\* @ORM\Column(type="string", length=64)*

*\*/*

private $password;

*/\*\**

*\* @ORM\Column(type="string", length=60, unique=true)*

*\*/*

private $email;

*/\*\**

*\* @ORM\Column(name="is\_active", type="boolean")*

*\*/*

private $isActive;

public function \_\_construct()

{

$this->isActive = true;

*// may not be needed, see section on salt below*

*// $this->salt = md5(uniqid(null, true));*

}

public function getUsername()

{

return $this->username;

}

public function getSalt()

{

*// you \*may\* need a real salt depending on your encoder*

*// see section on salt below*

return null;

}

public function getPassword()

{

return $this->password;

}

public function getRoles()

{

return array('ROLE\_USER');

}

public function eraseCredentials()

{

}

*/\*\* @see \Serializable::serialize() \*/*

public function serialize()

{

return serialize(array(

$this->id,

$this->username,

$this->password,

*// see section on salt below*

*// $this->salt,*

));

}

*/\*\* @see \Serializable::unserialize() \*/*

public function unserialize($serialized)

{

list (

$this->id,

$this->username,

$this->password,

*// see section on salt below*

*// $this->salt*

) = unserialize($serialized);

}

}

## 2) Configure Security to load from your Entity

*# app/config/security.yml*

security:

encoders:

AppBundle\Entity\User:

algorithm: bcrypt

*# ...*

providers:

our\_db\_provider:

entity:

class: AppBundle:User

property: username

*# if you're using multiple entity managers*

*# manager\_name: customer*

firewalls:

main:

pattern: ^/

http\_basic: ~

provider: our\_db\_provider

*# ...*

# **Login using username or password**

To do this, make your UserRepository implement a special [UserLoaderInterface](http://api.symfony.com/3.2/Symfony/Bridge/Doctrine/Security/User/UserLoaderInterface.html). This interface only requires one method: loadUserByUsername($username):

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | *// src/AppBundle/Repository/UserRepository.php*  namespace AppBundle\Repository;  use Symfony\Bridge\Doctrine\Security\User\UserLoaderInterface;  use Doctrine\ORM\EntityRepository;  class UserRepository extends EntityRepository implements UserLoaderInterface  {  public function loadUserByUsername($username)  {  return $this->createQueryBuilder('u')  ->where('u.username = :username OR u.email = :email')  ->setParameter('username', $username)  ->setParameter('email', $username)  ->getQuery()  ->getOneOrNullResult();  }  } |

To finish this, just remove the property key from the user provider in security.yml:

* [YAML](http://symfony.com/doc/current/security/entity_provider.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | *# app/config/security.yml*  security:  *# ...*  providers:  our\_db\_provider:  entity:  class: AppBundle:User |

# **Login via Google, Facebook or Twitter**

If your application logs users in via a third-party service such as Google, Facebook or Twitter, check out the [HWIOAuthBundle](https://github.com/hwi/HWIOAuthBundle) community bundle.

# **USING AJAX**

**El template Twig**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | {# app/Resources/views/comments/list.html.twig #}  {% extends 'base.html.twig' %}  {% block body %}  <div>  <a href="{{ path('comments\_new') }}">Nuevo</a>  </div>    {# la tabla la envio a otro twig para poder recargar la tabla y reutilizar ese código luego de la llamda ajax #}  <div id="table-wrapper">  {% include 'comments/render-table.html.twig' %}  </div>    <script src="{{ asset('public/assets/js/jquery.min.js') }}"></script>    {# ver la nstalacion de este bundle https://symfony.com/doc/master/bundles/FOSJsRoutingBundle/installation.html #}  <script src="{{ asset('bundles/fosjsrouting/js/router.js') }}"></script>  <script src="{{ path('fos\_js\_routing\_js', { callback: 'fos.Router.setData' }) }}"></script>    {# este es el JS que hace la llamada ajax #}  <script src="{{ asset('public/assets/js/delete\_comment.js') }}"></script>  {% endblock %} |

**La sección de la página que se recarga cada vez qe elimino un comentario**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | {# app/Resources/views/comments/render-table.html.twig #}  <table>  <thead>  <tr>  <th>Fecha</th>  <th>Comentario</th>  <th>Usuario</th>  <th>Acciones</th>  </tr>  </thead>  <tbody>  {% for comment in lista\_comentarios %}  <tr class="{{ cycle(['even', 'odd'], loop.index) }}">  <td>{{ comment.date | date('d-m-Y') }}</td>  <td>{{ comment.text }}</td>  <td>{{ comment.user.name }}</td>  <td class="eliminar">  {% if comment.user.id == app.user.id %}  {# <a href="{{ url('comments\_remove', {'id': comment.id}) }}">Eliminar</a> #}  <a comment-id="{{ comment.id }}" href="#">Eliminar</a>  {% endif %}  </td>  </tr>  {% endfor %}  </tbody>  </table> |

**El JS para la llamada AJAX**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | // web/public/assets/js/delete\_comment.js  $(document).ready(function ()  {  // utilizando de esta manera el evento permite la propagacion del evento  // asi los nuevos elementos también tendran el evento, esto no pasa si usamos  // de esta manera: $('td a').on('click', function (e) {  $(document).on('click', 'td a', function (e) {  e.preventDefault();    console.log($(this).attr('comment-id'));    $.ajax({  method: "POST",  url: Routing.generate('comments\_remove'), // esto lo incluye el FOSJsRoutingBundle  data: {id: $(this).attr('comment-id')},  dataType: 'json',  success: function (data)  {  $('.flash-success').text(data.mensaje); // presento el mensaje  $('#table-wrapper').html(data.lista\_comentarios\_html); // actualizo la tabla  },  error: function (jqXHR, exception)  {  if (jqXHR.status === 405)  {  console.error("METHOD NOT ALLOWED!");  }  }  });  });  }); |

**La accion en el Controlador**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | /\*\*  \* @Route("/comentarios/eliminar", options={"expose"=true}, name="comments\_remove")  \*/  public function removeAction(Request $request)  {  // verificar que solo se puede acceder a este controlador mediante una llamada ajax  if ($request->isXMLHttpRequest()) {  $id = $request->get('id');  $repository = $this->getDoctrine()->getRepository('AppBundle:Comentario');  // eliminar el comentario de la BDD  $comentario = $repository->findOneById($id);  $em = $this->getDoctrine()->getManager();  $em->remove($comentario);  $em->flush();    // obtener la lista de comentarios actualizada  $lista\_comentarios = $repository->findAll();  // Obtener solo el HTML del render no las cabeceras  $lista\_comentarios\_html = $this->render('comments/render-table.html.twig',array(  'lista\_comentarios'=> $lista\_comentarios  ))->getContent();    // Enviar la respuesta codificada como json  return new JsonResponse(array(  'mensaje' => 'Comentario Eliminado!',  'lista\_comentarios\_html' => $lista\_comentarios\_html  )  );  }  return new Response('Acción no permitida!', 400);  } |