# Blog: PHP and Symfony

This document defines a complete walkthrough of creating a **Blog** application with the [Symfony](https://symfony.com/) Framework, from setting up the framework through [authentication](http://symfony.com/doc/current/security.html) module, ending up with creating a **CRUD** around [Doctrine](http://www.doctrine-project.org/) entities.

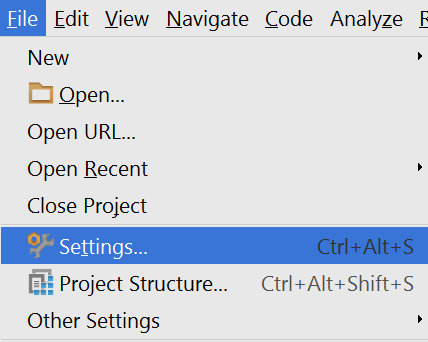
Make sure you have installed [XAMPP](https://www.apachefriends.org/download.html), [HeidiSQL](http://www.heidisql.com/download.php) and added [PHP root folder to the path environment variable](http://php.net/manual/en/faq.installation.php#faq.installation.addtopath).

**Chapters from I to III are for advanced users. There’s a** [**skeleton**](https://softuni.bg/downloads/svn/soft-tech/Sep-2016/Software-Technologies-Oct-2016/03.%20PHP-Blog-Basic-Functionality/php-sever-skeleton.zip) **which you can use and start from chapter IV.**

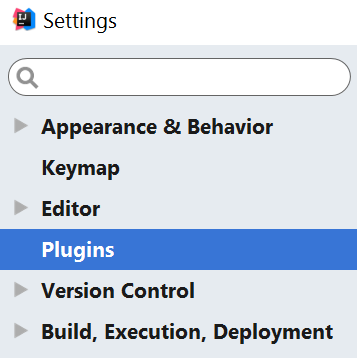
# Set Up Symfony Project

Symfony framework comes with various ways of creating a project, all of them involving the [presence of Symfony project](https://symfony.com/download). The most convenient way is to **create a project via your IDE**. Luckily there are several **plugins** for **PHPStorm** (and the other **IDEA**-based IDE’s) which help developing application with Symfony

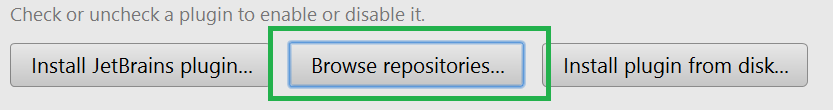
## Install Symfony-related Plugins



-

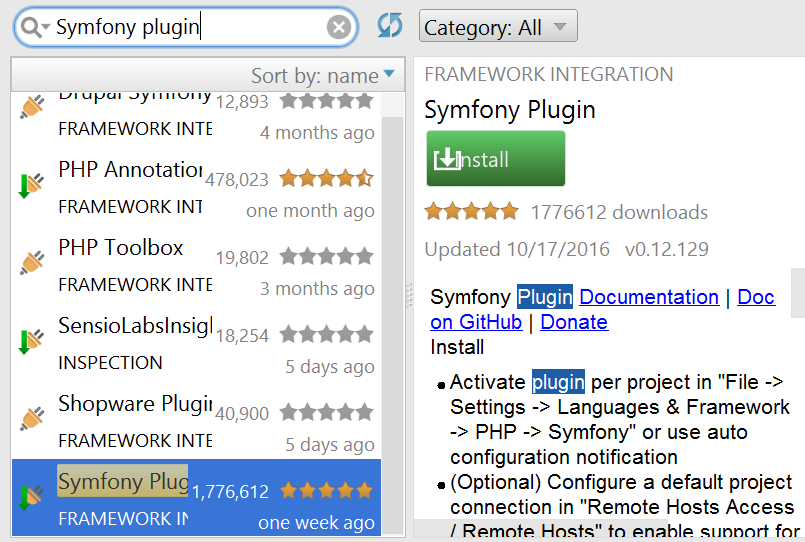


-

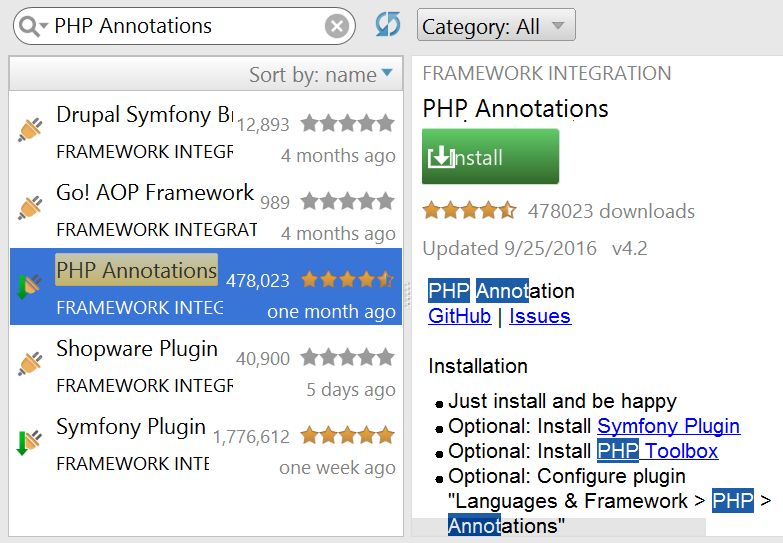


We need to install the following plugins:

1. Symfony Plugin

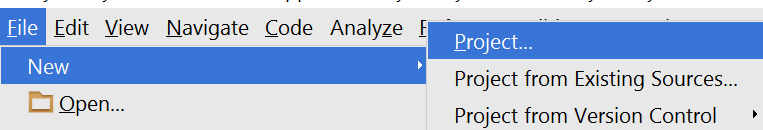


1. PHP Annotations

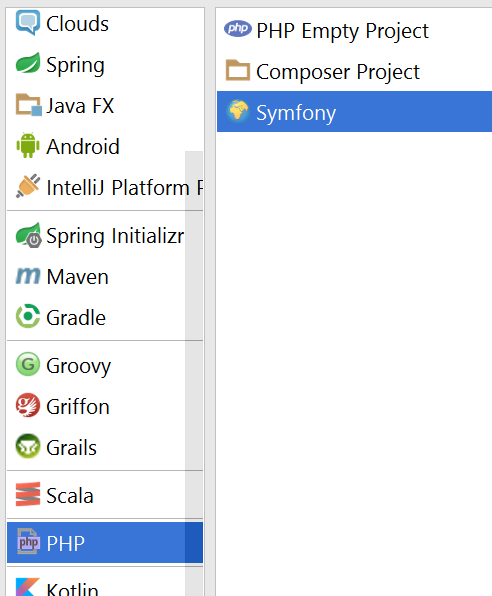


## Create Symfony Project from IDE

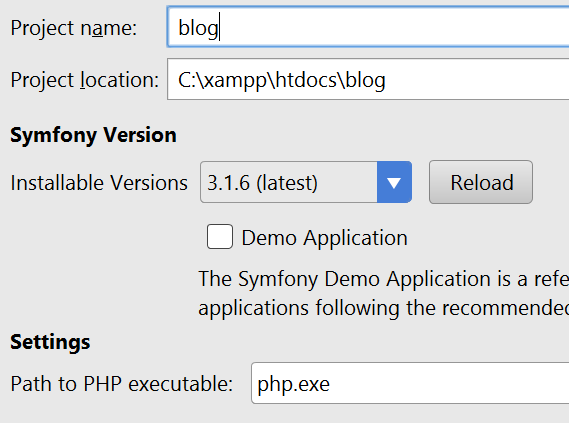
Once you have installed the plugins and restarted the **IDE**, you will have in the **Create Project** context menu either a **PHP subcategory** (IntelliJ) or directly a **Symfony** one (PHPStorm)



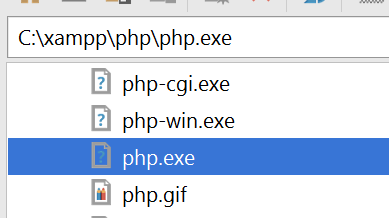
-



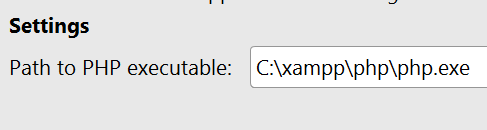
-



We need to specify the **php executable**, which most probably resides in **c:/xampp/php**

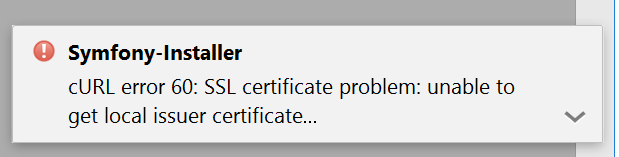


-

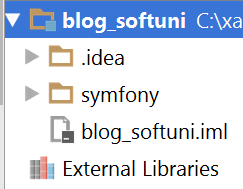


## Check Project Status

If you have received the following error:



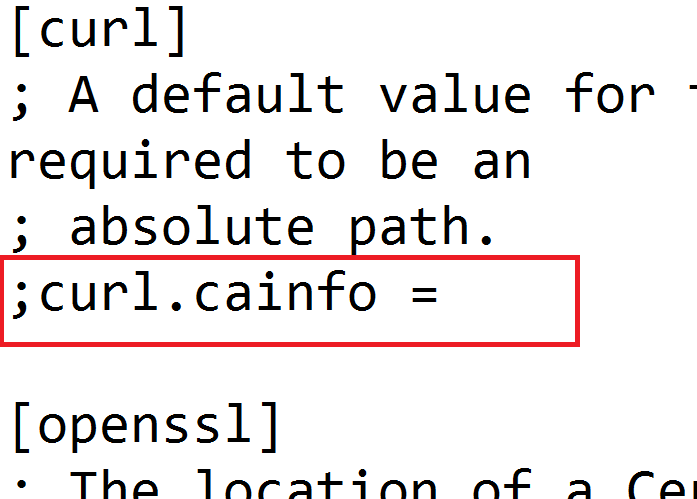
And your project looks like this:



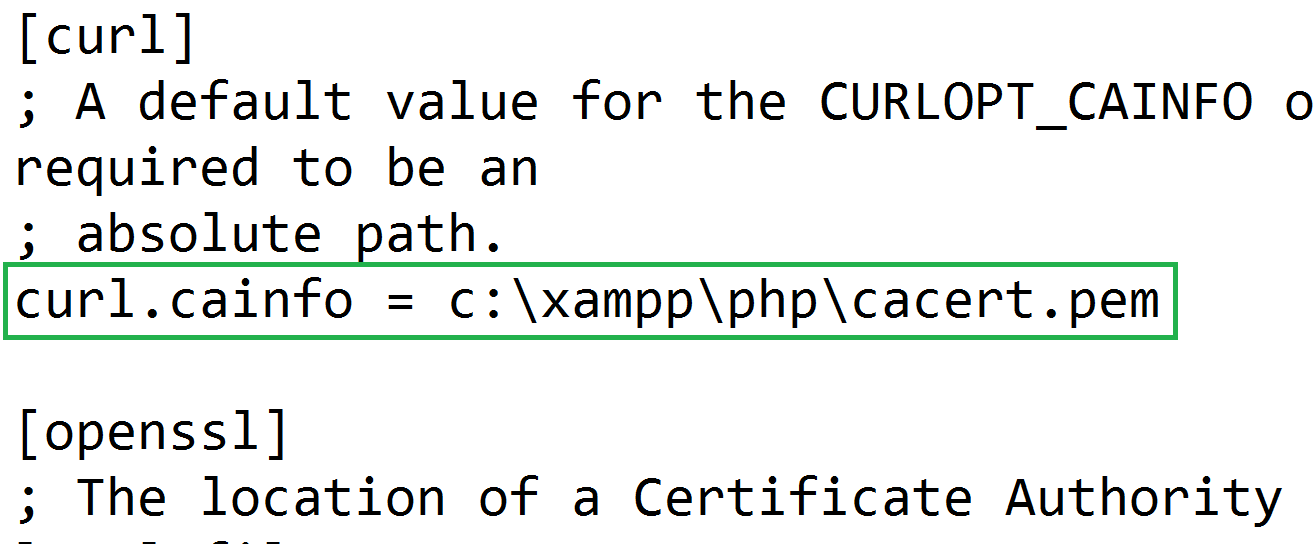
You most probably haven’t created the project properly. This could of possible missing curl.cainfo directive in **php.ini**.

Follow the following [instructions](http://stackoverflow.com/questions/37997669/curl-error-60-ssl-certification-issue-when-attempting-to-use-symfony) **ONLY IF YOU HAVE RECEIVED THE ERROR ABOVE, OTHERWISE SKIP THIS STEP.**

1. Save this file : <https://curl.haxx.se/ca/cacert.pem> in **c:/xampp/php**
2. Edit the **c:/xampp/php/php.ini** file and fine the following line



1. And make it: “curl.cainfo = c:\xampp\php\cacert.pem”

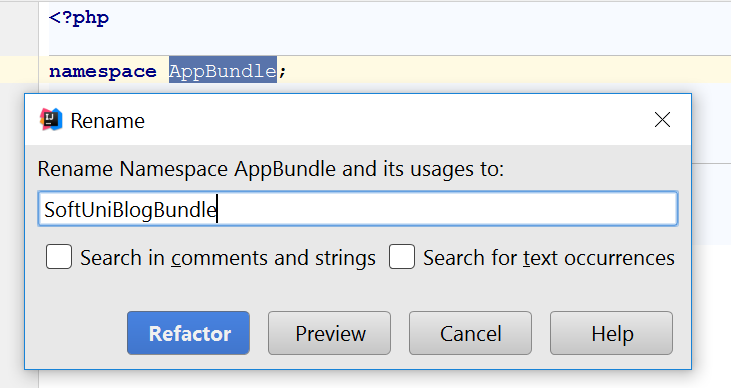


1. Create the project again

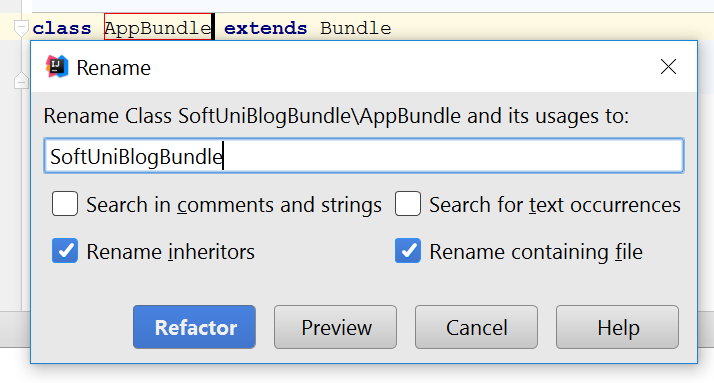
## Rename Default Bundle

The Default bundle located in src folder is called AppBundle. Rename with **SHIFT+F6** the following occurrences to SoftUniBlogBundle:

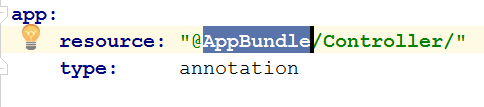
1. src/AppBundle folder
2. src/AppBundle/AppBundle.php
3. The namespace directive in src/AppBundle/AppBundle.php



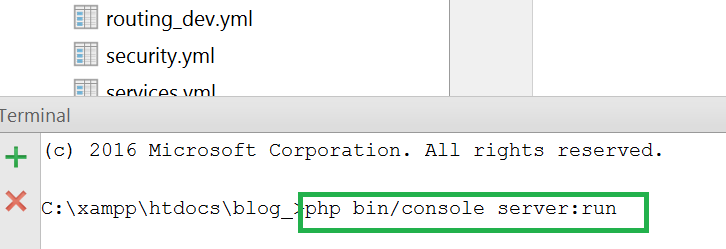
1. The classname in src/AppBundle/AppBundle.php



Change the occurrence in app/config/routing.yml to SoftUniBlogBundle too:



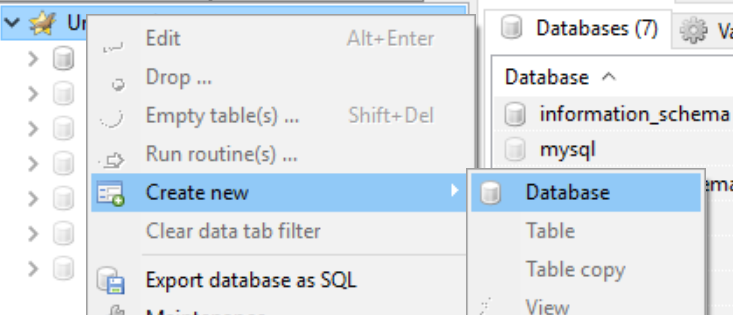
Start the server by running in the project folder the following command



And on see the result in <http://localhost:8000> ☺

## Create Database

Open HeidiSQL, connect to the MySQL instance and create a database named “blog”



And change the database name in app/config/parameters.yml to “blog”

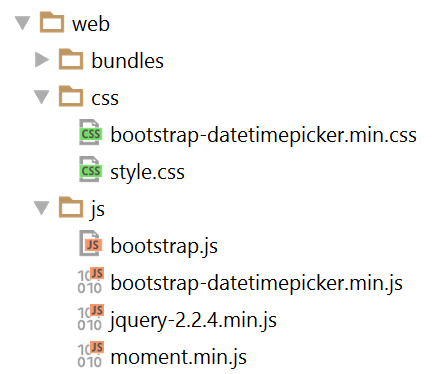


## Setup Layout

We will need a base layout for all of our templates. As we are using **Bootstrap**, we will need its css included in all pages, and the related scripts too. We can download the sample **blog design skeleton** from [here](https://softuni.bg/downloads/svn/soft-tech/Sep-2016/Software-Technologies-Oct-2016/03.%20PHP-Blog-Basic-Functionality/blog%20design.zip), where part of our **JavaScript** and **CSS** is included. In addition, we will need

1. [Bootstrap Date Time picker](http://www.malot.fr/bootstrap-datetimepicker/) for choosing dates in our forms
2. [Moment JS](http://momentjs.com/) for validating dates

All of our styles and scripts we need to include to our project. Let’s create two folders in “web” folder called respectively “css” and “js”, place the needed scripts and styles there, resulting with the following structure:



Then we need to use this styles and script setting up a base layout in app/resources/views/base.html.twig.

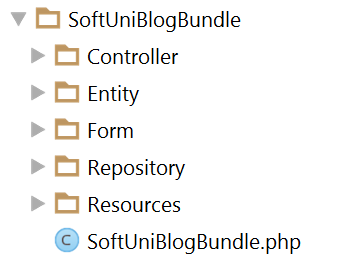
Setup a base layout as you wish or use the following one:

|  |
| --- |
| *{#  This is the base template used as the application layout which contains the  common elements and decorates all the other templates.  See http://symfony.com/doc/current/book/templating.html#template-inheritance-and-layouts #}* <!DOCTYPE **html**> <**html lang="en-US"**> <**head**>  <**meta charset="UTF-8"**/>  <**meta name="viewport" content="width=device-width, initial-scale=1"**/>  <**title**>{% **block** title %}SoftUni Blog{% **endblock** %}</**title**>  {% **block** stylesheets %}  <**link rel="stylesheet" href="**{{ asset(**'css/style.css'**) }}**"**>  <**link rel="stylesheet" href="**{{ asset(**'css/bootstrap-datetimepicker.min.css'**) }}**"**>  {% **endblock** %}  <**link rel="icon" type="image/x-icon" href="**{{ asset(**'favicon.ico'**) }}**"**/> </**head**>  <**body id="**{% **block** body\_id %}{% **endblock** %}**"**>  {% **block** header %}  <**header**>  <**div class="navbar navbar-default navbar-static-top" role="navigation"**>  <**div class="container"**>  <**div class="navbar-header"**>  <**a href="**{{ path(**'blog\_index'**) }}**" class="navbar-brand"**>SOFTUNI BLOG</**a**>  {% **if** app.user %}  <**a href="**{{ path(**'article\_create'**) }}**" class="navbar-brand"**>  Create Article  </**a**>  {% **endif** %}  <**button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navbar-collapse"**>  <**span class="icon-bar"**></**span**>  <**span class="icon-bar"**></**span**>  <**span class="icon-bar"**></**span**>  </**button**>  </**div**>  <**div class="navbar-collapse collapse"**>  <**ul class="nav navbar-nav navbar-right"**>  {% **if** app.user %}  <**li**>  <**a href="**{{ path(**'user\_profile'**) }}**"**>  My Profile  </**a**>  </**li**>  <**li**>  <**a href="**{{ path(**'security\_logout'**) }}**"**>  Logout  </**a**>  </**li**>  {% **else** %}  <**li**>  <**a href="**{{ path(**'user\_register'**) }}**"**>  REGISTER  </**a**>  </**li**>  <**li**>  <**a href="**{{ path(**'security\_login'**) }}**"**>  LOGIN  </**a**>  </**li**>  {% **endif** %}  </**ul**>  </**div**>  </**div**>  </**div**>  </**header**> {% **endblock** %}  <**div class="container body-container"**>  {% **block** body %}  <**div class="row"**>  <**div id="main" class="col-sm-9"**>  {% **block** main %}{% **endblock** %}  </**div**>  </**div**>  {% **endblock** %} </**div**>  {% **block** footer %}  <**footer**>  <**div class="container modal-footer"**>  <**p**>**&copy;** 2016 - Software University Foundation</**p**>  </**div**>  </**footer**> {% **endblock** %}  {% **block** javascripts %}  <**script src="**{{ asset(**'js/jquery-2.2.4.min.js'**) }}**"**></**script**>  <**script src="**{{ asset(**'js/moment.min.js'**) }}**"**></**script**>  <**script src="**{{ asset(**'js/bootstrap.js'**) }}**"**></**script**>  <**script src="**{{ asset(**'js/bootstrap-datetimepicker.min.js'**) }}**"**></**script**> {% **endblock** %}  </**body**> </**html**> |

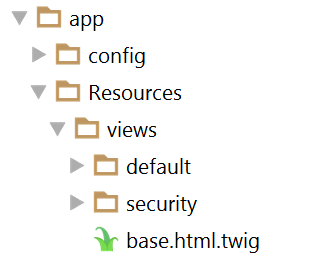
# Symfony Base Project Overview

Symfony is a modular enterprise web-framework, which comes with a solid vendor support, **bundle** system, **enterprise** mechanisms and is most-suitable for **MVC** architecture.

Initially the project comes with a main [bundle](http://symfony.com/doc/current/bundles.html), which can be threat as a plugin later. A **bundle** often has **Controller**, **Entities** and related components (e.g. Repositories, Forms, Commands…)

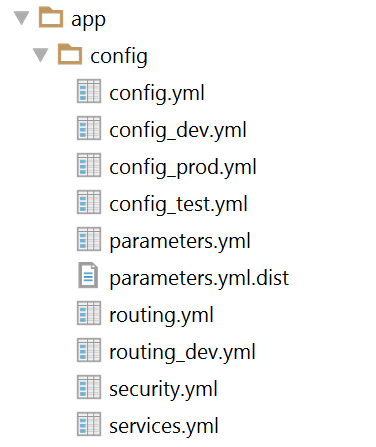


Standard templates (views) reside in the application folder (app) and are usually separated in folder named after the **controller names**.



The de-facto standard **View Engine** in Symfony is [Twig](http://twig.sensiolabs.org/).

The base configuration of the project is placed in app/config, where a configuration files for the [Doctrine](http://www.doctrine-project.org/) connection are defined, [Security](http://symfony.com/doc/current/security.html) management, [Routing](http://symfony.com/doc/current/routing.html) rules, registering [Services](http://symfony.com/doc/current/service_container.html) and so forth.



The parameters.yml.dist file is very important to contain the **same** keys as in parameters.yml, because installing new bundle will delete unused pairs.

# User Authentication

Symfony has very powerful **security** management system, where the common work for checking user **permissions and dispatching the request** is well abstracted, yet the configuration done once could be confusing. In the walkthrough below we will setup a **registration and login process** and accessing **secured** content.

## Creating User Entity

Our users should be stored in the database; this means we need users table. As tables are represented as objects in the **Object/Relation Mapping** paradigm, we need to create an **object representing that table**. The classes(objects) which represent tables are called **Models** and **Entities**.

In the de-facto, standard **ORM** in Symfony, called **Doctrine**, they call these objects **Entities.**

Let’s define our rules for a user:

* Should have a unique login name, let’s say email
* Should have a password
* Should have a full name, let’s say fullName

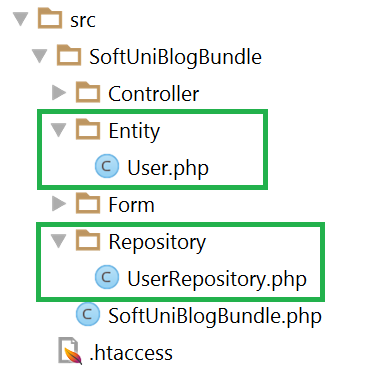
Doctrine comes with a [handy console tool](http://symfony.com/doc/current/doctrine/console.html) for managing the database, as well as creating entities. Let’s use the last one to create the entity called User with the entity generation wizard. To do this, we need to open a terminal window in the project root directory and type the following command:

|  |
| --- |
| php bin/console doctrine:generate:entity |

This will prompt us to enter an entity name. Entities are prefixed with the bundle they should belong to, and let’s imagine our bundle is called SoftUniBlogBundle (the default name is AppBundle). Type SoftUniBlogBundle:User (or AppBundle:User, if you bundle is called AppBundle).

Afterwards it will prompt us for the properties (fields) of the User object. As we have said above, it will have an email, password and a fullName, all of them are text fields (strings). Only the email should be unique, so when you are prompt for uniqueness there, type “true” instead of just clicking enter (which defaults to false)

When the last field (fullName) is created and you are prompt for another one, just click enter to exit the wizard. The result will be of created User entity and corresponding UserRepository.



## Setting Up Security Configuration

As we have said, Symfony comes with a couple of configuration files, one of which is called security.yml. We have to tell **how the password will be encrypted** and on which entity; **where database users will be stored** (corresponding entity) and which **will be the username field** (e.g. email, username, etc…); **where will be located the login form** (route name) and where this **login form will post to**;

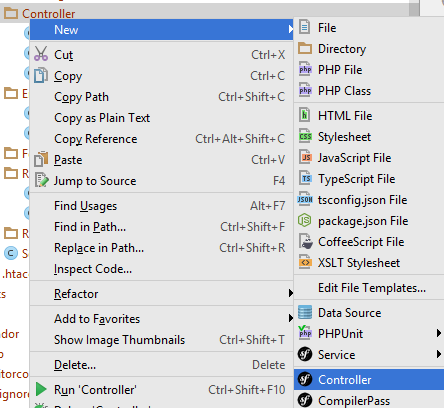
The following code is already configured security.yml assuming that the bundle is called SoftUniBlogBundle, the entity is called User, the username field is called email, the login form will reside and post to “security\_login” and after successful login will be located to “blog\_index”

|  |
| --- |
| **security:  encoders:** *# Our user class and the algorithm we'll use to encode passwords  # http://symfony.com/doc/current/book/security.html#encoding-the-user-s-password* **SoftUniBlogBundle\Entity\User:** bcrypt   **providers:** *# in this example, users are stored via Doctrine in the database  # To see the users at src/AppBundle/DataFixtures/ORM/LoadFixtures.php  # To load users from somewhere else: http://symfony.com/doc/current/cookbook/security/custom\_provider.html* **database\_users:  entity:** { **class:** SoftUniBlogBundle:User, **property:** email }   *# http://symfony.com/doc/current/book/security.html#firewalls-authentication* **firewalls:  secured\_area:** *# this firewall applies to all URLs* **pattern:** ^/   *# but the firewall does not require login on every page  # denying access is done in access\_control or in your controllers* **anonymous:** true   *# This allows the user to login by submitting a username and password  # Reference: http://symfony.com/doc/current/cookbook/security/form\_login\_setup.html* **form\_login:** *# The route name that the login form submits to* **check\_path:** security\_login  *# The name of the route where the login form lives  # When the user tries to access a protected page, they are redirected here* **login\_path:** security\_login  *# Secure the login form against CSRF  # Reference: http://symfony.com/doc/current/cookbook/security/csrf\_in\_login\_form.html* **csrf\_token\_generator:** security.csrf.token\_manager   **logout:** *# The route name the user can go to in order to logout* **path:** security\_logout  *# The name of the route to redirect to after logging out* **target:** blog\_index    **access\_control:** *# this is a catch-all for the admin area  # additional security lives in the controllers # - { path: '^/(%locale%)/admin', roles: ROLE\_ADMIN }* |

## Login Form

To create a login form, we need to create a so-called Controller which will catch this **route** (which above we called “security\_login”) and render the View with the login form.

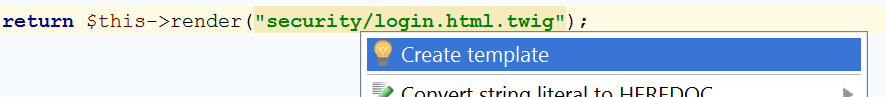
Let’s call our Controller – SecurityController:



Then we need a method (which we will call login()) that listens to that route and renders a view (let’s call it login.html.twig in the **folder security**)



The yellow background color in the view name tells us we don’t have that view created and could prompt us create it by clicking ALT+ENTER ☺



Before messing up with layouts (which we have setup and will use in the next chapters) we will just create a simple login form with no styles.

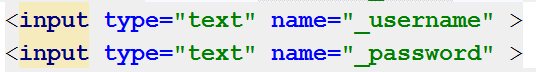
We need to define a <form> tag, which is posting to security\_login route. **Twig**, fortunately, provides a function url() that uses route names and generates URLs from them



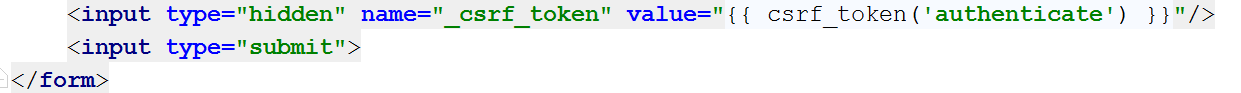
The form is named “**authenticate**” because we will use this name later to generate a [CSRF Token](https://en.wikipedia.org/wiki/Cross-site_request_forgery)

Symfony security requires the **username** (which is **email** in our case) and **password** fields to be named respectively \_username and \_password

We need to define these two text fields (or password field for the password type ☺)



And a field for the CSRF Token using the Twig’s helper method csrf\_token() which accepts the form name.



Now opening <http://localhost:8000/login> should render this login form



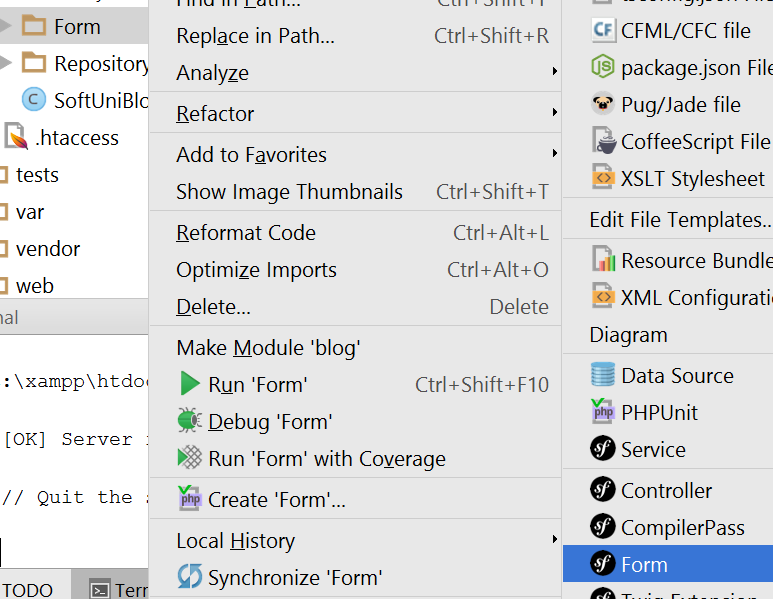
Not the most beautiful login form we have done so far ☺ But still it’s there! ☺

## Register Form

What is a login form without users – nothing. In order to have users, we need a registration form. By analogy, open the already generated DefaultController or create a new one (e.g. UsersController) and an action that listens on “register”

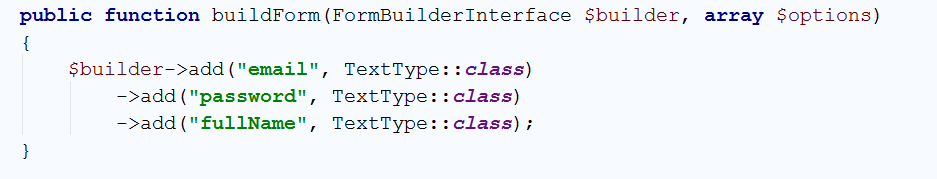
It will render the form the same way, but also needs to handle this form.

In order a form to work with an entity, it needs a corresponding [FormType](http://symfony.com/doc/current/form/data_transformers.html). Before we can continue creating the register action, we need to create a Form Type. Create a folder “**Form**” in src/SoftUniBlogBundle. Then create a Form Type as follows:



Let’s call it UserType.

In the scaffold method “buildForm()” we need to the define pairs – the entity fields and their corresponding types in the form. All of our three fields are text types, so we will use TextType from the Symfony\Component\Form\Extension\Core\Type\TextType namespace.



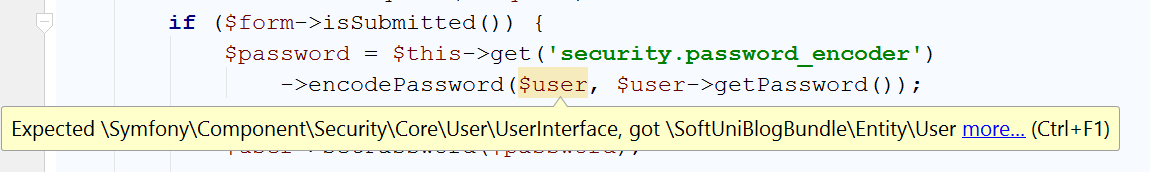
Going back to the controller’s registration method we can now create a form of UserType.



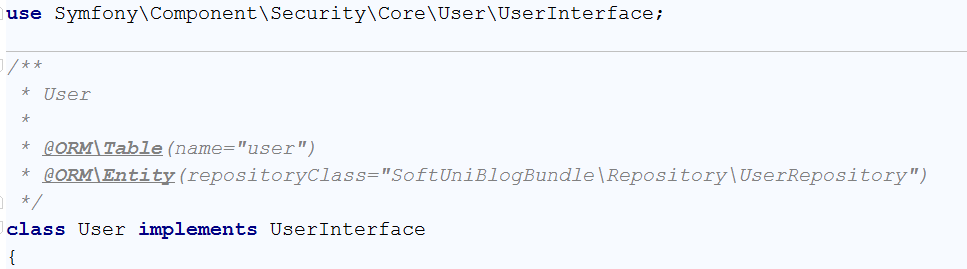
We have said here: Create a form of user type and after it’s submitted fill the $user object.

Then we need to tell the method – once the form is submitted and all the validations are passed (e.g. texts are filled), save the user entity in the database.

There’s one possible problem – the password will go plain into the DB. Luckily, in the security configuration we have registered an encryption provider, so we can use this provider to encode the password and then send it to the database



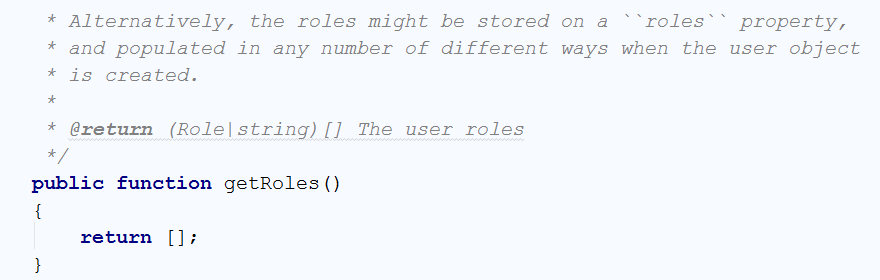
The encoder only works on **UserInterface** objects and our users is not one. What we need is to go to the User entity and make it implements the **UserInterface** interface.



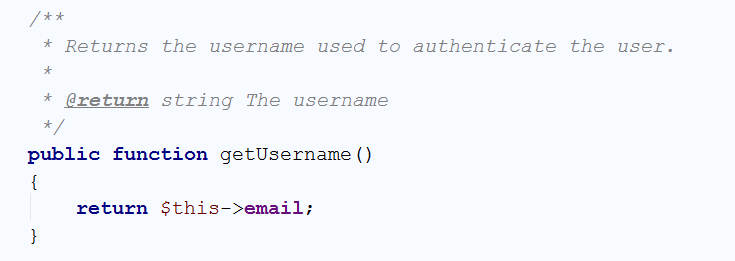
Then implement all of the missing methods with ALT+ENTER.

You can leave most of the blank (auto-generated), but some of them should be filled.

The first method is getRoles(). It should return an array of roles (could be empty), but not null:



The other one is the getUsername() method, which will be used for authentication. We need to return our $email field in it, because that’s our username:

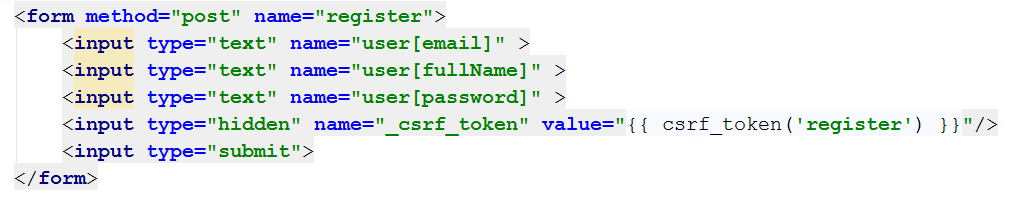


Now going back to the registration action, the error is gone. We can safely set the encoded password to the user object and persist it via [EntityManager](http://www.doctrine-project.org/api/orm/2.5/class-Doctrine.ORM.EntityManager.html) to the database

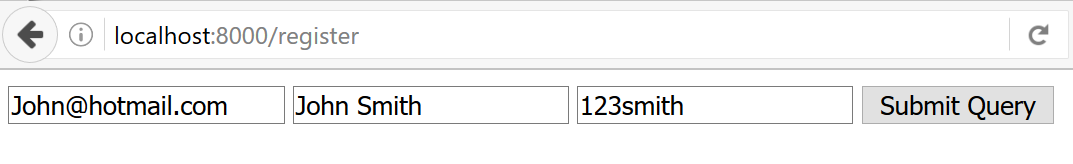


Here we have said that when everything is OK with the form, persist the user and redirect them to the login form. If the form is not submitted, then we need only to render the register form ☺

The form itself contains text fields with names corresponding to the object name and the properties as keys (like an associative array) e.g. the email field is called **user[email]:**



Open <http://localhost:8000/register> and test it:



# Creating Articles

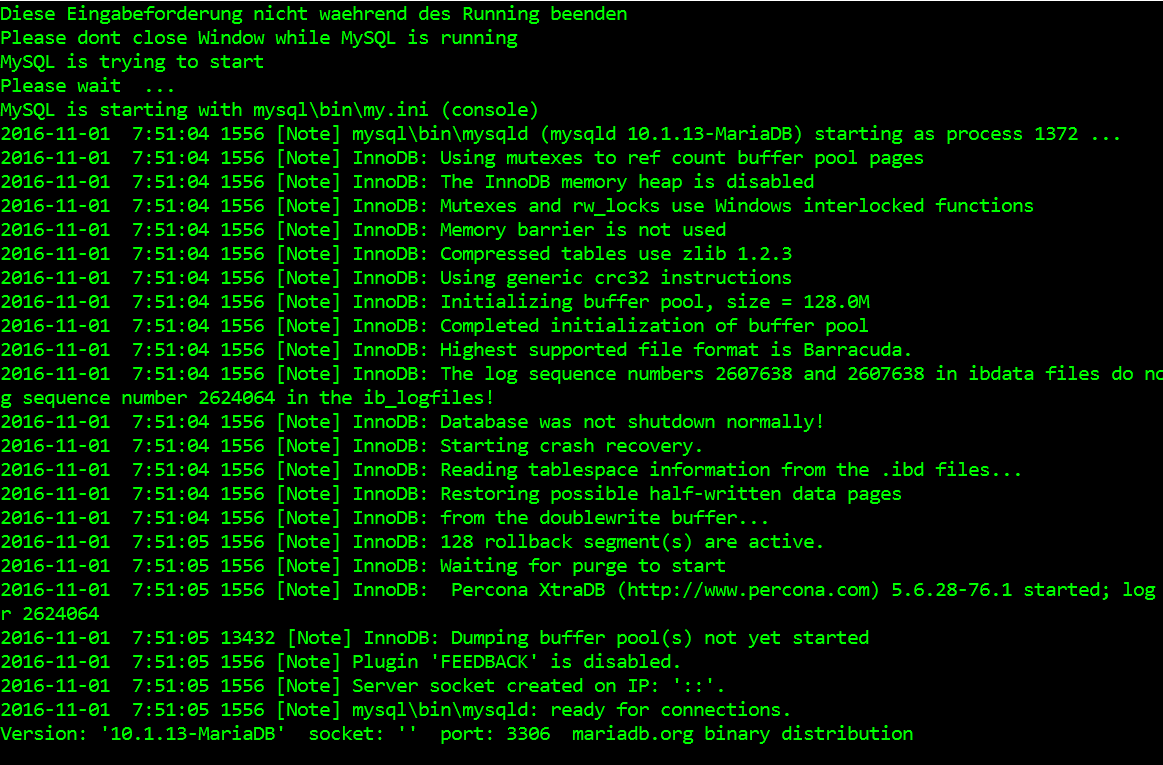
## Start MySQL (Only if you are here from the start)

**Skip this step if you have gone through the above III chapters.**

If you are still reading:

Download the [project skeleton](https://softuni.bg/downloads/svn/soft-tech/Sep-2016/Software-Technologies-Oct-2016/03.%20PHP-Blog-Basic-Functionality/php-sever-skeleton.zip), extract it in a shortest path you can make, e.g. in **c:\project**.

Before we start using our blog, we need to **create** a [database](https://en.wikipedia.org/wiki/Relational_database). We will use [MySQL](https://www.mysql.com/), which you are given in the skeleton. To start using MySQL, just **double-click** **mysql\_start.bat** from the root directory (e.g. **c:\project**). You will see a window like this one:



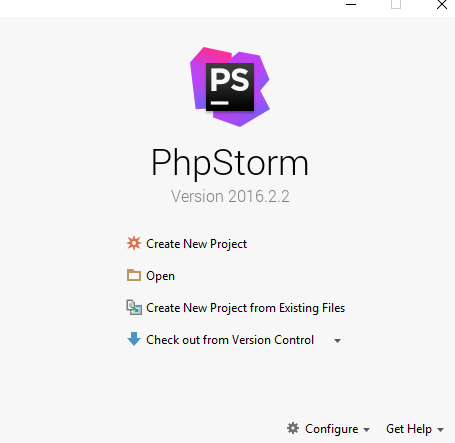
That’s it, MySQL is running. When you decide to stop working on the blog, just close the terminal and run the **mysql\_stop.bat** file.

## Open the Project (Only if you have done 0.)

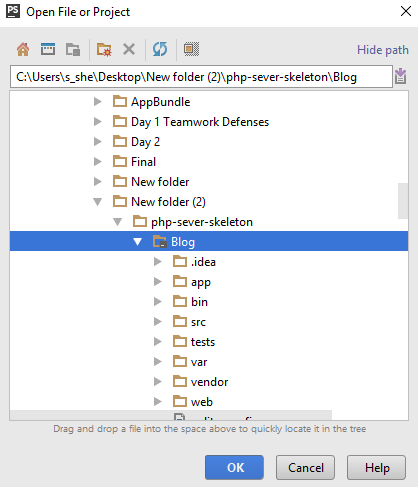
**Skip this step if you have gone through the above III chapters.**

If you are still reading:

For this step, we will open the project with **PhpStorm** or **IntelliJ** Idea. Starting from the home screen, click on “**Open**”:



Locate the skeleton folder that we gave to you and select the “**Blog**” **folder** from the extracted folder (e.g. **c:\project\Blog**):



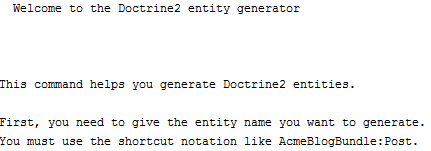
After you click “**OK**” the project should start loading and indexing. After a few seconds/minutes depending on your pc, you will be able to work with the project.

## Create the Article Entity

Open Terminal or Command Prompt (CMD) in the blog project root folder. Let’s model our articles. That means that we are going to create the defining properties of an article. To do that, we need to generate a [Doctrine Entity](http://docs.doctrine-project.org/en/latest/reference/working-with-objects.html). Our entity will describe what are we going to store in our database. The following command will **start entity generator wizard**:

|  |
| --- |
| php bin/console doctrine:generate:entity |

You should see this result:



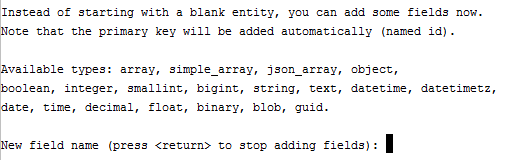
Now we need to choose **appropriate name for our entity**. Use the following name:

|  |
| --- |
| SoftUniBlogBundle:Article |

The result should be the following:



Just press ‘**Enter**’. Now we need to **define the properties** for our entity. You should see this:



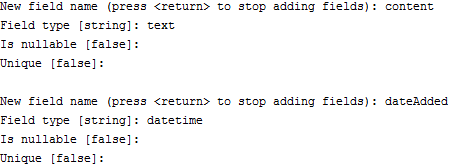
Our first field will be the “Title” of our article. Just write “**title**” and press ‘**Enter**’. You should see this:



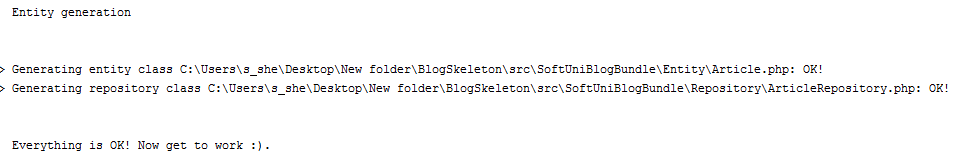
Press ‘**Enter**’. You should see “**Field length [255]**”. Press ‘**Enter**’ again. You will be asked if you want to make the field **nullable**. Press ‘**Enter**’. Finally, you will be asked to make your field **unique**. Just press ‘**Enter**’ one more time. Now you should see this:



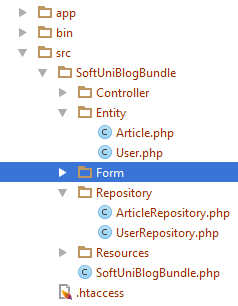
Similar to this, we should create 2 more fields for the “**content**” and “**date**”. Here is how we create them:



Finally, press ‘**Enter**’ one more time to close the wizard. You should see this:



Let’s see the project in **PhpStorm** (IntelliJ Idea):



Okay, Doctrine has created “**Article**” entity and “ArticleRepository”, which is a special type of class. Its job is to manage our data and simplify our work with the database.

## Add Summary to the Article Entity

Let’s go into the “**Article**” entity that Doctrine created in the previous step. It should contain all of the fields, that we created using the terminal, plus one **special** “**id**” **field**. It is the [primary key](http://www.mysqltutorial.org/mysql-primary-key/) for our table. On top of our entity we should see something that looks like a comment:

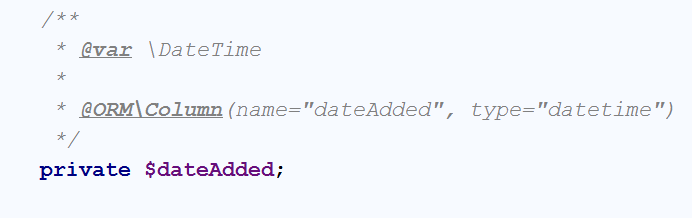


However, this is not just a comment. It is an annotation. More specifically, it is a [Doctrine Annotation](http://docs.doctrine-project.org/projects/doctrine-common/en/latest/reference/annotations.html). It tells Doctrine how are the tables and fields are going to be called in the database. At first glance, we see the annotation

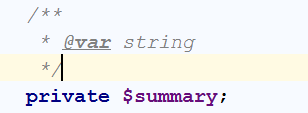
|  |
| --- |
| \* @ORM\Table(name=”article”) |

This defines the name of our table in the database. The names of the tables in the database should be pluralized. For that reason, rename the table to “**articles**”.

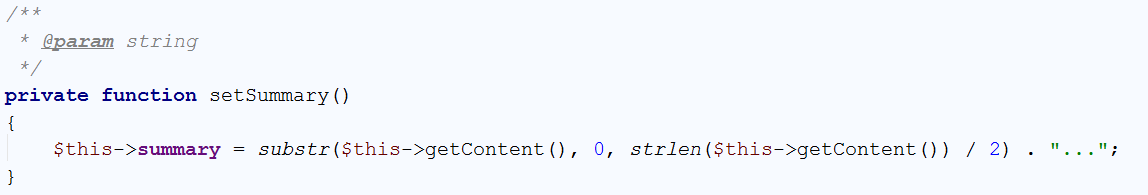
Now we need to create some fields, that will not get saved into the database. Find the “$dateAdded” field. You should see something like this:



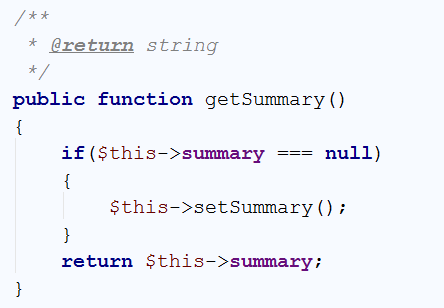
Below that, first add a new private field called “summary”. It will hold the short summary of the article:



Then we need to create [Mutator and Accessor](http://www.refulz.com/mutator-and-accessor-methods-in-php/) (Getter and Setter) methods for the summary. Let’s first start with the **mutator**. Its job is to set the value of summary to half of the article content. The code should look like this:



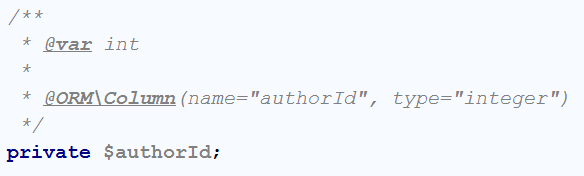
Now we should create the **accessor**. It should simply **return the saved value** in our **summary** variable. However, if summary is empty, it should **call the** **mutator to set the value**:



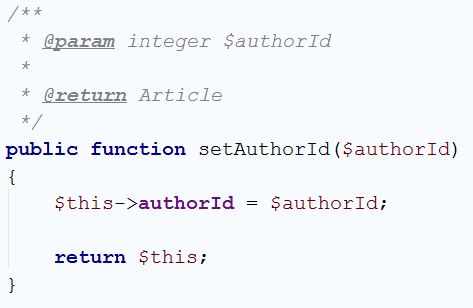
We’re done with the summary variable, but we still have more variable to implement.

## Create a Relationship between the User and the Article

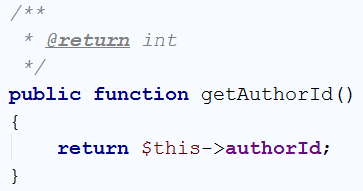
We’ve come to the part where we must connect each user with his articles. To do that, we will create 2 more field in the “**Article**” entity. Just below the private summary field, that we’ve created in the previous step, create new private field called “authorId”. Using that field, each article will know who is its author:



You have probably noticed that we’re going to **save this field in the table** using the @ORM annotation. This will **create a column in the table**, which will keep integer, representing a user. Similarly, to the summary, we need to create **getter and setter** methods for this field. Again, we’re starting with the mutator:



One special thing to note here is that the **mutator** actually returns the object, that it changes. Now simply **create the accessor**:

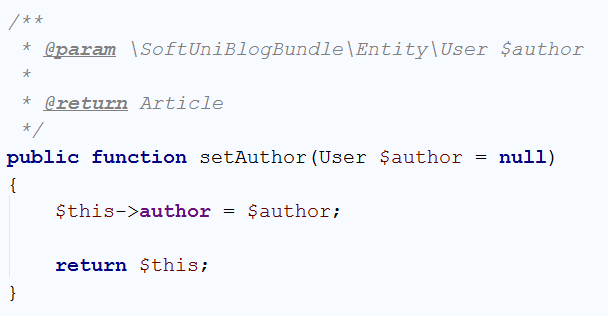


We’re done with the authorId, but the **connection** is **not** **ready** yet. In order for our article to actually have an author, we need to declare a private field of type “**User**”:

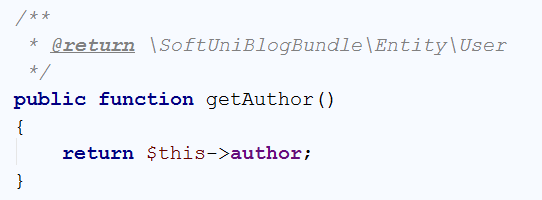


More new stuff! We’re using 2 new annotations. The first one is the “ManyToOne” annotation. Many to one relationship represents [OneToMany](https://en.wikipedia.org/wiki/One-to-many_(data_model)) relationship from the side of the “many”. In our case, we will use “one to many relationship” to tell the program that one user will have many posts. Because we are working with the Article entity, we are telling Doctrine that **many of our articles** will correspond **to one user**. The “inversedBy” parameters tells Doctrine that in the User entity we will have a private field called “**articles**”, which will keep all of the articles of one user. The other annotation is “JoinColumn”, which tells Doctrine how are we going to connect the fields in our entities. Our annotation tells Doctrine that the **field “**authourId**” in our article entity will correspond to the “id” field from the** Userentity.

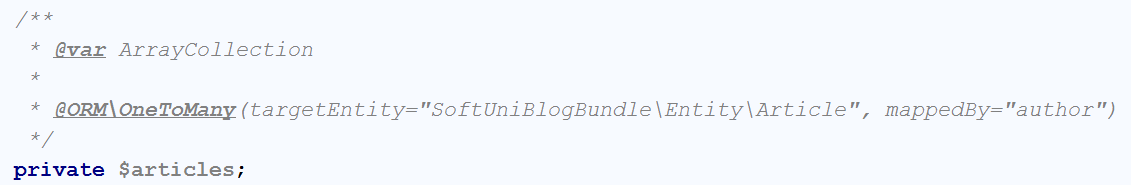
Now we should create the **setter** for the author field:



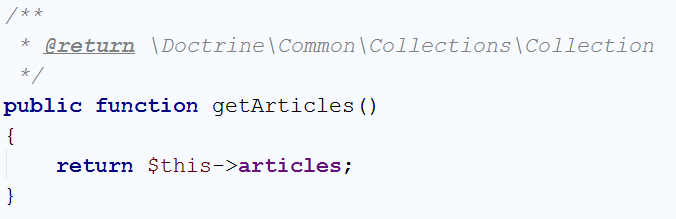
And our **getter**:



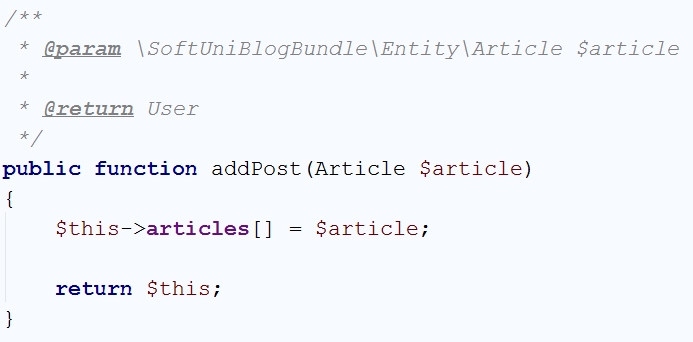
That’s it, we’re done with the Article entity for this step. We need to do the “one to many relationship” on the side of the User entity. Just below the private “password” field, create the following field:



This field will be of type ArrayCollection, that will keep all of the current user posts. As you can see, we define one-to-many relationship with the Article entity, using the author field, we’ve created earlier. For this field, **we won’t create setter** like for previous ones. Firstly, we should create the getter:



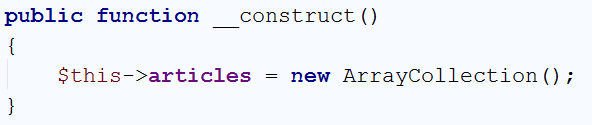
The setter however will be slightly different. It should **add a new post to the current user posts**. To do that, we should write the following code:



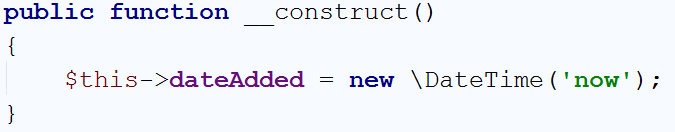
We’re done with the connection for now. Later **we will update the database** schema.

## Set Default Values for our Entities

Our next job is to create the so-called [constructors](http://php.net/manual/en/oop4.constructor.php) for our entities. The constructors are special methods that are called **each time a new object from the entity is created**. Let’s start with the User entity. Its constructor should be the following:



Every time we create a new user, it will receive empty array of articles. The Article on the other hand should look like this:



Each time a new article is created, this constructor will add the current time.

We are ready with this part, now **we can update the database**(schema).

## Updating the DB with our Article Entity

There are many ways to update or create the tables that we need. The first one is to create them **manually**. That will take a lot of time and because of that we won’t do it that way. We will create them **using** [Doctrine](http://docs.doctrine-project.org/en/latest/). **Open** a **Terminal**/**CMD** in the project **root** **folder**. Let’s write the following command:

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

This will result in the following warning:



It basically tells us, that we are doing an operation that is not safe. To do it, **we need to force Doctrine** to execute our command. In order to do that we need to add “**--force**” after our previous command:

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

The result of this command should be the following:



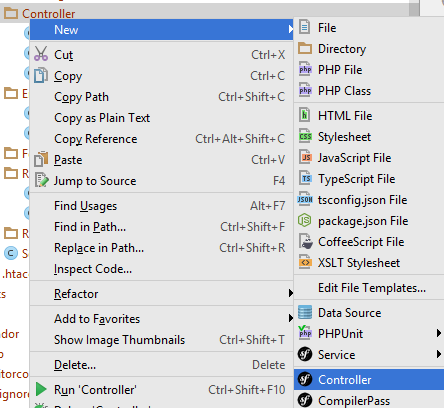
If we take a **look at the DB in HeidiSQL**, we will see that the table “**articles**” is created:



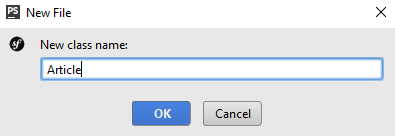
We are ready, to start making our blog.

## Creating the Article Controller

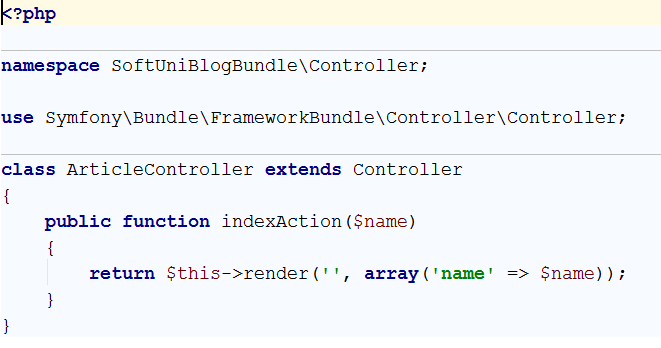
Now we should create a class that will control how the articles are going to be viewed, created, edited and deleted. We will create it in the **Controller** folder. If you are using **PhpStorm or IntelliJ IDEA** and you have the **Symfony plugin installed**, you should see this when you right-click on the Controller folder:



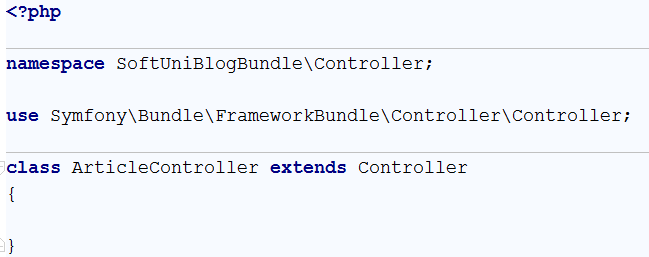
Give it the name Article:



We have just created an ArticleController in the **Controller** folder, that looks like this:



Delete the indexAction method, we won’t need it. We should be happy with the following result:



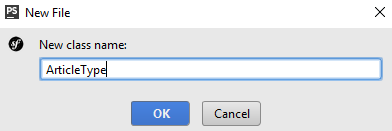
We have a controller, but we need **form template**.

## Creating the Article Type Form

Our next step is to **create a form template**, that we are going to fill, each time when we’re **creating or editing** an article. To create this form, just right-click on the **Form** folder and choose new **Form**:



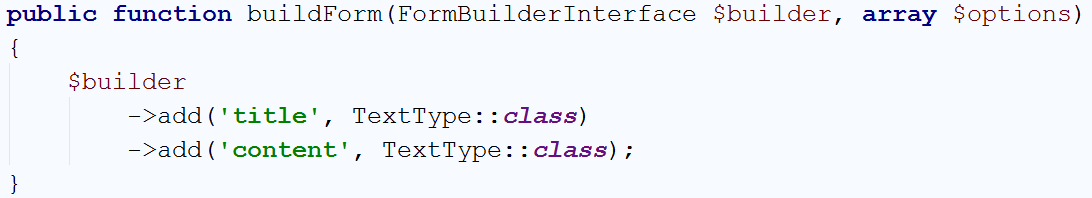
Give it the name “ArticleType”:



We should receive something like this:



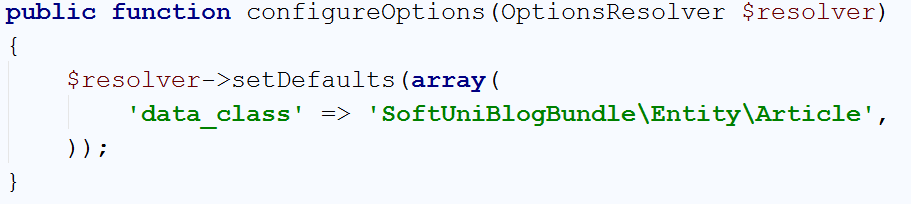
You may notice that we have 2 empty functions. “buildForm” will create our form and “configureOptions” will tell our form that it is for the Article entity. Let’s start with the form creator:



It is pretty simple form. It should only contain **title** and **content** fields, both of type text. You should use specific using for the “TextType” to work. If you have another one **ending in \TextType** already imported – delete it and add:

|  |
| --- |
| **use** Symfony\Component\Form\Extension\Core\Type\TextType; |

Let’s create the logic for our “configureOptions” function:



The default value for our resolver **tells controller that is going to use the form**, in what type of object it should save the date from our form. That’s it.

## Implementing Article Create Function

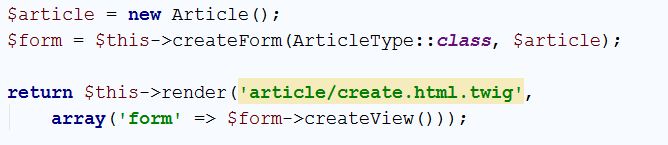
Go back to the article controller, we need to create a new function. We will name it “createAction” and create few annotations for it:



Let’s start from the first annotation. It tells our project that the function will receive **one parameter** of type [Request](http://api.symfony.com/3.1/Symfony/Component/HttpFoundation/Request.html). We will save what request is for some other time. The second annotation is more interesting. It defines a “[Route](http://symfony.com/doc/current/routing.html)”. The **route represents** the **URL**, that the **current** **method will correspond to**. In this case the function will be called when we go to <http://localhost:8000/article/create>. Each time we **use this URL**, the router will **call our function**. To **simplify** the **redirection** between our **pages**, we give a simpler name like “article\_create”. The third annotation is to make sure, that only **logged in users** will **use** our **function**. Without it, every guest **would be able** **to create a new article** and we **don’t** **want** **that**. The final parameter specifies that our **function** will **return** a **response**. We will talk about this later. In order for those annotations to work correctly, make sure you are using the right imports:

|  |
| --- |
| **use** Sensio\Bundle\FrameworkExtraBundle\Configuration\Route; **use** Sensio\Bundle\FrameworkExtraBundle\Configuration\Security; **use** Symfony\Component\HttpFoundation\Request; |

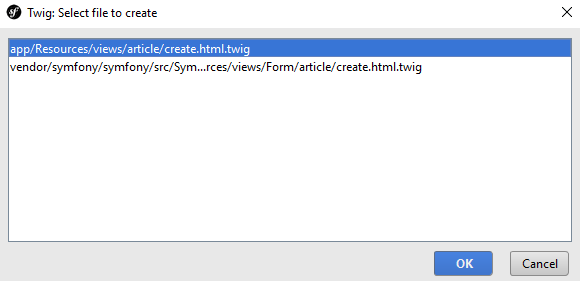
Now let’s write some real code. In the function, write the following:



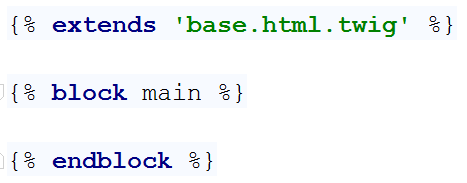
What is this code doing? It’s simple – it **creates a new article**. Then it **creates** a **new form** from the template we’ve created earlier and tells the **form** that it **should** **fill our new article**. Finally, it **sends the form to a** **view** that we are going to **render** on the screen. Render means draw. Symfony uses [Twig](http://twig.sensiolabs.org/). Twig is a [templating engine](https://en.wikipedia.org/wiki/Template_processor), which job is to **display our data** in an **easier** **way**, than creating the HTML by ourselves. The important part here is that we don’t **have such template yet** and PhpStorm (IntelliJ IDEA) tells us, by making **yellow rectangle** over the name of our template. To create it, just click on the template name and then press ‘**Alt + Enter**’. This will open a context menu in which you call tell your IDE to create the template for you:



Then you need to choose the first option:



Congrats, you are looking at an **empty template**. Write the following code:



This code does 3 things. The **first one** is to ‘extend’ an existing template. What does that mean? It means, that **we’ve** **created the base design of the blog for you**, including all **styles** and **scripts** that you may need. **You** **can** now simply **reuse** this **base** **template** in all of the templates you are going to create. The **second** statement replaces a block called “**main**” in the base template. This means that all of the HTML in the base template for the “**main**” block will be replaced by the code you are going to write in a second.

Just because we don’t want you to focus on HTML and Twig, we will give all of the code, that you need to write in the main block:

|  |
| --- |
| <**div class="container body-content span=8 offset=2"**>  <**div class="well"**>  <**form class="form-horizontal" action="**{{ path(**'article\_create'**) }}**" method="POST"**>  <**fieldset**>  <**legend**>New Post</**legend**>   <**div class="form-group"**>  <**label class="col-sm-4 control-label" for="article\_title"**>Post Title</**label**>  <**div class="col-sm-4 "**>  <**input type="text" class="form-control" id="article\_title" placeholder="Post Title"  name="article[title]"**>  </**div**>  </**div**>   <**div class="form-group"**>  <**label class="col-sm-4 control-label" for="article\_content"**>Content</**label**>  <**div class="col-sm-6"**>  <**textarea class="form-control" rows="6" id="article\_content"  name="article[content]"**></**textarea**>  </**div**>  </**div**>   {{ form\_row(form.\_token) }}   <**div class="form-group"**>  <**div class="col-sm-4 col-sm-offset-4"**>  <**a class="btn btn-default" href="**{{ path(**'blog\_index'**) }}**"**>Cancel</**a**>  <**button type="submit" class="btn btn-primary"**>Submit</**button**>  </**div**>  </**div**>  </**fieldset**>  </**form**>  </**div**> </**div**> |

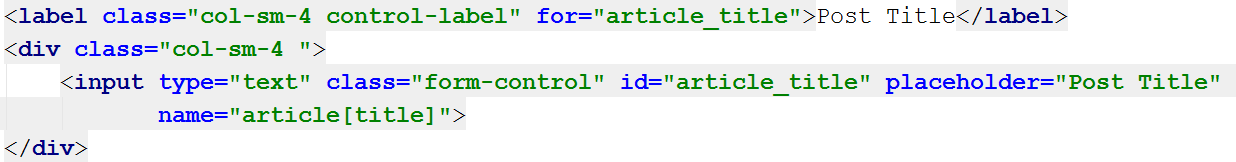
However, let’s explain few parts of that template.

The first part we are going to discuss is:



We are using some css class, this part you should be familiar with. The really interesting parts are the **action** and **method** attributes of our form. First, we are going to talk about the method. This attribute defines what type of [request](http://www.w3schools.com/TAGS/ref_httpmethods.asp) we are going to use. To simplify things, let’s explain the requests shortly. The request we are going to use is “**POST**”. That means that we want to **send** **data** to some place. In our case, it tells the [HTTP](https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol) protocol that we want to **send our title and content** to a place in our blog. The other type of request that we’re interested in is “**GET**”. It tells HTTP that we want to **get** **some** **data** from somewhere. There are other types of requests, but we’re not going to bother you with them now. Let’s talk about the **action**. The action attribute defines **from**/**to** where we want to **GET**/**POST** our data. Remember the name of the route we gave our function earlier? Yeah, we want to send a **POST** **request** with our **title** and **content** **back** to the **function** we’ve created earlier. We will see how to use the data from the request later on.

The second part from the template that deserves a quick look is:



The first thing to notice is that the **for** attribute of the **label** and the **id** attribute of the **input** have the same value. Now take a look at the **name** attribute of the **input**. It looks like dictionary value. When you are mapping your entities in the twig templates, it’s important to note that the first part of the **name** is the **name of the entity**. In the square brackets, we put the **name of the field** from the entity we’re going to fill.

Another interesting thing is:



This is a special twig code. It creates a new **invisible** **field** in our form, that validates our form. Without it, our form won’t work. It you want to know more you should check about [CSRF](https://en.wikipedia.org/wiki/Cross-site_request_forgery).

Finally, one more special twig code that we saw earlier as well:



This “path” command uses route name, and redirects to the given route.

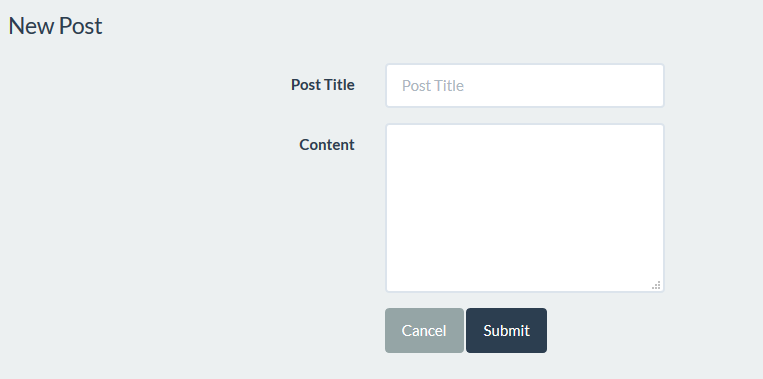
Enough for the templates for now, let’s start the blog and see if it works. To do that we need to open the Terminal/CMD in the root folder of our blog, or use the built-in terminal in PhpStorm (IntelliJ Idea). Don’t forget to start MySQL if you haven’t by now. Enter the following command:

|  |
| --- |
| php bin/console server:run |

If everything works, you should see this:



Open the browser and go to the address. You should see almost empty page. Now you need to register a new user and login. After login, in the URL enter <http://localhost:8000/article/create>. It should redirect you to form like this one:



**Fill the form and click** “Submit”. The **page gets refreshed**, but if we check the table in the **database**, **it is empty**. Let’s fix the problem. Get back to your function in the article controller. The problem is that we’ve never used the data from our form. Add to your function the following code:

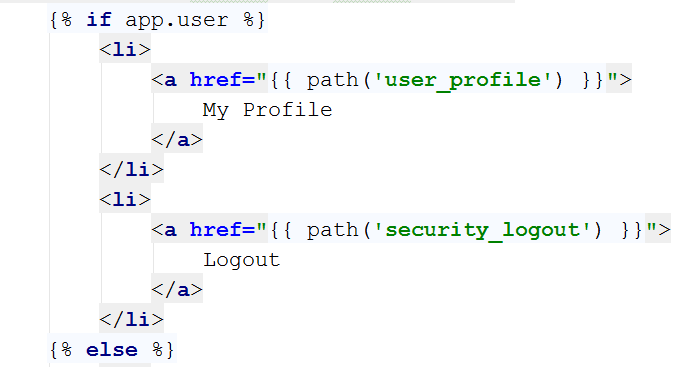


This code **takes the data from request (make sure the imported “use” statement at the beginning of the class is Symfony\Component\HttpFoundation\Request)** and **fills** the **form**. After the form is filled, we check if there is **any data** in the form and if it **is valid**. If everything is okay, then we get the currently **logged** **in** **user** and assign him as **author** of the **article**. Then we get the **entity manager** from **doctrine** and using the “persist” function we **add** our **new article** in the **database**. Finally, we call the “flush” function, which sends the article to our database. **After** the article is **sent** to the database, we **redirect** the **view** to the **index page** of our blog.

While we’re changing the code, open the base template:



Find this part of the code:



Add a new “<li>” element which will redirect to the **create article** page:



Let’s go to our blog and login. Now on the right-side of the navigation bar, we see the new button:



Let’s try to create new article. After pressing the “**Submit**” button, we should get redirected to the home page. Let’s see if we got anything new in the database:



Hooray, we did it! Now we can create articles. The problem is that we can’t see them from our blog. We will fix that.

# Listing Articles

## Listing All Articles

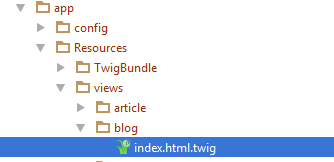
Let’s go to the home controller. When you open it, you will find a function called “indexAction”. Its **only** **job** at the **moment** is to **call** the **index** **view**, without any data. We will change that. **Write** the following **code** in the **beginning** of the **function**:



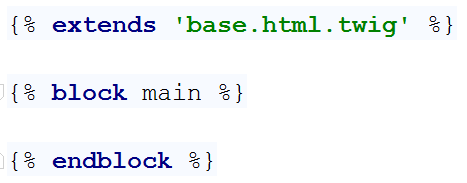
This will get all of our articles from the database. Let’s pass them to the view. Edit the return statement like this:



We’re done here, go to the view, and examine it:



You should see this:



In the main block, write the following code:

|  |
| --- |
| <**div class="container body-content"**>  <**div class="row"**>  {% **for** article **in** articles %}  <**div class="col-md-6"**>  <**article**>  <**header**>  <**h2**>{{ article.title }}</**h2**>  </**header**>   <**p**>  {{ article.summary }}  </**p**>   <**small class="author"**>  {{ article.author }}  </**small**>   <**footer**>  <**div class="pull-right"**>  <**a class="btn btn-default btn-xs"  href="#"**>Read more **&raquo;**</**a**>  </**div**>  </**footer**>  </**article**>  </**div**>  {% **endfor** %}  </**div**> </**div**> |

There are few key moments that we want to take a look at. The first one is:



This is a simple foreach loop in twig. It will traverse the array of articles we’ve sent to the view through the controller. There is also a closing statement few lines below:

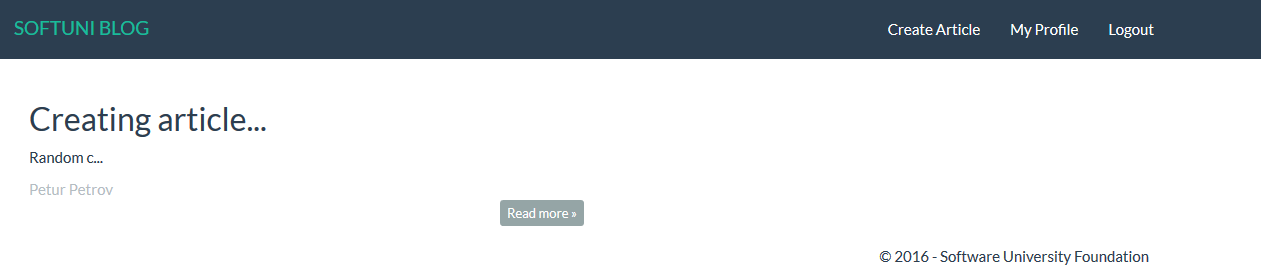


Between those two rows, there are a couple of twig calls. The first one is:

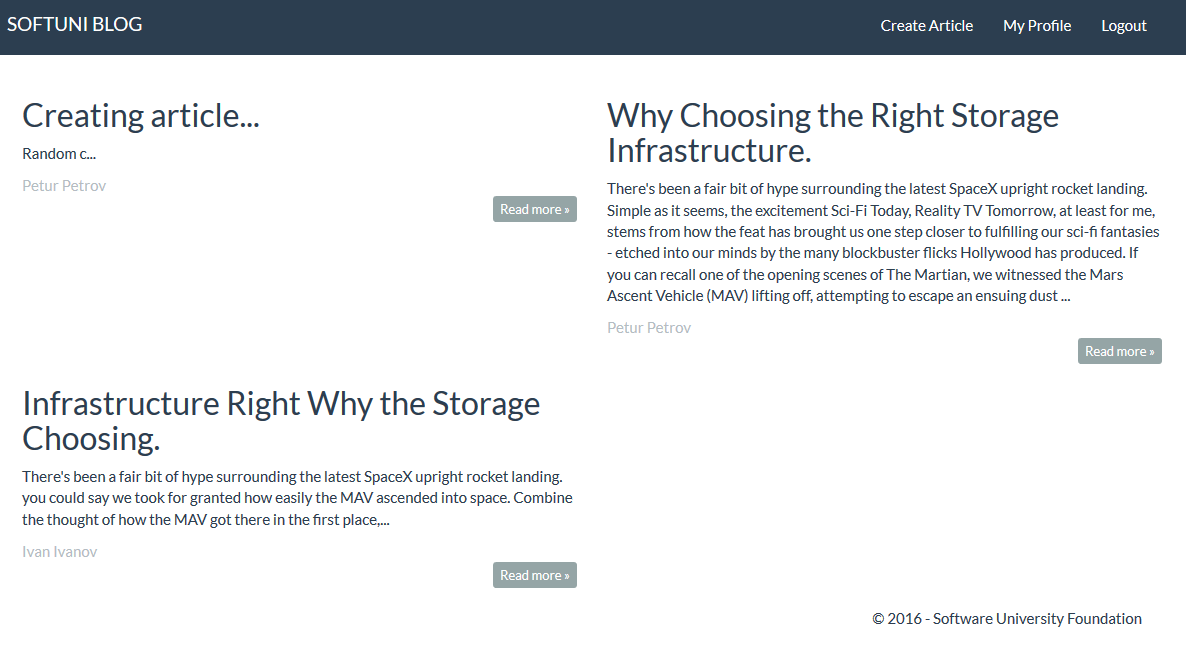


This will print the title for each article. We have the same thing for the summary and author of the article.

For now, let’s start the blog and see what we have:



It works! Let’s create few more articles:



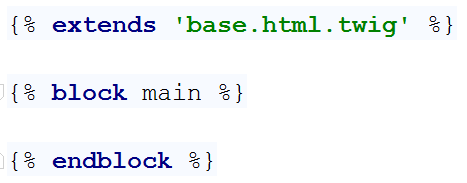
Looks good. The problem is that if we press “**Read more**” nothing happens. We should fix that.

## Showing Single Article

To implement the single article page, we need to go back to the article controller. Create the following method:



Let’s take a look at the annotations. The route annotation is having curly braces (‘{‘, ‘}’) and some parameter inside them. That is the parameter, that the function takes. Everything else is standard. If we take a look at the function, we can see that we are looking for a specific **id** in the database. This row will give us only the article with the given **id**. Then we send it to the view. Create the view, like we did earlier. The generated view will contain the base structure we are already familiar with:



Write the following code in the main block:

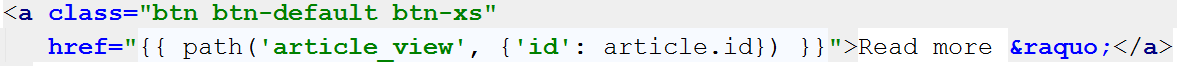
|  |
| --- |
| <**div class="container body-content"**>  <**div class="row"**>  <**div class="col-md-12"**>  <**article**>  <**header**>  <**h2**>{{ article.title }}</**h2**>  </**header**>   <**p**>  {{ article.content }}  </**p**>   <**small class="author"**>  {{ article.author }}  </**small**>   <**footer**>  <**div class="pull-right"**>  <**a class="btn btn-default btn-xs" href="**{{ path(**'blog\_index'**) }}**"**>back **&raquo;**</**a**>  </**div**>  </**footer**>  </**article**>  </**div**>  </**div**> </**div**> |

This code is really simple, with the only difference from the previous one being that we have only one article and we are printing the content instead of the summary.

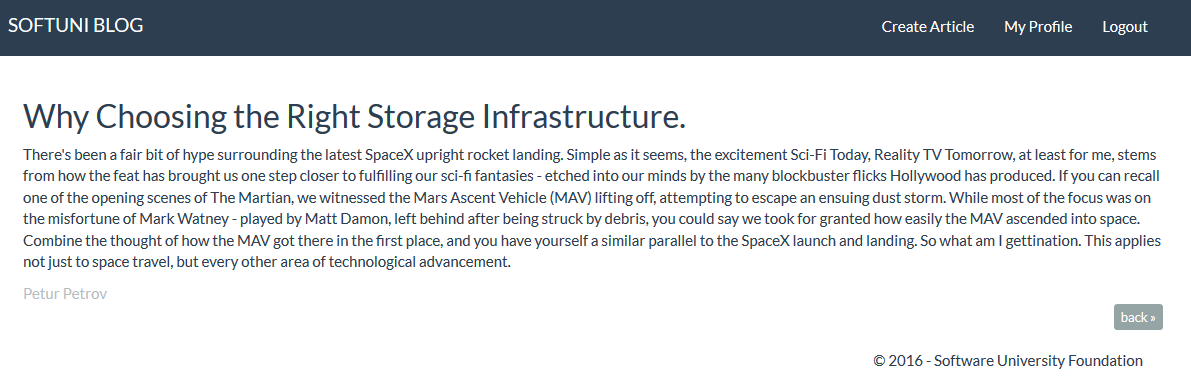
Let’s start the blog and see if it works. The answer is no, it doesn’t. Right now, the button read more doesn’t redirect to the right route. Let’s go back to the index view and find this piece of code:



Change it to:



Let’s try it now:



Another victory! In the next chapter, we will create the functionality to edit articles.

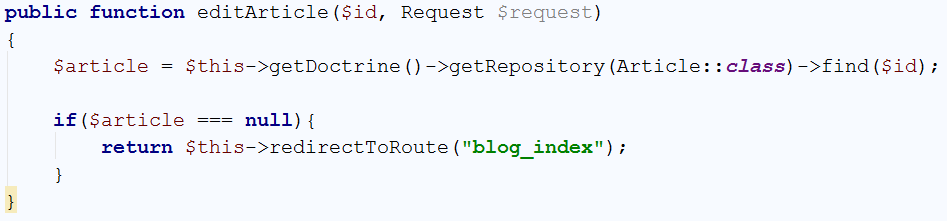
# Editing Articles

## Creating the Action

Let’s create the functionality to **edit articles**. Go to your ArticleController. Create a new action called editArticle():



With the annotations, we **define the route** and we tell that this function is going to be **only accessed** by **logged in user**. Now we need to check **if the article exists** and **if it doesn’t** we are going to **redirect to the home page**:



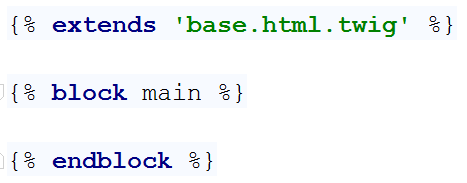
So far, so good. But what if the article actually exists? We need to **render a form**, which will **allow us to edit the article**:



The form will be the **same form** we’ve used for the create() **function**. Let’s create the view now.

## Creating the View

You should start with this:



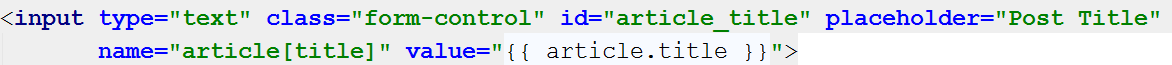
In the main block, write the following code:

|  |
| --- |
| <**div class="container body-content span=8 offset=2"**>  <**div class="well"**>  <**form class="form-horizontal" action="**{{ path(**'article\_edit'**, {id: article.id}) }}**" method="POST"**>  <**fieldset**>  <**legend**>Edit Post</**legend**>   <**div class="form-group"**>  <**label class="col-sm-4 control-label" for="article\_title"**>Post Title</**label**>  <**div class="col-sm-4 "**>  <**input type="text" class="form-control" id="article\_title" placeholder="Post Title"  name="article[title]" value="**{{ article.title }}**"**>  </**div**>  </**div**>   <**div class="form-group"**>  <**label class="col-sm-4 control-label" for="article\_content"**>Content</**label**>  <**div class="col-sm-6"**>  <**textarea class="form-control" rows="6" id="article\_content"  name="article[content]"**>{{ article.content }}</**textarea**>  </**div**>  </**div**>   {{ form\_row(form.\_token) }}   <**div class="form-group"**>  <**div class="col-sm-4 col-sm-offset-4"**>  <**a class="btn btn-default" href="**{{ path(**'article\_view'**, {id: article.id}) }}**"**>Cancel</**a**>  <**button type="submit" class="btn btn-success"**>Edit</**button**>  </**div**>  </**div**>  </**fieldset**>  </**form**>  </**div**> </**div**> |

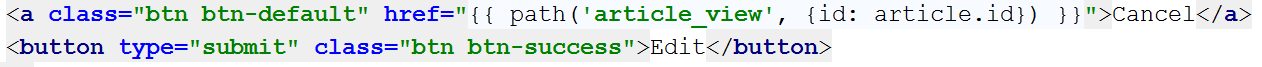
This template may look familiar to you. That is because this is the create view, with few changes. First of all, the action attribute of the <form>, leads to a different route:



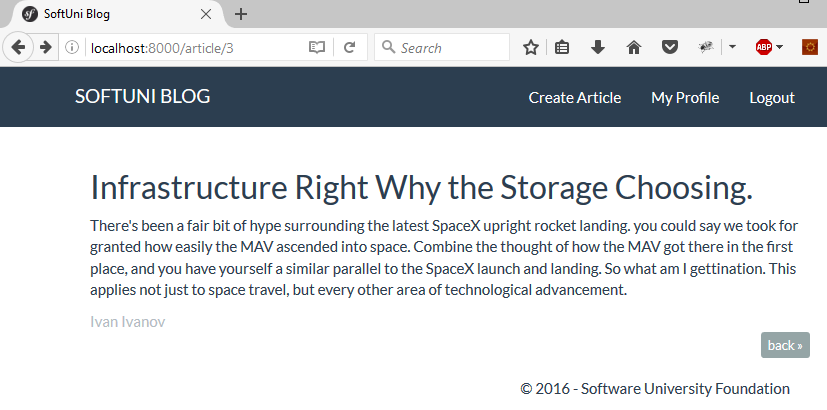
Then, by default every **input field** is **filled** with our **existing article**, using the value attribute:



Finally, the “Cancel” button return us to the single article view and we’ve made the “Edit” button green:

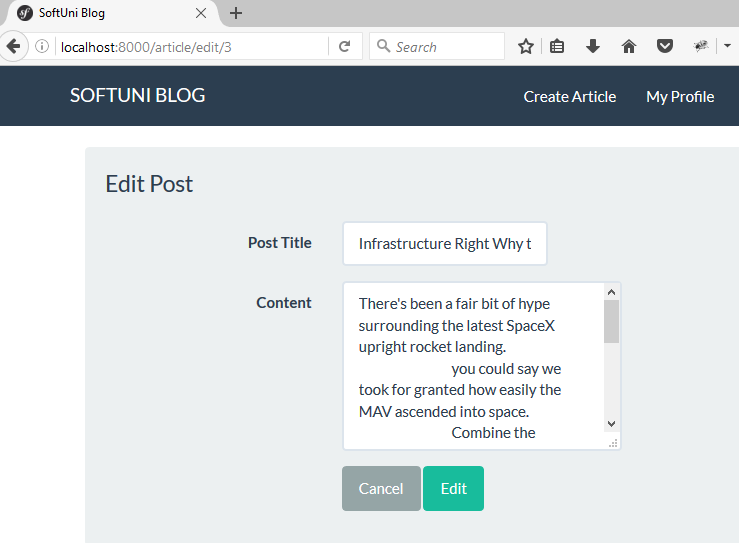


That’s all. Let’s see what we have. Go to **your blog** and open a **single article view**:



The URL for me is <http://localhost:8000/article/3>. I will change it to <http://localhost:8000/article/edit/3>.

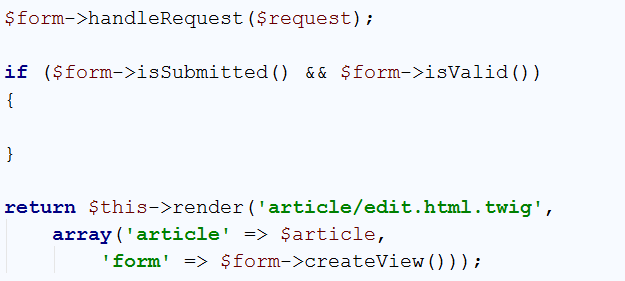
You should see something like this:



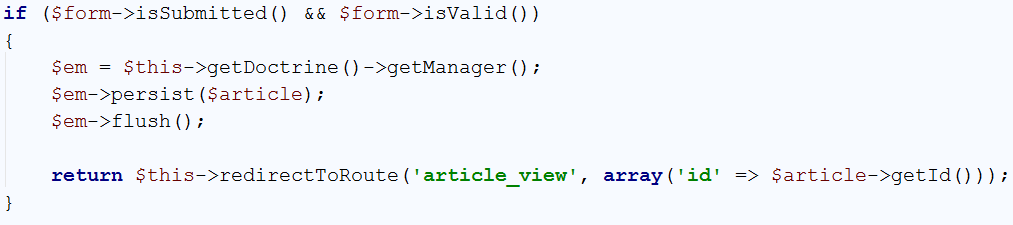
It looks good but **it doesn’t work**. That is because we are **not saving our changes**. We need to fix that.

## Saving the Edit Changes

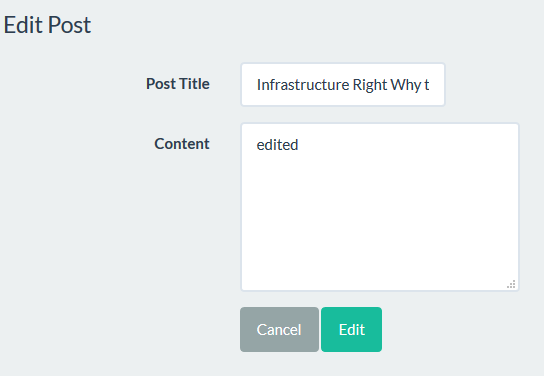
Let’s go back to the ArticleController and our edit() function. **Above** the final return statement, we need to **get the data** from our **form**:



Now we’re **taking the data** from our **form** and the only thing left is to **save the changes**:



Let’s try to **edit the post**, we’ve opened earlier. I will change the **content** to “edited”:



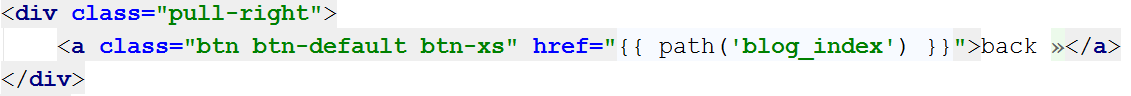
After I **press** “**Edit**” I should be **redirected** to the **article view**. And here it is:



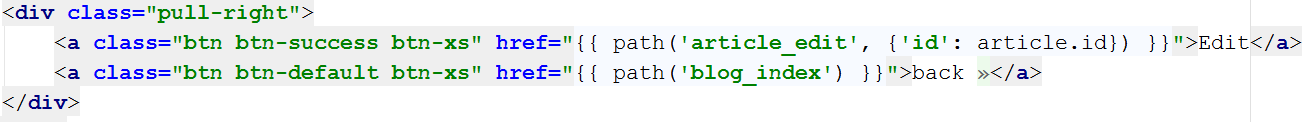
We’ve created the edit functionality, but it’s hard to use it.

## Adding Edit Button to the Single Article View

Open the “article.html.twig” file. **Find** the **back** **button**:



Let’s **add another button** next to it in the <div>. We will make it green, and it will **redirect to the** edit() **function**:



Let’s see if it works:



After we **click** it, we get **redirected to the edit page**:



That is everything when it comes to editing. Next chapter – **deleting**.

# Deleting Articles

## Creating the Action

Let’s create a new delete() function in our ArticleController. It should look like that:



The contents of the function will be **very similar** to our edit() function:



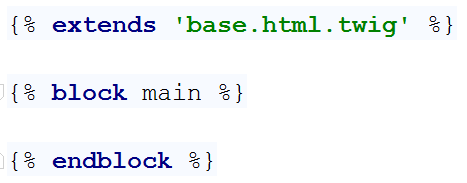
There are very few differences. First and **most important** is the function we are going to **call** when we **submit our form**. For this action, the function we are going to call is:



This function will **delete** the **article** with the given id. Then we are going to **redirect** the user to the home page. Now we must create the view.

## Creating the View

You should start with this:



In the main block, write the following code:

|  |
| --- |
| <**div class="container body-content span=8 offset=2"**>  <**div class="well"**>  <**form class="form-horizontal" action="**{{ path(**'article\_delete'**, {id: article.id}) }}**" method="POST"**>  <**fieldset**>  <**legend**>Delete Post</**legend**>   <**div class="form-group"**>  <**label class="col-sm-4 control-label" for="article\_title"**>Post Title</**label**>  <**div class="col-sm-4 "**>  <**input type="text" class="form-control" id="article\_title" placeholder="Post Title"  name="article[title]" value="**{{ article.title }}**" disabled**>  </**div**>  </**div**>   <**div class="form-group"**>  <**label class="col-sm-4 control-label" for="article\_content"**>Content</**label**>  <**div class="col-sm-6"**>  <**textarea class="form-control" rows="6" id="article\_content"  name="article[content]" disabled**>{{ article.content }}</**textarea**>  </**div**>  </**div**>   {{ form\_row(form.\_token) }}   <**div class="form-group"**>  <**div class="col-sm-4 col-sm-offset-4"**>  <**a class="btn btn-default" href="**{{ path(**'article\_view'**, {id: article.id}) }}**"**>Cancel</**a**>  <**button type="submit" class="btn btn-danger"**>Delete</**button**>  </**div**>  </**div**>  </**fieldset**>  </**form**>  </**div**> </**div**> |

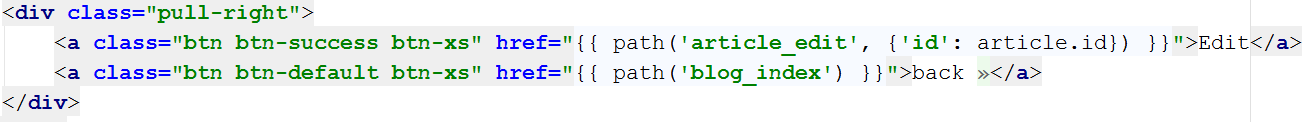
This code may look familiar to you. This is the **edit view** with only 3 changes. The first change is that our form action redirects to the delete function. The second one, is that we’ve changed the **delete** **button color** to **red**. The third and final one is that our **text inputs** have the disabled attribute:



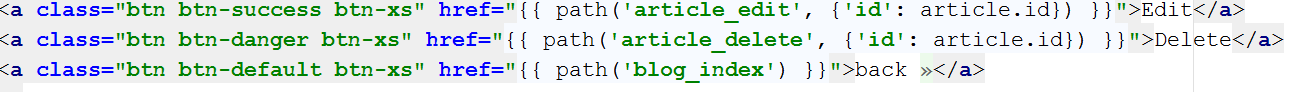
The only thing left is to **create a button** in the **single article view**, which will call our delete() function.

## Adding Delete Button to the Single Article View

In the “article.html.twig” file find the place where we’ve added the edit button:



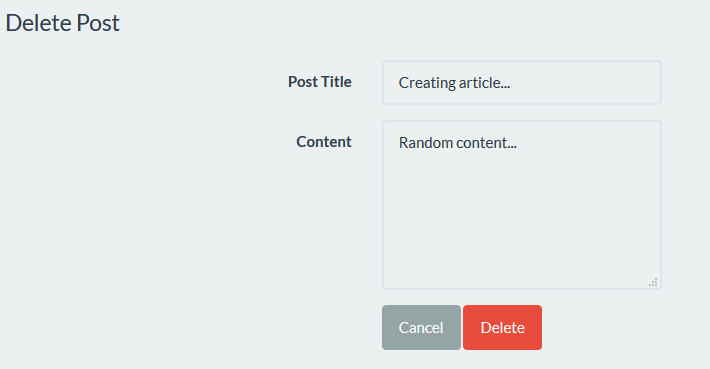
You’ve guessed it, we should create the delete button here:



Everything should be working. Let’s see how it looks:



**Click** the **delete** button. You should be **redirected** to the **delete view**:



If you try to delete the article, you will see that it **works**! Now we have another problem. **Everyone can delete any article**. Even the **articles** that **belong to other users**. We will **fix** that in the following chapters.

# Implementing Roles

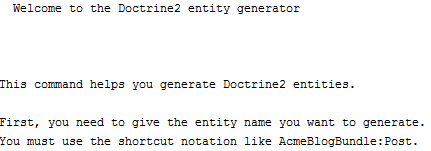
Before we start working with roles we should tell you how can you use them. **Roles are used** to create **user authorization**. For example, if you want a given page from your blog to be **only accessed by admins**, then you should create **admin role**. If you want only the **authors of articles** to be able to **edit** them, you can also do that with roles. Roles are **the natural way to filter** **content** based on **user privileges**.

## Creating the Role Entity

Let’s start by creating the role entity. In the **terminal/CMD** write the following:

|  |
| --- |
| php bin/console doctrine:generate:entity |

You should see this result:



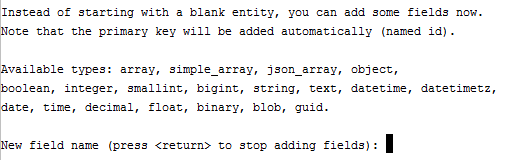
Now we need to choose **appropriate name for our entity**. Use the following name:

|  |
| --- |
| SoftUniBlogBundle:Role |

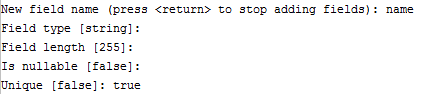
The result should be the following:



Just press ‘**Enter**’. Now we need to **define the properties** for our entity. You should see this:



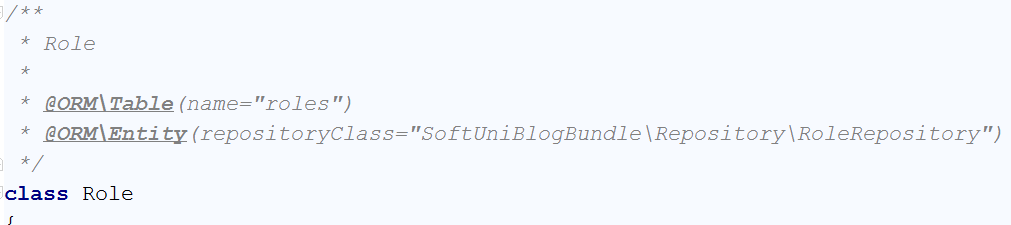
Our **roles** will have **only one column** and that is the **name** of the role:



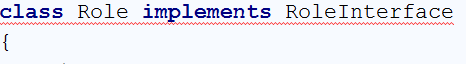
That’s all, just press ‘**Enter**’ again. In the “**Entity**” folder we should have the Role entity.

## Modifying the Role Entity

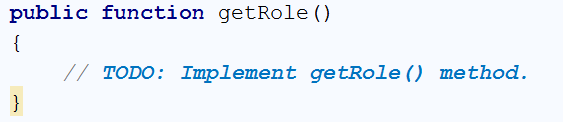
Open the Role entity. First thing to do – pluralize the table name:



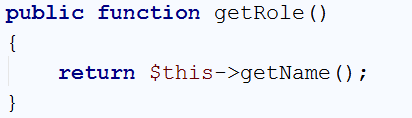
This entity will be special, just like the User entity. It will implement the RoleInterface [interface](http://www.techflirt.com/tutorials/oop-in-php/abstract-classes-interface.html). This is a contract between our entity and the interface. Write the following:



As you can see, it makes the row red. We need to press ‘**Alt+Enter**’ and choose “Add method stubs”. This will add empty getRole() function to the bottom of our class:

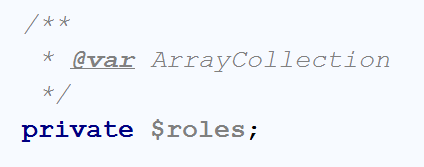


We will just tell it to **return the role name**:

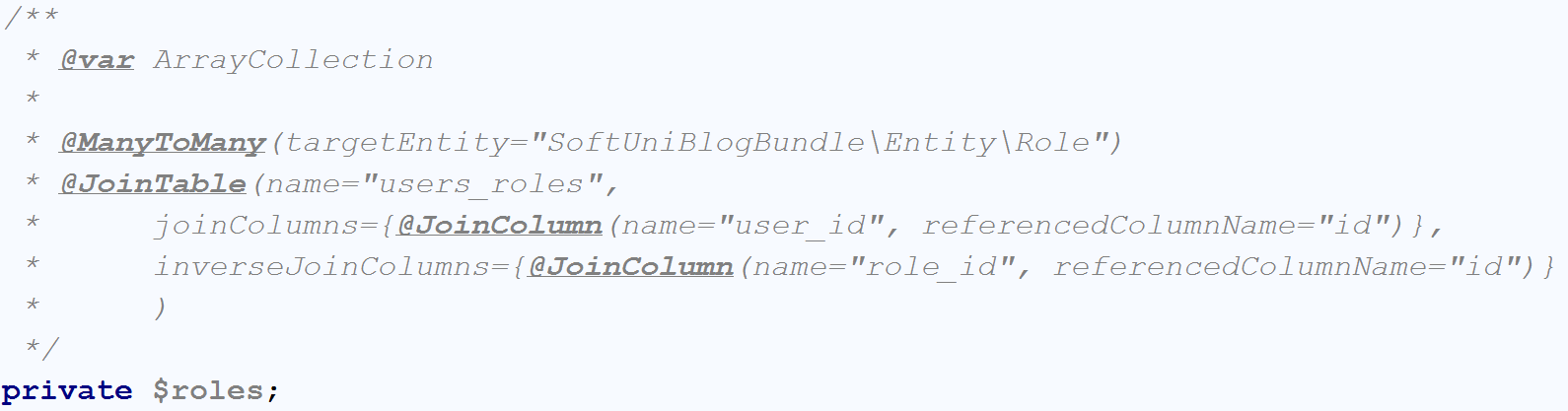


## Create the User-Role Relationship

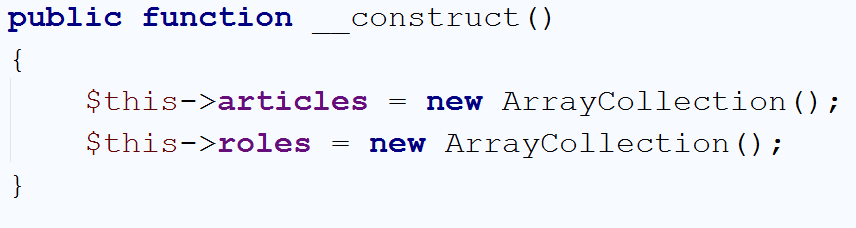
We need to tell the database how are we going to use the roles and users together. Let's create a **new** **private field** in the User entity called "roles":



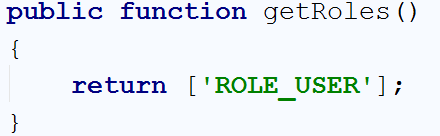
That is not enough. We need to tell Doctrine that our relation is going to be [Many-To-Many](http://www.tomjewett.com/dbdesign/dbdesign.php?page=manymany.php). We're going to do that, by writing the following annotations:



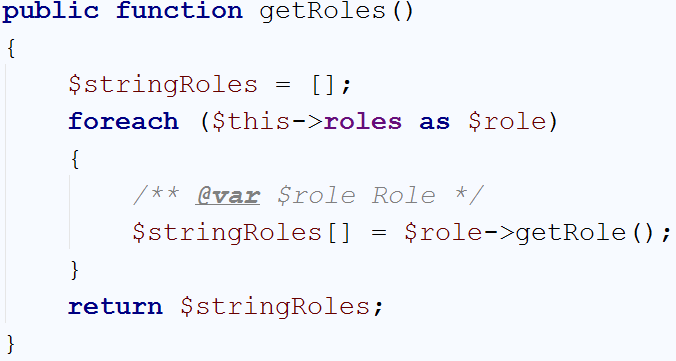
The "@ManyToMany" annotation tells Doctrine that **many users** will have **many roles**. The "@JoinTable" annotation will **create a table** that will **keep our relations**. While we're still in the User entity, let’s **initialize** the $roles ArrayCollection in our constructor:



Now you should find the getRoles() function that looks like this:



Right now, **every user** is having the "ROLE\_USER" role, which is something we will change. We will **change the function** to **return all of the roles** the current **user has in the database**:



That way it will get the **private field** we've created earlier and **take each role's name**.

One last thing for now is to **create a function** that will **add new role** to the **current user roles**:



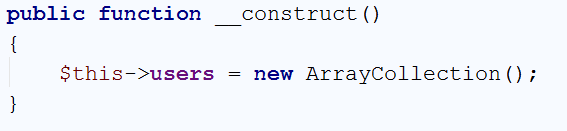
Let's create the relationship from the Role side.

## Create the Role-User Relationship

In the Role entity create the following field:

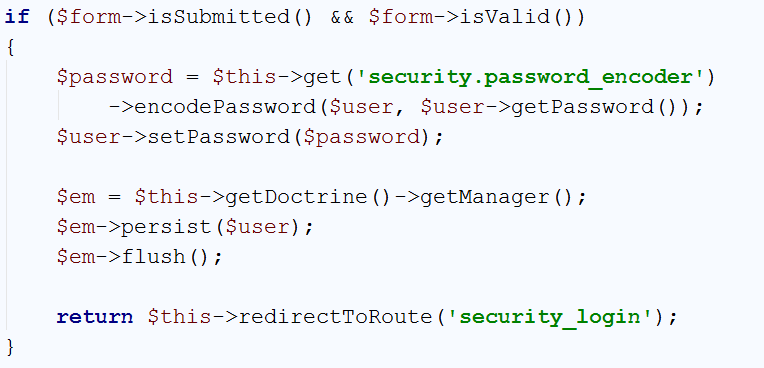


It tells Doctrine that **many roles** will have **many users** and that the relation is **mapped by** roles field in the User entity. Let's create a constructor that will initialize the ArrayCollection:

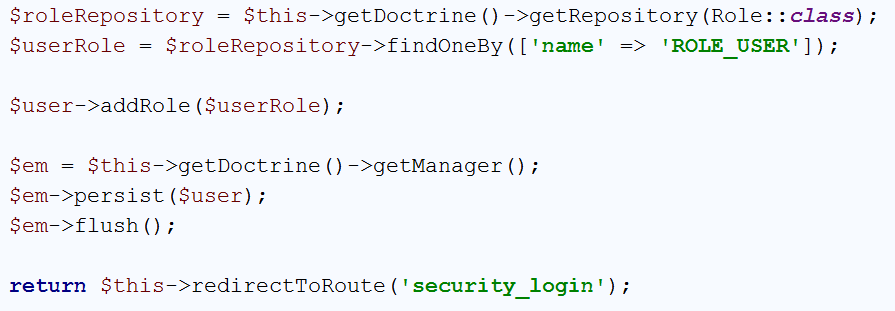


## Edit the User Registration

We've changed how the users are being are initialized and because of that we need to change the registerAction() function in our UserController. Right now, if the user registration form **is valid** and **submitted** we do the following:



We need to add few lines of **code before** we **save** the **user** to the **database**:



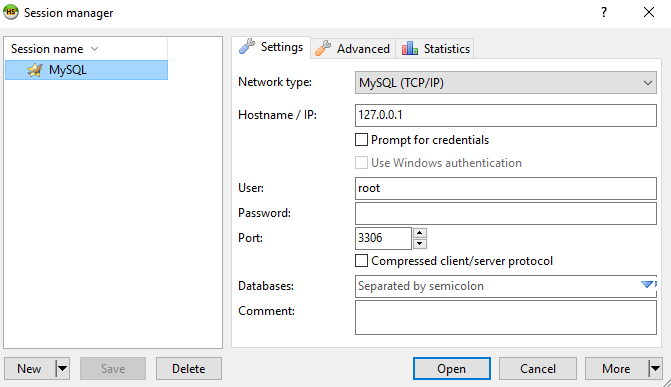
First we get the RoleRepository and then **we get the role** with **name** "ROLE\_USER". Finally, we assign it to the user. It is important that you have the **following using imported on top of your file**:

|  |
| --- |
| **use** SoftUniBlogBundle\Entity\Role; |

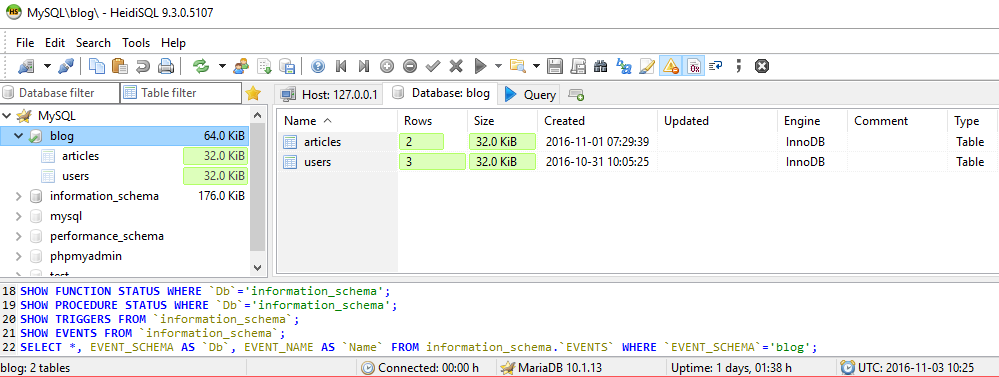
If you have **any other** "use" statement ending with "\Role" **delete** it!

## Update the Database Schema

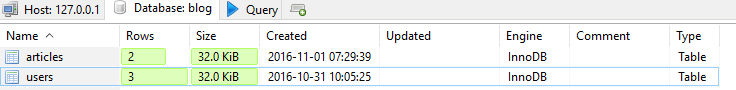
**Before we update the database**, we will **delete all existing tables and data**. We are doing that, because it **will be easier** than **modifying the DB manually**. Open **HeidiSQL**. In the first screen that you see, just click "**Open**":



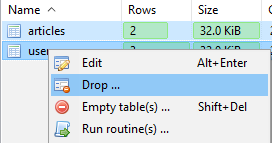
You should see something like this:



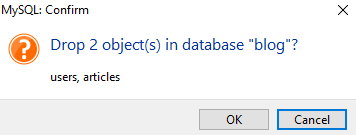
On the **left side** of the **window** you should see the "**blog**" **database**. Click on it. Then in the main part of the window you should see this:



**Select all tables** with your mouse and **right click**. Select the option "Drop…":



You will be asked if you are sure. Click **OK**:



That **will delete our tables and all of the data stored in them**. Now that we've done that, we **can update the database schema using Doctrine**.

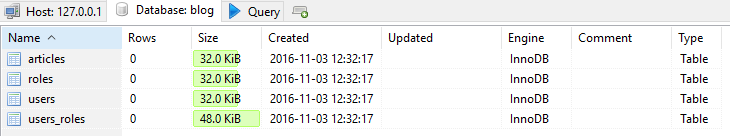
Like we've done before, we **will update the database schema using terminal/CMD**. We will force the update to make sure everything works fine:

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

The result of this command should be the following:



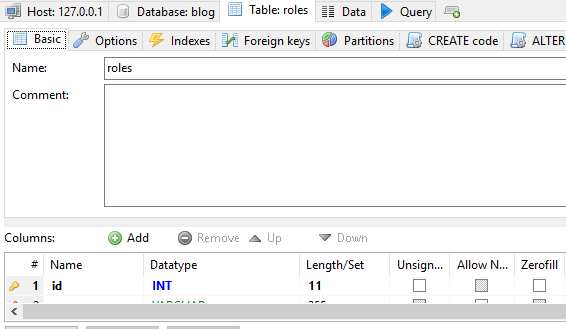
Back in **HeidiSQL** click on the **blog database** and then press "**F5**". In the main part of your window, you should see this:



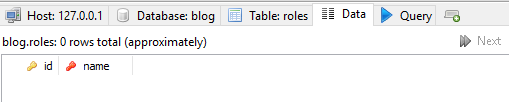
We have successfully created the new tables. Let's start using our roles now.

## Create Roles

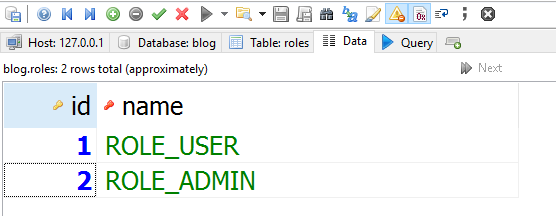
Still in **HeidiSQL**, **double-click** on the roles table. The main screen should change to this:



In the navigation, you will see "**Data**" tab. Open it and you should see this:



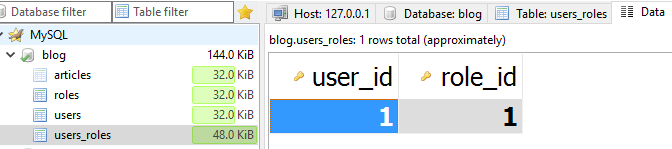
Our database is empty. Click on the "**green plus**" in the **main toolbar**  and you will **be able to enter data** in a **new row**. **Create two roles**, one called "**ROLE\_USER**" and one "**ROLE\_ADMIN**". It should look like that:



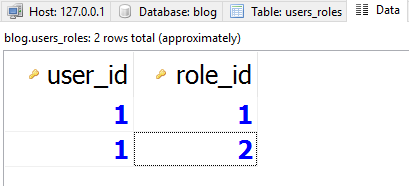
Now we can create new user.

## Register New Admin

Let’s register new user, but **don't login** yet. **After you've created the user**, open the **data tab** for the "users\_roles" table in our database:



Add a **new row**, like you've done earlier and **enter the following data**:



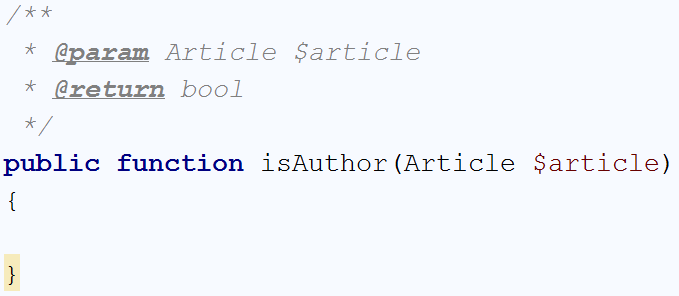
This should make our user a special one – '**admin**'. We will see how we will use that in the next chapter.

# Creating Validations

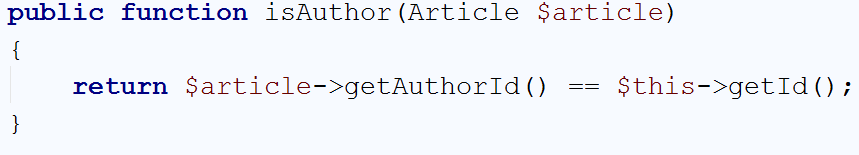
We've already talked about how **every user** is able to **edit** and **delete** **all** **articles**. That shouldn't work like that. Before we start, **create few users** (not admins) and **create at least two articles** with **two users**.

## Create User Helper Functions

Before we start creating **validations**, we should make our life easier. Open the User entity and let's **create the following function**:

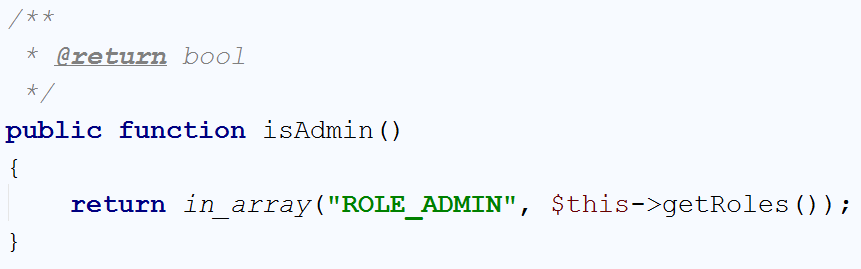


It will check if the **currently logged-in user** **is** the **author** of a **given article**. You should write the following code:



That's all we need to check for that function. If the authorId from the **article** matches the **current user** Id, we will return true.

We need one more **function** that will tell us if the currently logged-in **user** **is admin**:

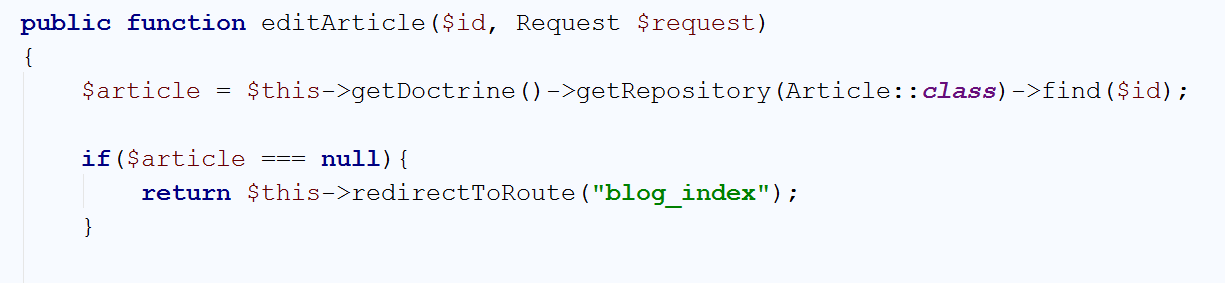


The in\_array() function in PHP is the Contains() method in C#. We check if the roles ArrayCollection **contains** "ROLE\_ADMIN" and if it does, we return true.

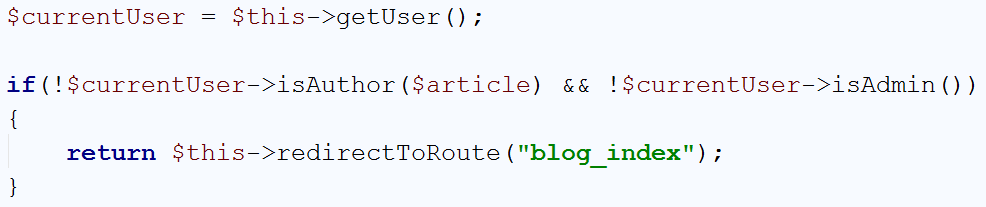
We can now start making **validations**.

## Validating the Edit Article Function

Find the editArticle() function in the ArticleController. At the moment, it should start like that:



The code we are **going** **to** **write now** will be **right after** the existing if statement. Our code should take the **current user** and **check** if he **is the author** of the **article**. If he **isn't**,we will **check** if he **is admin**. If **both** are false, we will **redirect him to the home page** of the blog. Write the following code:

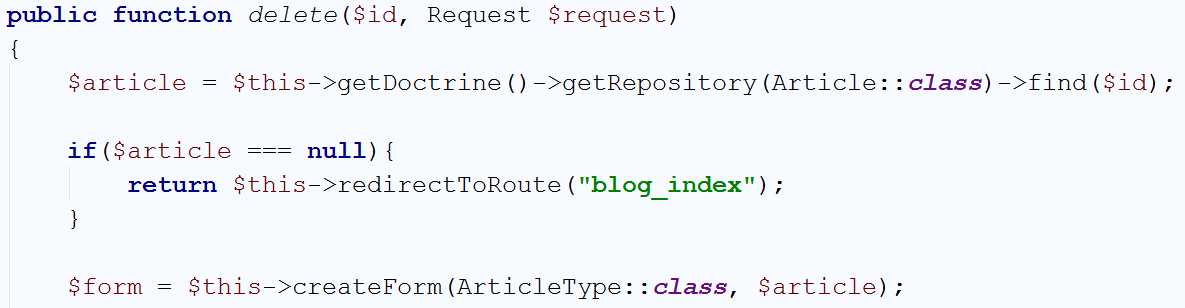


If you've created 2-3 users with 2-3 articles each, you can test if it works. Simply **try to edit an artic**le that is **authored** by another user. If you are **logged with the admin user** we've created earlier, you should be able to **edit all posts**. If you are logged with a **regular user**, you should be **able to only edit your own posts**.

We need to create the same check for the delete() action.

## Validating the Delete Article Function

The delete() function in the ArticleController should begin like this:



Just like in the edit() function, we need to write the **exact same code after the** null check **statement**.

In the end, it should look like this:



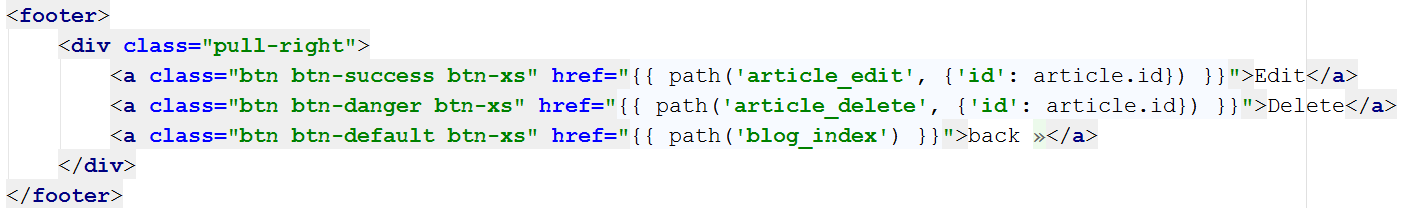
Test it and see if it works. If everything is working as it should, **the admins will be able to delete all posts**, and only the **authors will be able to delete their own posts**.

We have another problem. Do we need to **allow all users to see** the "**Edit**" and "**Delete**" buttons? **No**, they should be **visible only to the authors and admins**. We will deal with that in the next chapter.

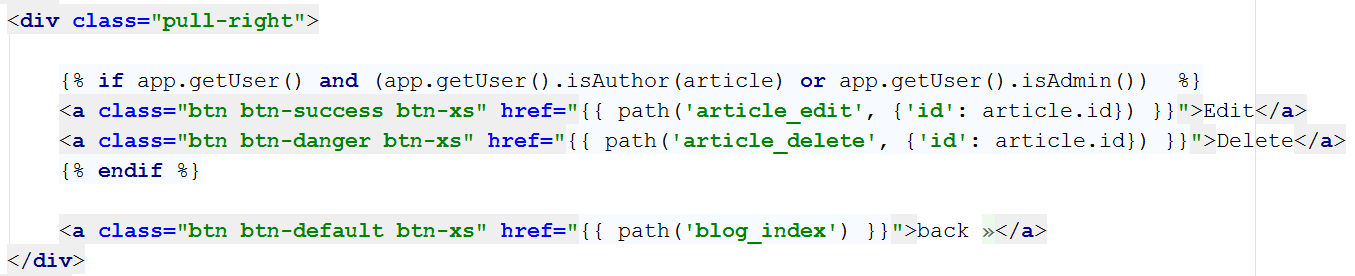
# Creating View Helper Validations

## Create View Helper Validation

Currently, **every user** that opens the **single article view**, will be able to see the "**Edit**" and "**Delete**" buttons. In order to fix that, we will open the article.html.twig template. You should remember the buttons we've added earlier:



In order to **make them invisible** for users that are **not the article author or admins**, we will create an if statement with **twig**:



Let's split that check in two parts. The first part is:



This part will secure that the other check will be **executed** **only if there is a logged-in user**. Without it, if the user is **guest** (not logged-in), **Twig** will **throw** Runtime\_Error. The second part of the check is:

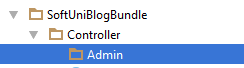


As you've probably figured it out by now, this will check **if the user is author** of the article **or admin**. In the next chapter, we're going to create an admin panel.

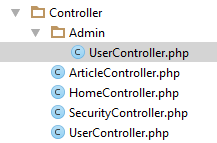
# Creating the Admin Panel

## Listing All Users

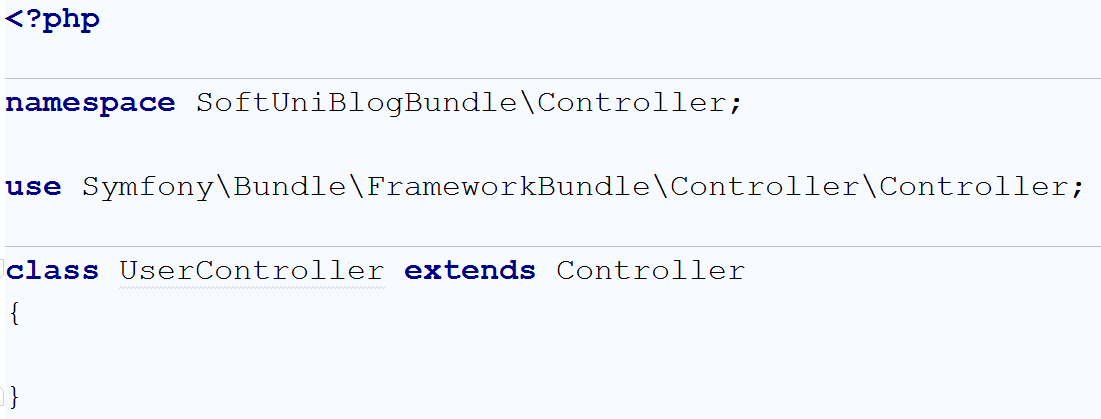
The first thing we are going to create is listing our users. Let's start by **creating folder named** "**Admin**" in the **Controllers** folder:



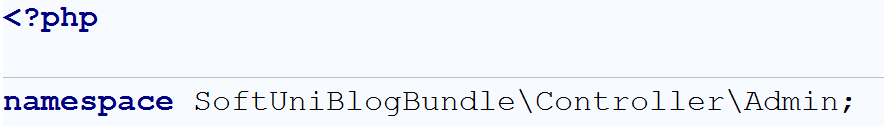
Copy your UserController and paste it in the **Admin** folder. You should receive something like this:



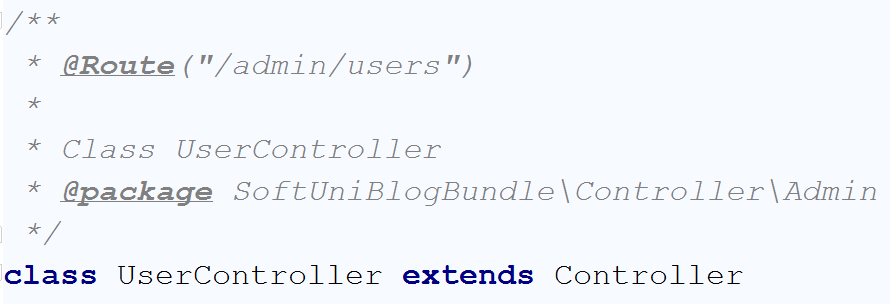
**Open** the new copy that is in the **Admin** folder. **Delete all of its contents**, and you should have something like this:



Edit the namespace, so it contains the "Admin" folder like this:



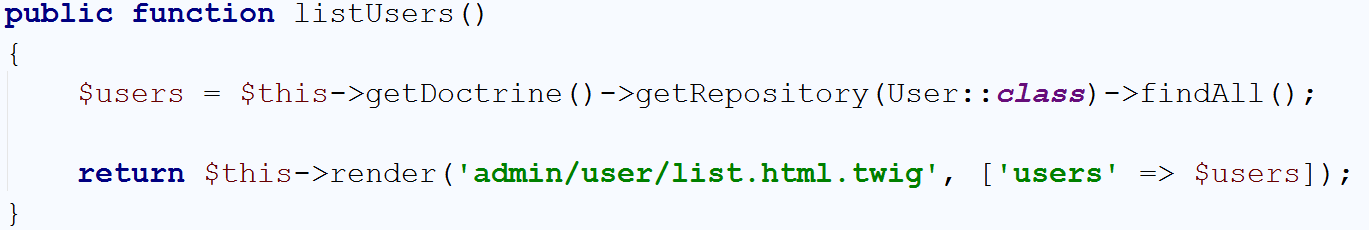
We will use this controller **to manipulate our users**. First we will **create a route for the controller** itself. That means that **every function** inside the controller **will use this route + its own route**:



Let's create our listing function. It will take all users and list them in a table:



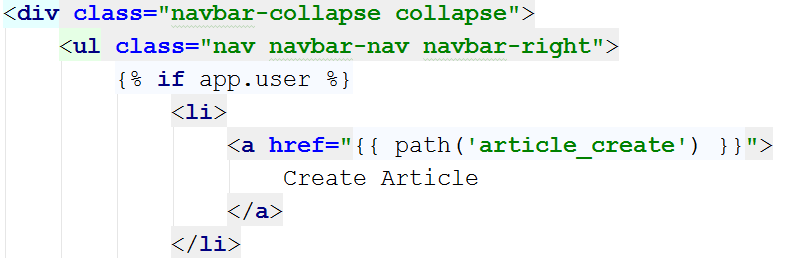
Take a look at the route. Although our **function route is only** "**/**", the **whole route** to access the page **will be** "**/admin/users/**". Our function should only take all users from the database and give them to the view:



Our view will be a **table**. It will show the **admins in blue** color, and the **normal users in gray**:

|  |
| --- |
| {% **extends 'base.html.twig'** %}  {% **block** main %}  <**div class="container body-content"**>  <**div class="well"**>  <**h2**>All Users</**h2**>  <**div class="row"**>  <**table class="table table-striped table-hover "**>  <**thead**>  <**tr**>  <**th**>#</**th**>  <**th**>Full Name</**th**>  <**th**>Email</**th**>  <**th**>Actions</**th**>  </**tr**>  </**thead**>  <**tbody**>  {% **for** user **in** users %}   <**tr**>  <**tr** {% **if** user.isAdmin() %}**class="info"** {% **endif** %}>  <**td**>{{ user.id }}</**td**>  <**td**>{{ user.fullName }}</**td**>  <**td**>{{ user.email }}</**td**>  <**td**>  <**a href="#"**>Edit</**a**>  <**a href="#"**>Delete</**a**>  </**td**>  </**tr**>  {% **endfor** %}  </**tbody**>  </**table**>  </**div**>  </**div**>  </**div**> {% **endblock** %} |

Let's create an **admin button** in the **base layout**. Find the part:



Add the following if statement in the beginning:

|  |
| --- |
| {% **if** app.user.isAdmin() %}  <**li class="dropdown"**>  <**a href="#" class="dropdown-toggle" data-toggle="dropdown" role="button" aria-expanded="false"**>Admin<**span class="caret"**></**span**></**a**>  <**ul class="dropdown-menu" role="menu"**>  <**li**><**a href="**{{ path(**'admin\_users'**) }}**"**>Users</**a**></**li**>  </**ul**>  </**li**> {% **endif** %} |

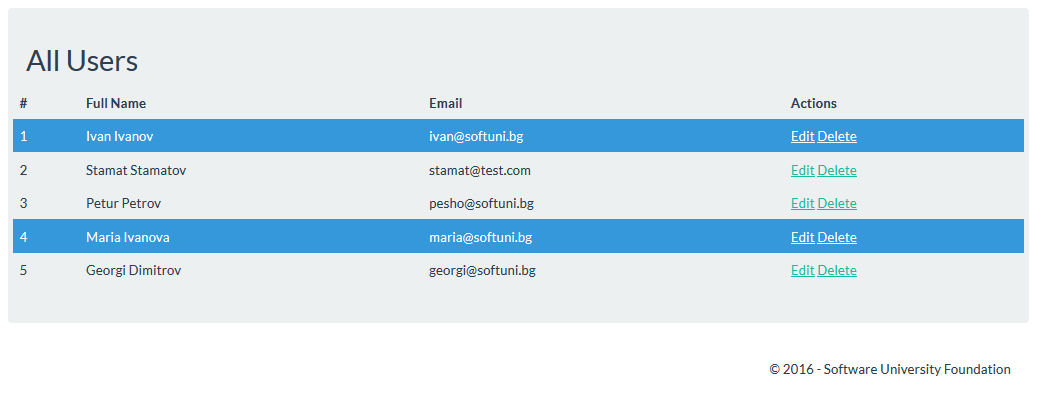
Your code should look like this now:



Now let's go to our home page:



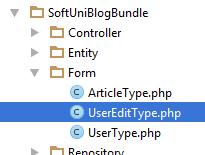
If we are **logged with an admin**, you can see the **admin drop-down menu**. Go to the user listing page:



We can see that I have 2 admins and 3 normal users. Let's create the **edit and delete functionality**.

## Editing User

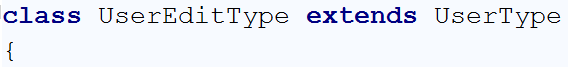
First, we want to answer the question – "What do we want to edit?". The answer is simple – **everything**, including the roles. However, **we can't do that** with the current form called "UserType". We need to create new form, that will include the user roles as well, as everything else. We will call it "UserEditType":



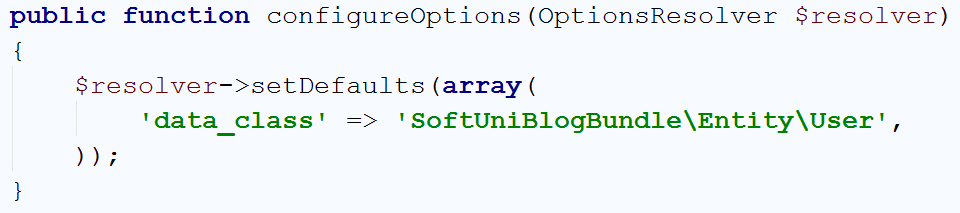
By default, our form will extend something called "AbstractType":

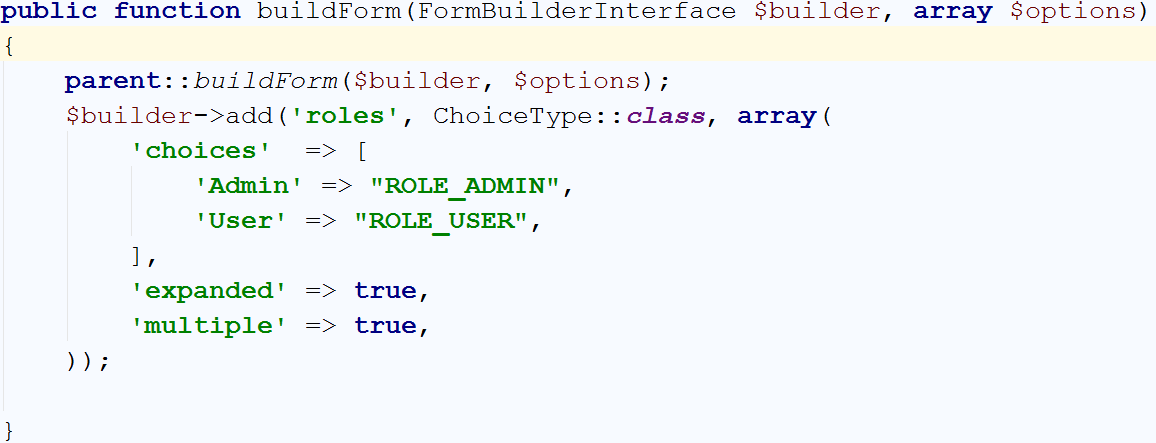


We want to change it, to our custom UserType:



In the configureOptions() function, we need to specify our User entity:



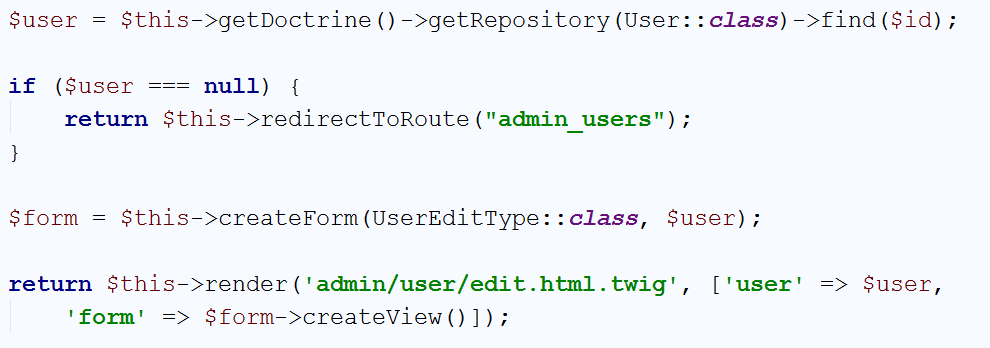
The interesting thing is in the buildForm() function. Let's take a look at it: 

What it does, is take the normal UserType form and **add new field called** "roles" to it. How is this happening? Its called [inheritance](https://en.wikipedia.org/wiki/Inheritance_(object-oriented_programming)). The first line in the function **adds to our builder**, the form that the **UserType form creates**. Then we simply **add the new field**. Let's examine this field. Its field of type ChoiceType and it has several properties. The first one is **associative array** that keeps key-value pairs. The **key is what the user sees** and the **value is what we get** from the form. Then we have the "expanded" and "multiple" properties, which change the type of the field to checkbox. Without them it would either be **drop-down**, **radio button** style or something else.

Now we need to create the function in our controller. This is probably the longest function we're going to write in our program, but let's start with the simple parts:



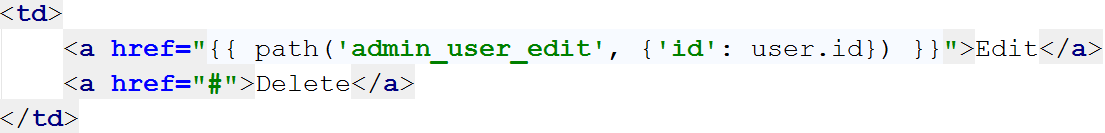
Now we need to get the user, **check if it is existing user** and **create the form**:



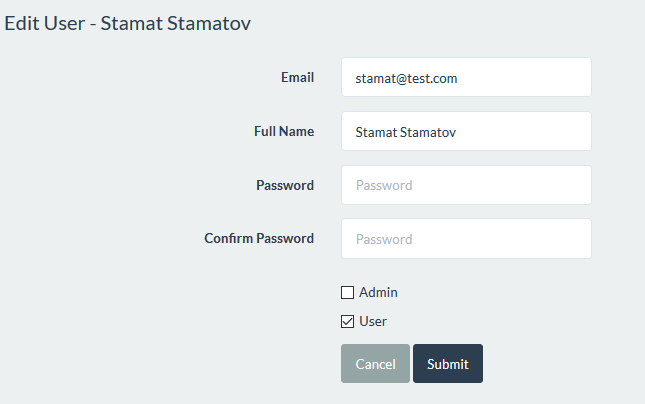
Let's create the view, before we continue with the editing process:

|  |
| --- |
| {% **extends 'base.html.twig'** %}  {% **block** main %}  <**div class="container body-content span=8 offset=2"**>  <**div class="well"**>  <**form class="form-horizontal" action="**{{ path(**'admin\_user\_edit'**,{**'id'** : user.id}) }}**" method="post"**>  <**fieldset**>   <**legend**>Edit User - {{ user.fullName }}</**legend**>  <**div class="form-group"**>  <**label class="col-sm-4 control-label" for="user\_email"**>Email</**label**>  <**div class="col-sm-4 "**>  <**input class="form-control" id="user\_email" value="**{{ user.email }}**" name="user\_edit[email]" required type="email"**>  </**div**>  </**div**>  <**div class="form-group"**>  <**label class="col-sm-4 control-label" for="user\_fullName"**>Full Name</**label**>  <**div class="col-sm-4 "**>  <**input type="text" class="form-control" id="user\_fullName" value="**{{ user.fullName }}**" name="user\_edit[fullName]" required**>  </**div**>  </**div**>  <**div class="form-group"**>  <**label class="col-sm-4 control-label" for="user\_password\_first"**>Password</**label**>  <**div class="col-sm-4"**>  <**input type="password" class="form-control" id="user\_password\_first" placeholder="Password" name="user\_edit[password][first]"**>  </**div**>  </**div**>  <**div class="form-group"**>  <**label class="col-sm-4 control-label" for="user\_password\_second"**>Confirm Password</**label**>  <**div class="col-sm-4"**>  <**input type="password" class="form-control" id="user\_password\_second" placeholder="Password" name="user\_edit[password][second]"**>  </**div**>  </**div**>  <**div class="form-group"**>  <**div class="col-sm-4 col-sm-offset-4"**>  {{ form\_widget(form.roles) }}  </**div**>  </**div**>  {{ form\_row(form.\_token) }}  <**div class="form-group"**>  <**div class="col-sm-4 col-sm-offset-4"**>  <**a class="btn btn-default" href="**{{ path(**'admin\_users'**) }}**"**>Cancel</**a**>  <**button type="submit" class="btn btn-primary"**>Submit</**button**>  </**div**>  </**div**>  </**fieldset**>  </**form**>  </**div**>  </**div**> {% **endblock** %} |

Big piece of code. Let's go back to the **listing view** and **fix the edit button**:

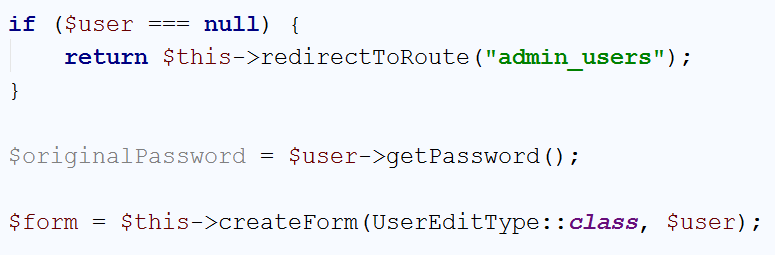


Let's see how it looks:

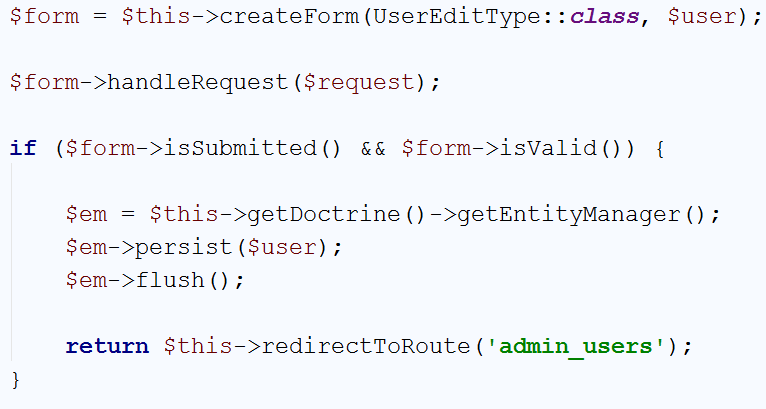


The **email**, **full name** and **roles** are **filled**. That is what we want. But you may ask – why are the password fields empty? Because we have no easy way to decrypt it. Right now, every user is having **his password encrypted** and even if we display it, **no one will be able to understand it**. That’s why we will do something else. We will **tell our controller** to **ignore** the **password fields if they are empty**. Let's make the form work now.

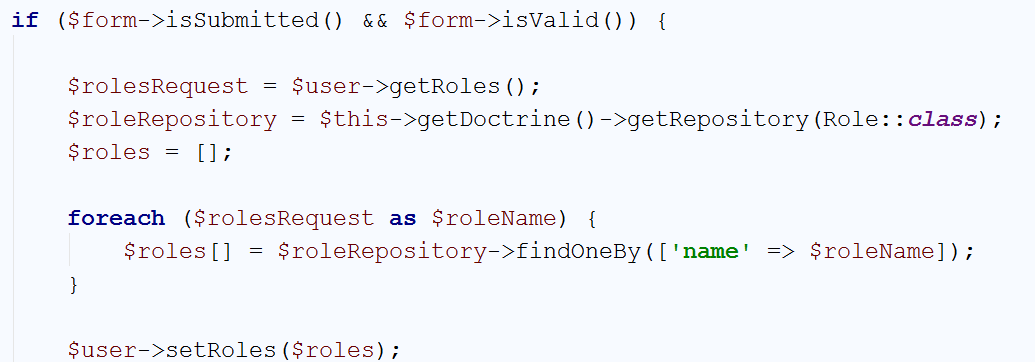
Go back to the controller. The first thing we want to do is to **save the user password before we generate our form**:



Then we can handle our request:



That's not all. We need to get the roles and check the password. In the beginning of the if statement, write the following code:

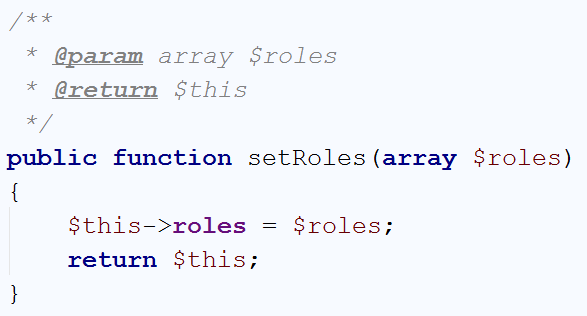


This code will get all roles (as strings) from the user **after we've submitted the form** and foreach one will **find the real Role (entity) from the database**. Then we simply set those roles to the user. Finally let's do the password check:

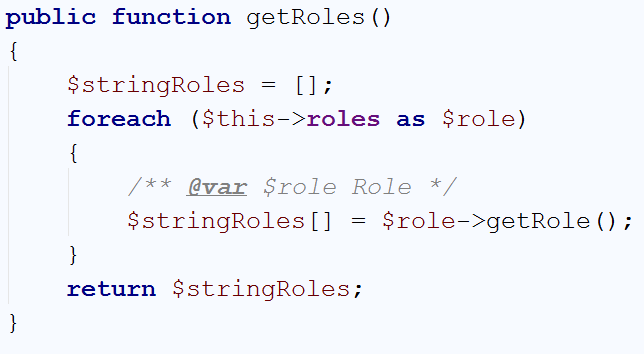


If **we have edited the password** in our form, **it will encrypt it** and set it for the user. **If not**, we will just **use the old password**.

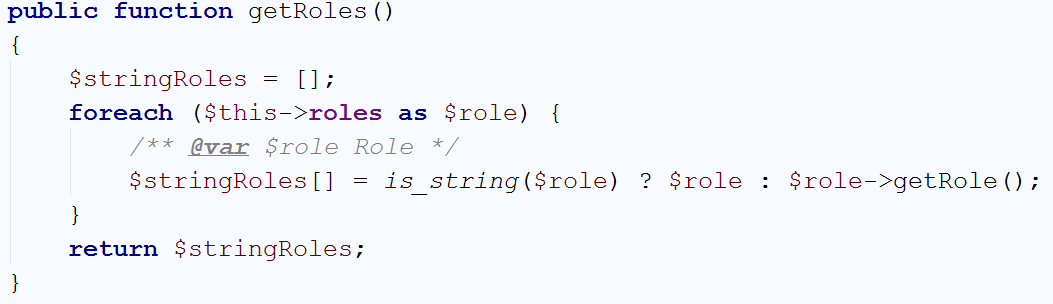
That's not all. If you try to use the form now, you will receive an error message. To fix that, we need to edit our User entity. Let's create a new function on the bottom of the class:



This function will provide us a way to **replace all of the current roles** with **new array of roles**. We will use it with the **roles of the form**. Find our getRoles() function:



And edit it like that:



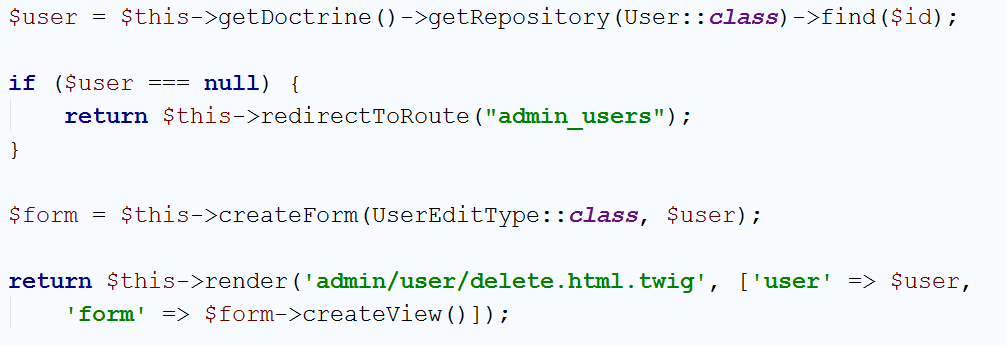
That should be everything. If you test the form now, you can see that it works.

## Deleting User

Editing users is nice, but we may want to delete users as well. To do that we should start with creating new function in our UserController:



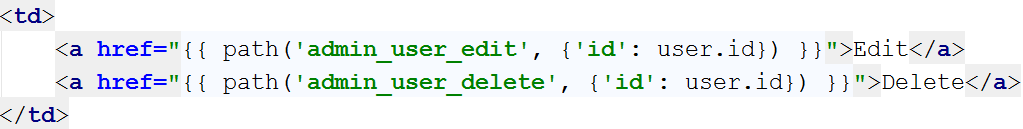
We will start simply by viewing the form:



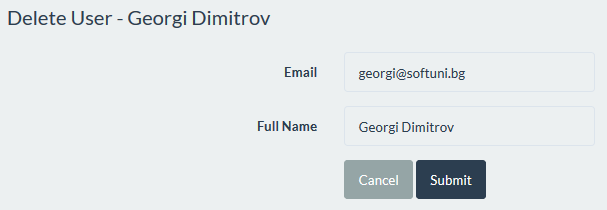
Our view should be something along those lines:

|  |
| --- |
| {% **extends 'base.html.twig'** %}  {% **block** main %}  <**div class="container body-content span=8 offset=2"**>  <**div class="well"**>  <**form class="form-horizontal" action="**{{ path(**'admin\_user\_delete'**,{**'id'** : user.id}) }}**" method="post"**>  <**fieldset**>  <**legend**>Delete User - {{ user.fullName }}</**legend**>  <**div class="form-group"**>  <**label class="col-sm-4 control-label" for="user\_email"**>Email</**label**>  <**div class="col-sm-4 "**>  <**input class="form-control" id="user\_email" value="**{{ user.email }}**" name="user\_edit[email]" required type="email" disabled**>  </**div**>  </**div**>  <**div class="form-group"**>  <**label class="col-sm-4 control-label" for="user\_fullName"**>Full Name</**label**>  <**div class="col-sm-4 "**>  <**input type="text" class="form-control" id="user\_fullName" value="**{{ user.fullName }}**" name="user\_edit[fullName]" disabled**>  </**div**>  </**div**>   {{ form\_row(form.\_token) }}  <**div class="form-group"**>  <**div class="col-sm-4 col-sm-offset-4"**>  <**a class="btn btn-default" href="**{{ path(**'admin\_users'**) }}**"**>Cancel</**a**>  <**button type="submit" class="btn btn-primary"**>Submit</**button**>  </**div**>  </**div**>  </**fieldset**>  </**form**>  </**div**>  </**div**> {% **endblock** %} |

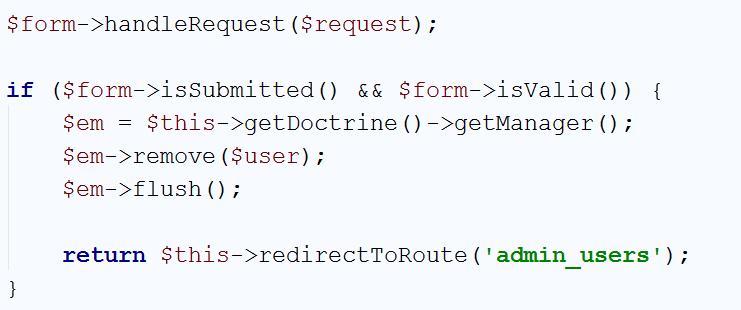
We need to **fix the link** in the table where we list the users:



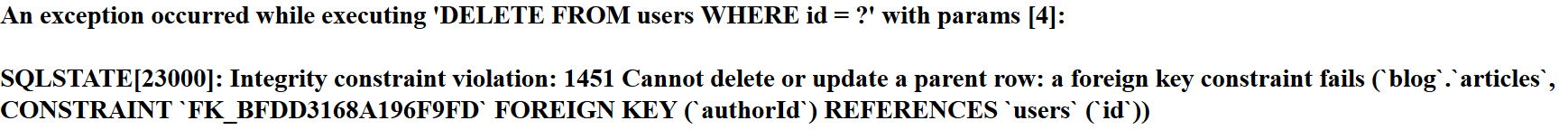
Let's see how it looks:



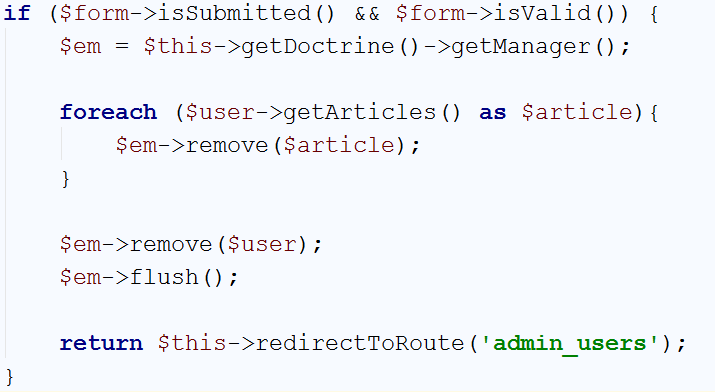
We must **finish the logic** inside the controller. **Handle the request and delete the user**:



If you **try to delete user** that is **not author** of any posts **it** **works just fine**. However, if you **try to delete author of one or more articles**, we will receive an **error message** like this one:



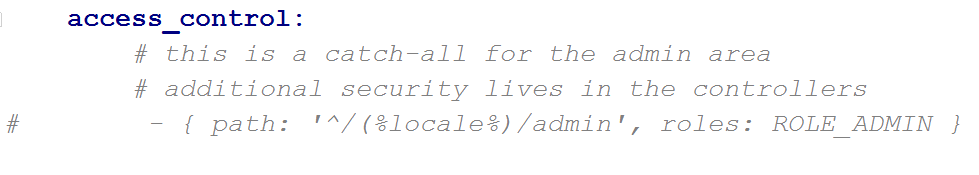
That is because we **first need to delete all the articles** related to the user and **then delete the user** itself. We will do this by editing the code in the function we wrote earlier:



Try it now, it will work like a charm. That is the **edit and delete** for our users **ready**.

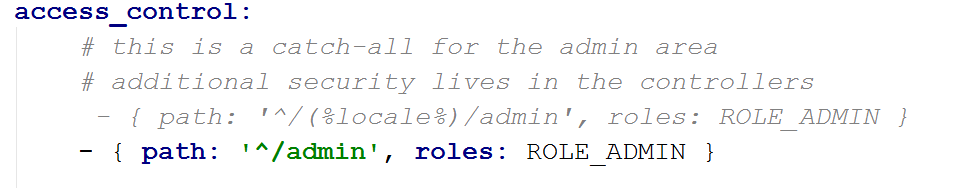
## Limit the Admin Panel to Admins

At the moment, **everyone can access our admin page**. **Normal users can't see the admin drop-down**, but if they **try a routing** i.e. <http://127.0.0.1:8000/admin/users/edit/2> **it will allow them** to edit the user. The fix is really simple. Find the file called security.yml located in "app/config" folder of your project. It will contain many comments and commands, but we are interested in that part:



In this section of the file **we can define areas of our application**, **available to only specific groups of people**.

Write the following code:



This code translates to – **allow only users with the role** "ROLE\_ADMIN" to **access all pages starting** with "/admin". If we try to use the admin panel with regular user, we get the following error:



That's all, **we've created an admin panel** and made it **limited** it to **admins only**.

# Creating Categories

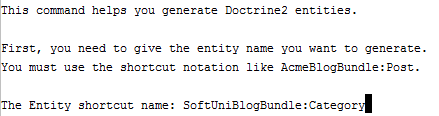
Let's improve our blog a bit. We will create categories for our articles.

## Creating the Category Entity

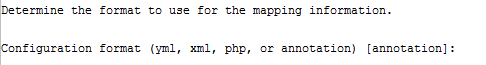
You should be already familiar with the process of **creating entities**, but let's go through it again. Open the Terminal/Command Prompt and write the following:

|  |
| --- |
| php bin/console doctrine:generate:entity |

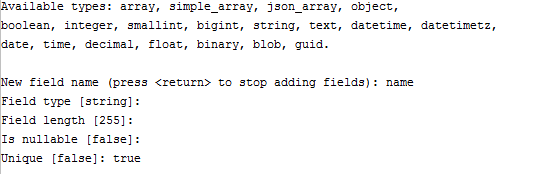
You should remember how to use the wizard, so let's create our **category** entity:



Just press **'Enter'** on the next step:



As for fields, we need only one:



Press **'Enter**' again to exit the wizard. Let's review the newly created entity:



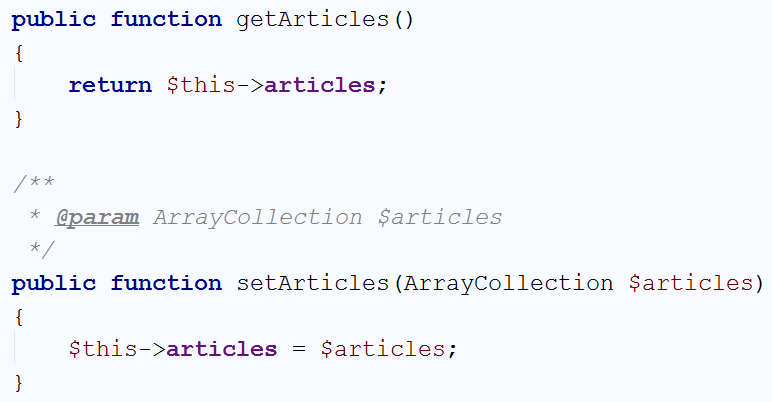
As we've said earlier the **names of our tables should be pluralized**. That means **changing the table name** to "categories". Everything else is okay for now.

## Mapping Categories to Articles

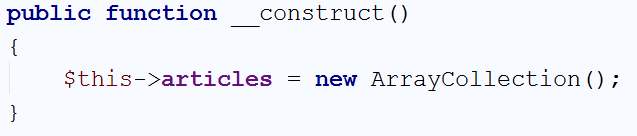
The next thing we want to do is **create our relation**. It will be of type One-To-Many. That means that **one category will contain many articles**. Start by **creating private field** in our Category entity that will contain the articles:



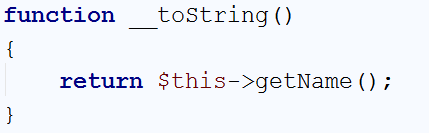
The annotation tells **Doctrine** that we want **one category to be related to many articles**. We will also need getter and setter for it:



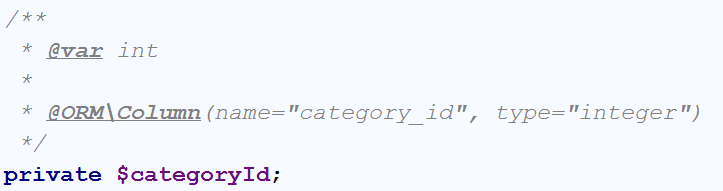
We also need to **initialize** the field in a **constructor**:



The last thing to do is create a \_\_toString() override function that will **return the name of our category**:



Now we need to switch to our Article entity. First of all, we need to **save the id of the category** that our **article belongs to**. We will use **that field to map the category** to the **article**:



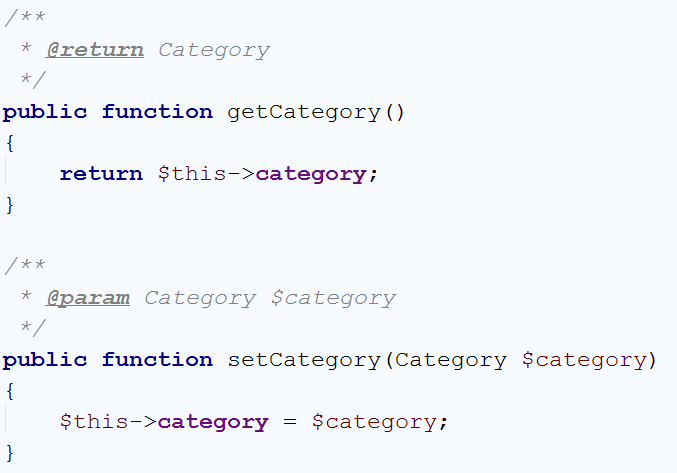
We will obviously need getter and setter for it as well:



Now that we have that, we can now create our field, which will simplify using our category:



You should be familiar with the two annotations you see here, because we've used them earlier. The first one tells **Doctrine** that **many articles will be kept in one category**. The second one specifies the field in the Category that will **correspond to our** "category\_id" in the database. We will again also need getter and setter:



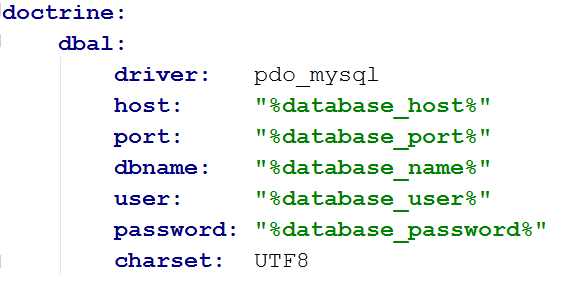
This should be all for this part. Now we need to edit the database.

## Editing the Database

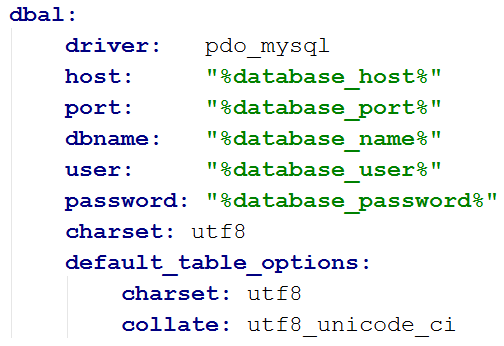
We will delete our whole database now. All users, articles and roles. Open the console interface you are using and write the following command:

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

This command deleted our database. Now we want to **edit the file** called "config.yml" located in "app/config". Find the section containing this code:



We will do slight change to it. Edit the last lines like that:



We're just **changing the database, to support more symbols** (Cyrillic, Chinese, etc.). Now write the following command in the terminal:

|  |
| --- |
| php bin/console doctrine:database:create |

The result should be:



Finally, write the following command that you should be already familiar with:

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

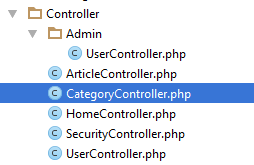
That should result in:



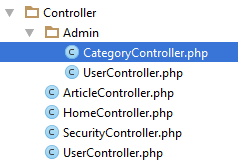
Don't forget to create your roles in the roles tables as we did in chapter 8 – part 7. That's all for recreating the DB.

## Listing Categories in the Admin Panel

Right now, **we can't create** **articles**. That is a huge problem, having in mind that we're creating a blog. In order to fix that we **need to create at least one category** and give the user option to choose in which category he wants to post his article. Let's create new CategoryController in our "Admin" folder. When we create the controller, it will be placed in the default "Controller" folder and we need to move it:

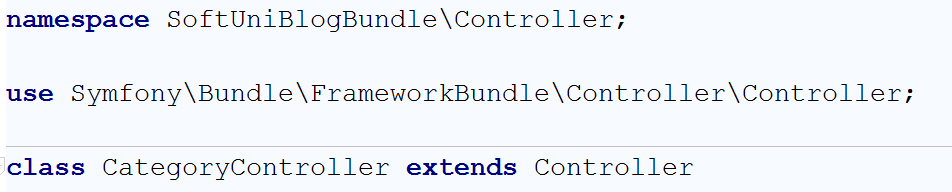


After we move it, we should have that:

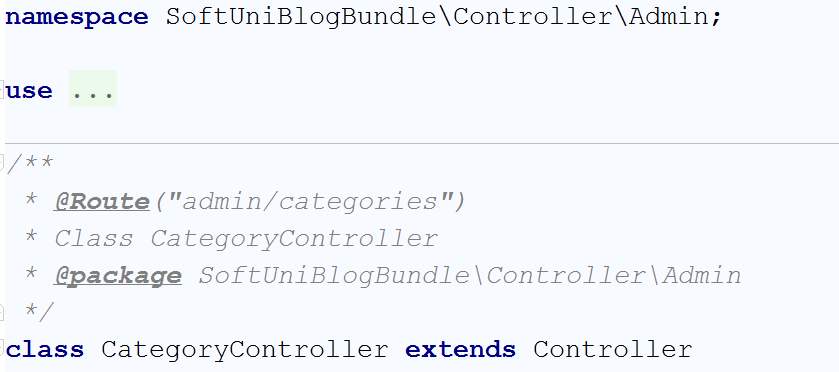


You may ask why do we want the controller to be in our admin section. Because we want **only** **admins** to be able to **create**, **update** or **delete** categories.

Let’s start with the definition of the controller:



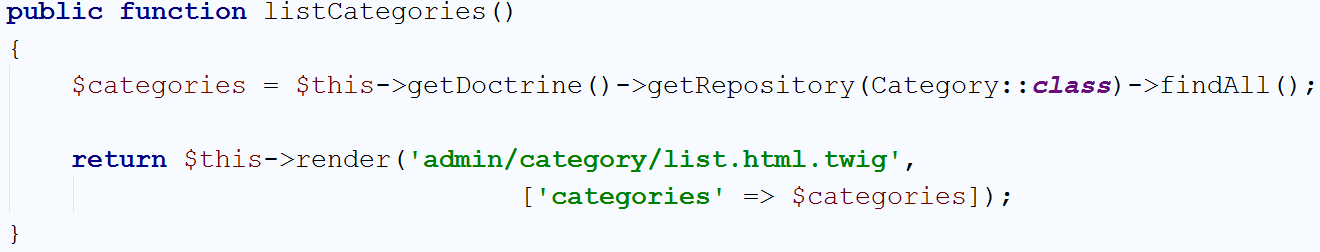
First of all, we need to fix the namespace like we did with the UserController. Then we will create **default routing for the categories** as well:



Now we need to **create the first function**, that will **list all of our existing** **categories**:



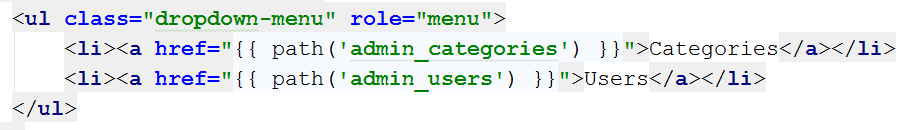
Its job will be to **get all categories** from the database and **send them to our view**:



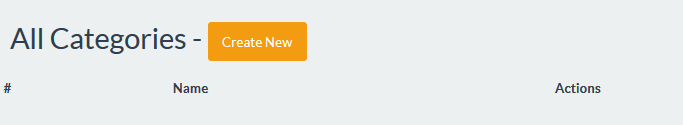
Our view will be really similar to the users listing view:

|  |
| --- |
| {% **extends 'base.html.twig'** %}  {% **block** main %}  <**div class="container body-content"**>  <**div class="well"**>  <**h2**>All Categories -  <**a href="#" class="btn btn-warning"**>Create New</**a**>  </**h2**>  <**div class="row"**>  <**table class="table table-striped table-hover "**>  <**thead**>  <**tr**>  <**th**>#</**th**>  <**th**>Name</**th**>  <**th**>Actions</**th**>  </**tr**>  </**thead**>  <**tbody**>  {% **for** category **in** categories %}   <**tr**>  <**tr**>  <**td**>{{ category.id }}</**td**>  <**td**>{{ category.getName() }}</**td**>  <**td**>  <**a href="#"**>Edit</**a**>  <**a href="#"**>Delete</**a**>  </**td**>  </**tr**>  {% **endfor** %}  </**tbody**>  </**table**>  </**div**>  </**div**>  </**div**> {% **endblock** %} |

Now we want to add the listing function to our **admin drop-down** in the base **view**:



Let's start it and see how it looks:



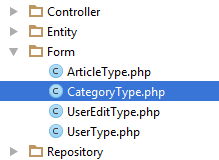
Pretty standard, but it will do the job. Next step – Creating categories.

## Creating Categories

You probably can imagine that we need to **create new function** in the CategoryController, that will **create the categories**:



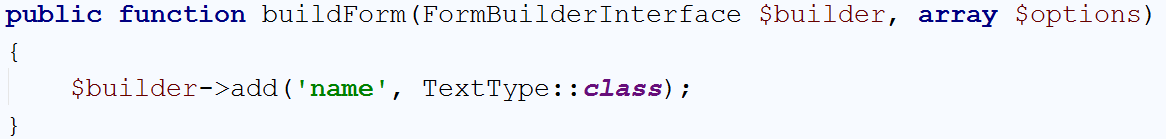
Before we continue, we need to **create a form type** for our **category**. Let's create a **new form** and call it "CategoryType":



Everything in the form is pretty simple. We will start the configureOptions() function:

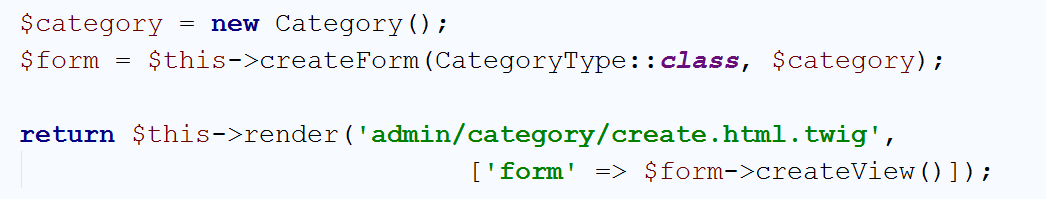


This will just set the **default class** that **the form is going to fill** to be of type Category. Then we have the buildForm() function:



The form will contain only **one field** of type TextType, that we are going to use for the **category name**.

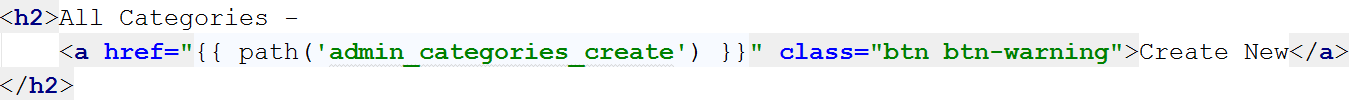
We can go **back to our controller** and more specifically to the **create function**. We will use something like that to render the view:



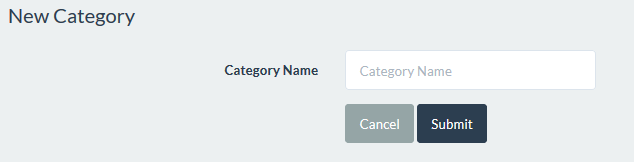
And the view will be simple enough as well:

|  |
| --- |
| {% **extends 'base.html.twig'** %}  {% **block** main %}  <**div class="container body-content span=8 offset=2"**>  <**div class="well"**>  <**form class="form-horizontal" action="**{{ path(**'admin\_categories\_create'**) }}**" method="POST"**>  <**fieldset**>  <**legend**>New Category</**legend**>   <**div class="form-group"**>  <**label class="col-sm-4 control-label" for="article\_title"**>Category Name</**label**>  <**div class="col-sm-4 "**>  <**input type="text" class="form-control" id="article\_title" placeholder="Category Name" name="category[name]"**>  </**div**>  </**div**>   {{ form\_row(form.\_token) }}   <**div class="form-group"**>  <**div class="col-sm-4 col-sm-offset-4"**>  <**a class="btn btn-default" href="**{{ path(**'admin\_categories'**) }}**"**>Cancel</**a**>  <**button type="submit" class="btn btn-primary"**>Submit</**button**>  </**div**>  </**div**>  </**fieldset**>  </**form**>  </**div**>  </**div**> {% **endblock** %} |

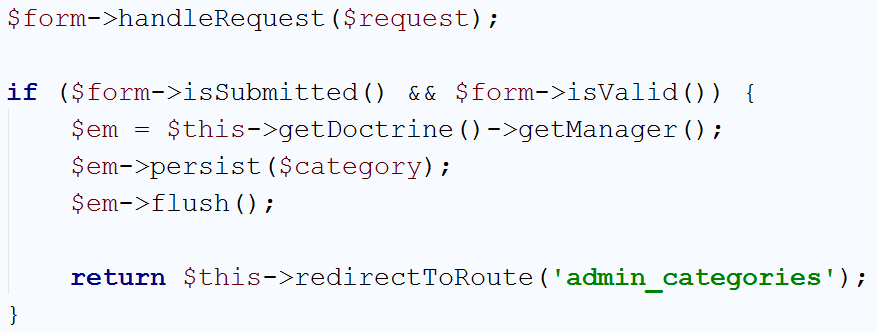
We should fix the link on **the create button in the listing view** now:



And when we click it, we should see this form:



The only thing left is to **handle the form** and **add the category to the database**:



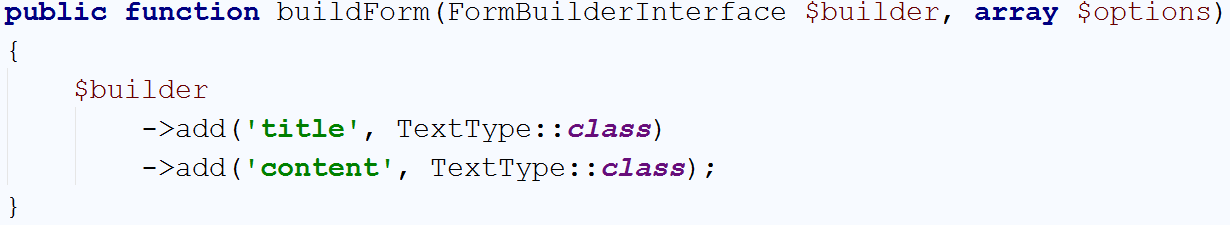
Now you should be able to create categories and list them like that:



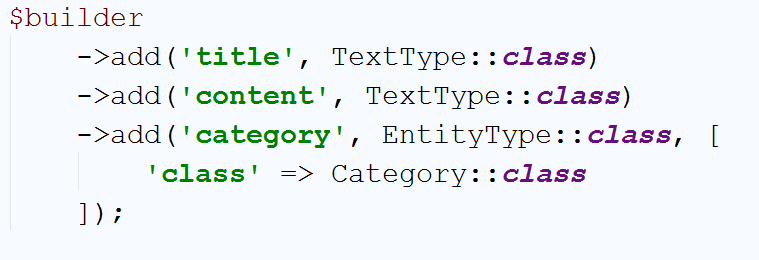
Now we can finally fix the article creation function.

## Article Creation

We need to open the ArticleType form that we’ve created earlier. The buildForm() function should look like that:



We will add another field, which will be of a new type, that you haven’t seen before:



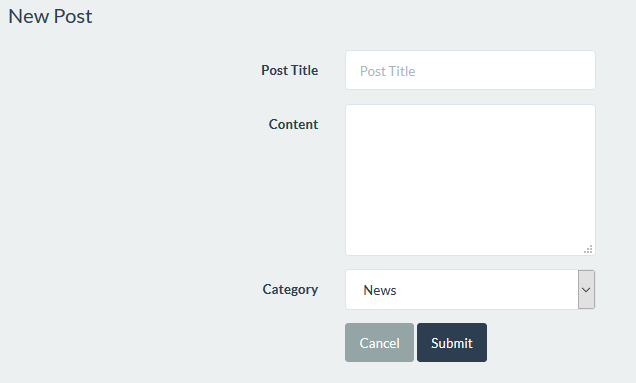
This type is a special one. It defines from which Entity our **form will get data**. We are telling it to **create a drop-down menu** from our Category entity. The last thing left to do is to **edit our article create view**. Find this part of the code:



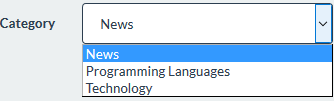
Exactly above it, insert the following code:

|  |
| --- |
| <**div class="form-group"**>  <**label class="col-sm-4 control-label" for="article\_category"**>Category</**label**>  <**div class="col-sm-6"**>  {{ form\_widget(form.category) }}  </**div**> </**div**> |

That’s all! Now we can **create articles again**. The form now looks like this:



If we **expand the drop-down**, we can see that **all of our categories are in there**:



The next step is to **change** our **home page**.

## Listing All Categories on the Home Page

Now that we have categories, we want to change our home page quite a bit. We want **all categories** to be **shown on the front page** as **hyperlinks** and when you **click on a category** to get redir**ected to a view which will hold all articles in the given category**.

First of all, we want to edit our HomeController. We want to create a **new function**:



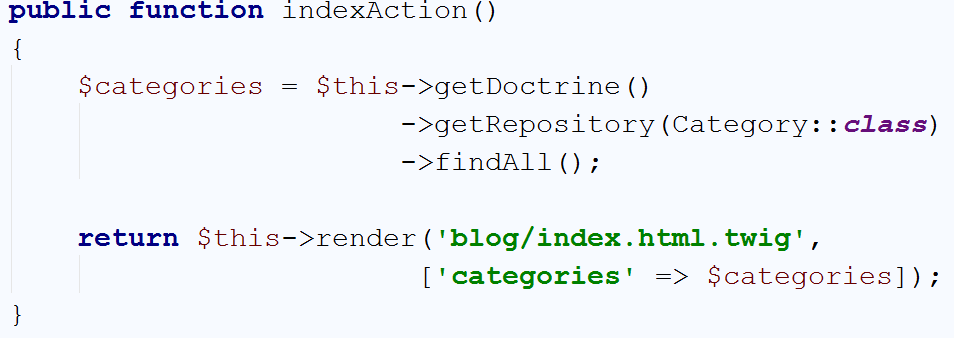
The idea behind it, is to create a new view, which will display **all of the articles** that are **in** the **selected category**. The view should look like this:

|  |
| --- |
| {% **extends 'base.html.twig'** %}  {% **block** main %}  <**div class="container body-content"**>  <**div class="row"**>  {% **for** article **in** articles %}  <**div class="col-md-6"**>  <**article**>  <**header**>  <**h2**>{{ article.title }}</**h2**>  </**header**>   <**p**>  {{ article.summary }}  </**p**>   <**small class="author"**>  {{ article.author }}  </**small**>   <**footer**>  <**div class="pull-right"**>  <**a class="btn btn-default btn-xs"  href="**{{ path(**'article\_view'**, {**'id'**: article.id}) }}**"**>Read more **&raquo;**</**a**>  </**div**>  </**footer**>  </**article**>  </**div**>  {% **endfor** %}  </**div**>  </**div**> {% **endblock** %} |

Its **identical** with what we have currently in the "blog/index" view. We will change that now. Go back to your HomeController and find the indexAction() function:



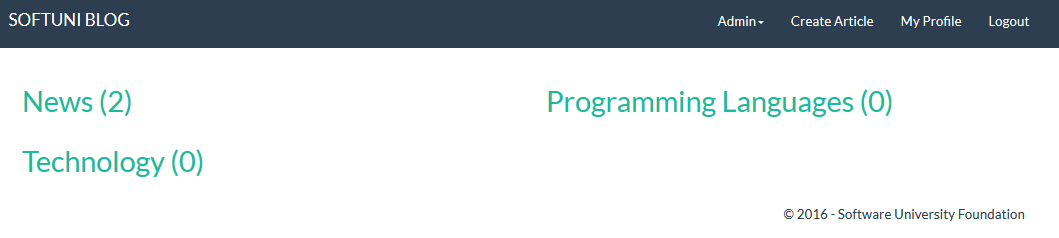
We will change that to:



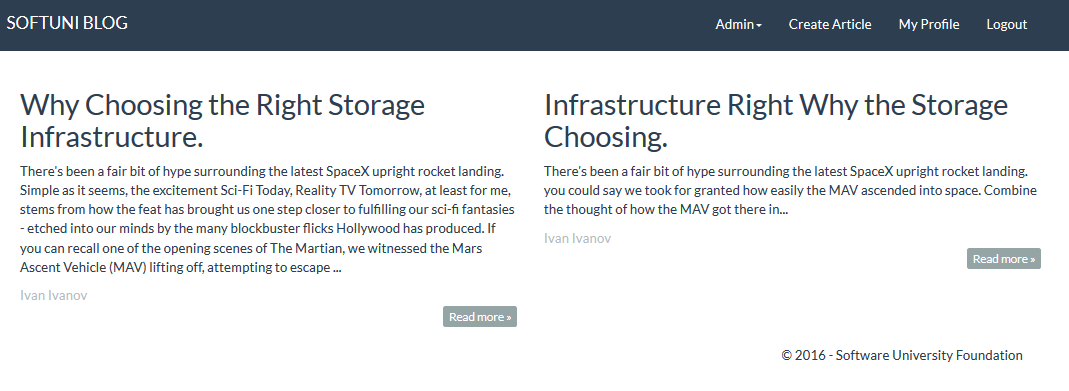
As you can see, **instead** of **sending all articles** to the view, we are **sending all categories**. We will edit the "blog/index" view to look like this:

|  |
| --- |
| {% **extends 'base.html.twig'** %}  {% **block** main %}  <**div class="container body-content"**>  <**div class="row"**>  {% **for** category **in** categories %}  <**div class="col-md-6"**>  <**header**>  <**h2**>  <**a href="**{{ path(**'category\_articles'**, {**'id'**: category.id}) }}**"**>  {{ category.name }} ({{ category.articles.count }})  </**a**>  </**h2**>  </**header**>  </**div**>  {% **endfor** %}  </**div**>  </**div**> {% **endblock** %} |

If we run the blog right now, we should see this:



The **number** **next** to each **category name** is the **count of articles** we have in the category. If we **click on a category**, we should **see** the **articles**, like we did on the **old home page**:



But what if we want to **edit or delete an article**?

## Article Edit and Delete

If we want to **edit or delete articles**, we want to only **add** the **drop-down for our categories** to the **views**.

In the view "article/edit" find the row:



Above it, add the following code:

|  |
| --- |
| <**div class="form-group"**>  <**label class="col-sm-4 control-label" for="article\_category"**>Category</**label**>  <**div class="col-sm-6"**>  {{ form\_widget(form.category) }}  </**div**> </**div**> |

This will allow us to **edit the category** of a **given article**. To **delete** a **given article**, we will open the "article/delete" view and again find the row:



Above it, add the following code:

|  |
| --- |
| <**div class="form-group"**>  <**label class="col-sm-4 control-label" for="article\_category"**>Category</**label**>  <**div class="col-sm-6"**>  {{ form\_widget(form.category, { **'attr'**:{**'readonly'**:**'readonly'**}}) }}  </**div**> </**div**> |

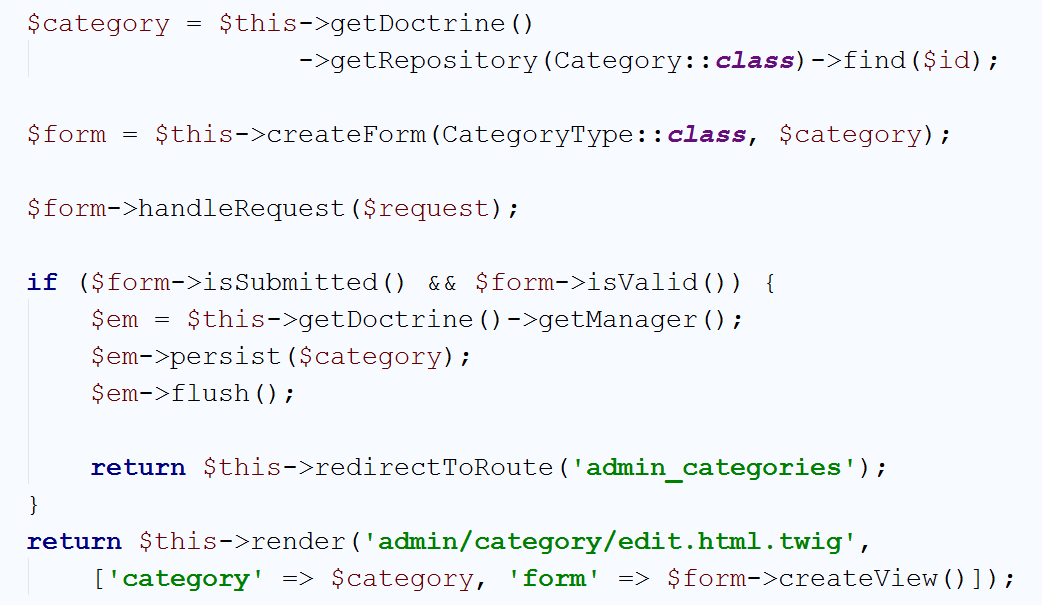
And **this** **will** **allow us to delete an article** as well. What if we want to **edit a category**? The answer will wait for you in the next part of this chapter. ☺

## Editing Categories

It's time to create the **edit** **function (**CategoryController**)**:



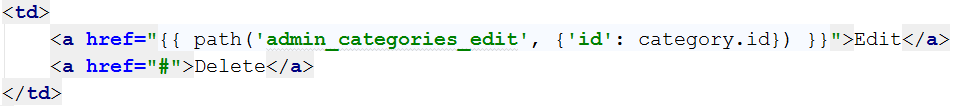
This time we will create all of the code at once:



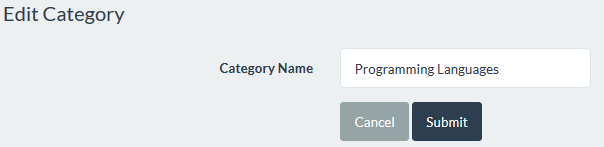
We **get the current category** and **create the category form**. If the **form** is **submitted** and **valid**, we **edit** the **category** in the **database**. The **view** will be the **create view** with filled **category name**:

|  |
| --- |
| {% **extends 'base.html.twig'** %}  {% **block** main %}  <**div class="container body-content span=8 offset=2"**>  <**div class="well"**>  <**form class="form-horizontal" action="**{{ path(**'admin\_categories\_edit'**, {**'id'**: category.id}) }}**" method="POST"**>  <**fieldset**>  <**legend**>Edit Category</**legend**>   <**div class="form-group"**>  <**label class="col-sm-4 control-label" for="article\_title"**>Category Name</**label**>  <**div class="col-sm-4 "**>  <**input type="text" class="form-control" id="article\_title" placeholder="Category Name" name="category[name]" value="**{{ category.name }}**"**>  </**div**>  </**div**>   {{ form\_row(form.\_token) }}   <**div class="form-group"**>  <**div class="col-sm-4 col-sm-offset-4"**>  <**a class="btn btn-default" href="**{{ path(**'admin\_categories'**) }}**"**>Cancel</**a**>  <**button type="submit" class="btn btn-primary"**>Submit</**button**>  </**div**>  </**div**>  </**fieldset**>  </**form**>  </**div**>  </**div**> {% **endblock** %} |

Finally, we should **fix** **the** **hyperlink** in our **listing view**:



If we try to edit a category, we should see this:



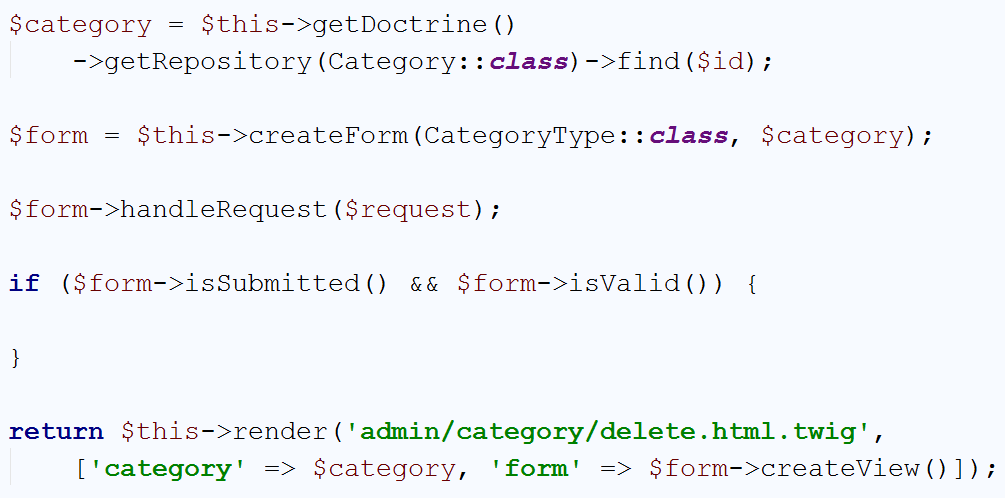
That's all, the **last** **part** that we have in this chapter is the **category deletion**.

## Deleting Categories

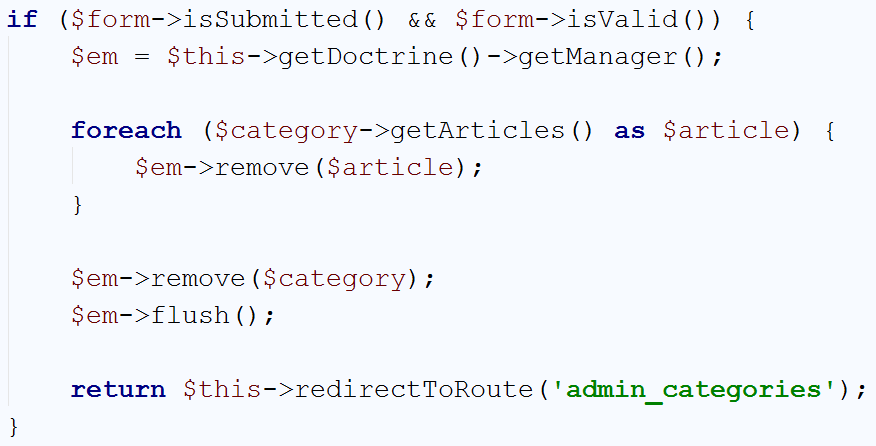
Like in the **edit** **part**, we will start with the function in our CategoryController:



This time, we will split the code in two parts. The first part is **creating and handling the form** and **sending it to the view**:



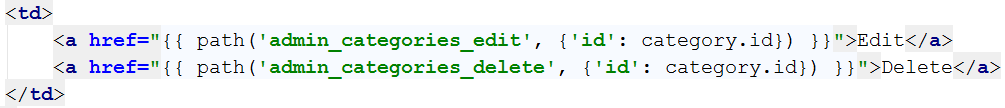
We've seen **code similar to this** in **all functions** in **our other controllers**. Now we need the logic that will **delete the category**:



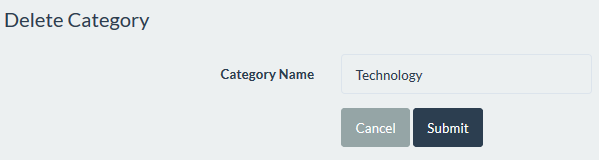
The special part here that we **delete all articles** that are **inside of our category**. We **cannot delete** the **category** if it's **not** **empty**. Now we only need to create the **delete view**:

|  |
| --- |
| {% **extends 'base.html.twig'** %}  {% **block** main %}  <**div class="container body-content span=8 offset=2"**>  <**div class="well"**>  <**form class="form-horizontal" action="**{{ path(**'admin\_categories\_delete'**, {**'id'**: category.id}) }}**"  method="POST"**>  <**fieldset**>  <**legend**>Delete Category</**legend**>   <**div class="form-group"**>  <**label class="col-sm-4 control-label" for="article\_title"**>Category Name</**label**>  <**div class="col-sm-4 "**>  <**input type="text" class="form-control" id="article\_title" placeholder="Category Name" name="category[name]" value="**{{ category.name }}**" disabled**>  </**div**>  </**div**>   {{ form\_row(form.\_token) }}   <**div class="form-group"**>  <**div class="col-sm-4 col-sm-offset-4"**>  <**a class="btn btn-default" href="**{{ path(**'admin\_categories'**) }}**"**>Cancel</**a**>  <**button type="submit" class="btn btn-primary"**>Submit</**button**>  </**div**>  </**div**>  </**fieldset**>  </**form**>  </**div**>  </**div**> {% **endblock** %} |

And finally, **fix the hyperlink** in the **listing view**:



Let's how it looks:



It works perfectly! We're ready with this chapter as well.

# Creating Tags

## Creating the Tag Entity

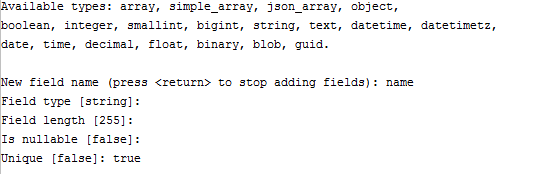
Creating the Tag entity is really similar to the Category entity. It will contain **only one field** called "name", which **must be unique**. Open the Terminal/Command Prompt and write the following:

|  |
| --- |
| php bin/console doctrine:generate:entity |

Create our **Tag** entity:



When it comes to fields, we've said it earlier, we need only one:



Press **'Enter**' again to exit the wizard. Let's review the newly created entity:

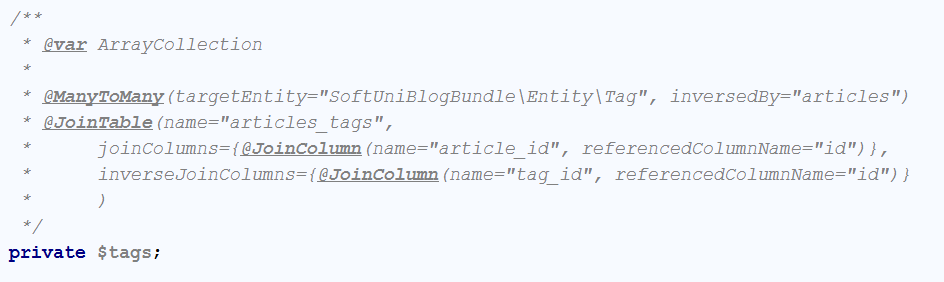


Again - The **names of our tables must be pluralized**. That means that we need to change the **table name** to "tags".

## Mapping Tags to Articles

The relation between tags and articles should be Many-To-Many. That is the same relation we've used in to create the Users-Roles Many-To-Many relation. Many Tags will be used in many Articles. The relation should be defined in the Article entity, and the Tag entity should only keep the mappedBy and targetEntity properties of the @ORM\ManyToMany() annotation.

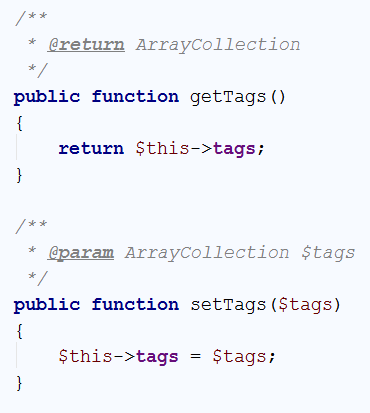
Let's start with the Article entity. First we want to define new field that will keep our tags:



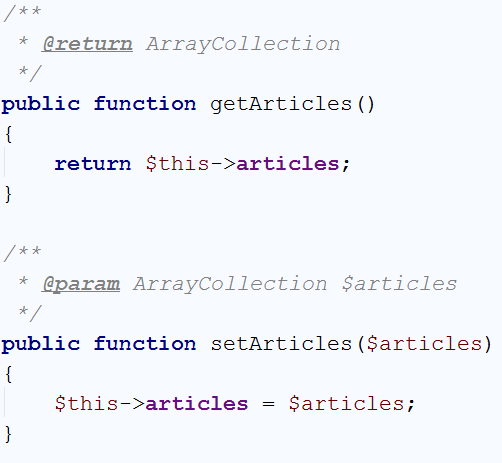
The relation will be established **when we create the following field** in the Tag entity:



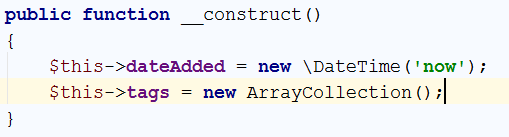
Now we need to create the respective getters and setters for the fields. The tags field in the Article entity:



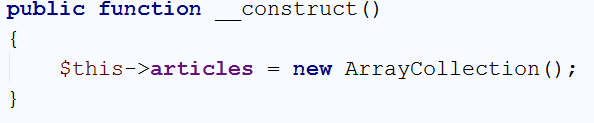
And the articles field in the Tag entity:



The last thing to do is **initialize the fields** in the respective **constructors**. The **Article** entity:



Create a constructor for the Tag entity:



To print, Edit and Del Tags we must write the following function in the Tag entity:

**public function** \_\_toString()  
{  
 **return** $this->getName();  
}

Finally, we want to **update the database**:

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

The result should be:



Now we can start using our tags.

## Creating Tags

The Tag creation process should happen when you create article. That means that the Article create view will have one more input field of type text that will keep all of the tags with comma between each tag. That field **won't be** in the Article **form type**, and because of that we will get the tags using the "$request->get();" syntax. Then you should **split the string** that you receive by '**,**' and **(space)** and then save each tag in the database.

We will start by opening the "article/create" view and finding this part:



Above it, add the following code:

|  |
| --- |
| <**div class="form-group"**>  <**label class="col-sm-4 control-label" for="article\_tags"**>Tags</**label**>  <**div class="col-sm-4 "**>  <**input type="text" class="form-control" id="article\_tags" placeholder="Tags"  name="tags"**>  </**div**> </**div**> |

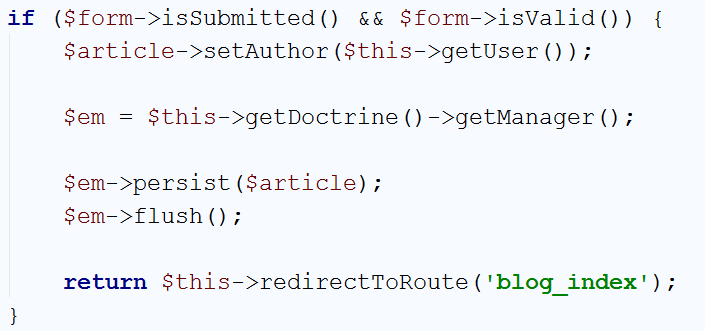
It will create a new text field in our create form.

Let's go to our ArticleController. Let's create a new function that will take the **string from our textbox** and **extract the tags**. Then it should simply **return the tags**:

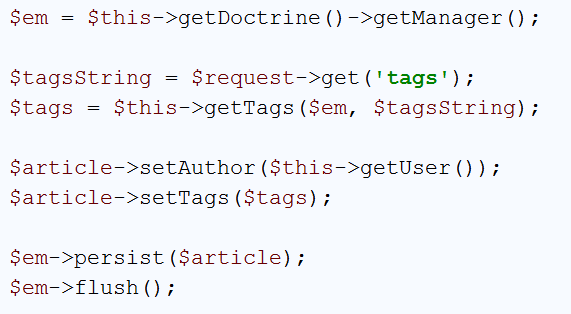


With this piece of code, we split all tags by ',' and then we are **iterating** over the **string array** we have. If **there is existing tag in the database** with the **current name**, we are **taking it**. Otherwise, we just **create a new tag** and **insert it to the database**. Finally, return the tags.

Now, find the create() function. Right now, in the if statement we have:



We need to **edit** the code. We will get the EntityManager first, then **get the tags**. Then we will **set the tags and author** to our article. Finally, we will persist the **article** in the database:



That’s all when it comes to creating tags. You can **create few articles**. Check if the tags are created in the database.

## Editing and Deleting Tags

When you **edit or delete tags**, you should do it when you **edit or delete articles**. Go to the "article/edit" view and find the following line:



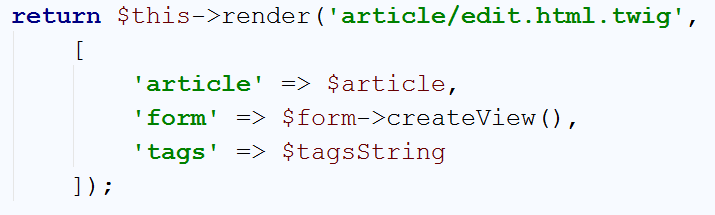
Above it, add the following code:

|  |
| --- |
| <**div class="form-group"**>  <**label class="col-sm-4 control-label" for="article\_tags"**>Tags</**label**>  <**div class="col-sm-4 "**>  <**input type="text" class="form-control" id="article\_tags" name="tags" value="**{{ tags }}**"**>  </**div**> </**div**> |

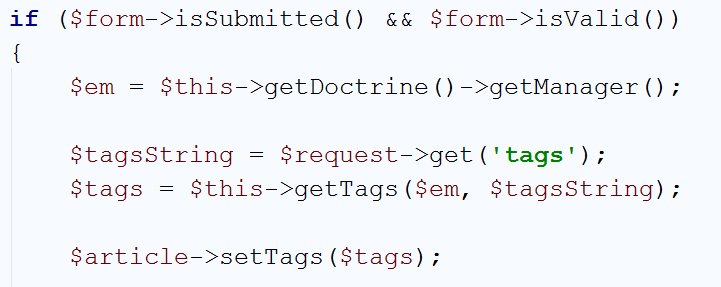
Now, in the editArticle() function of ArticleController we want to do the following:



This will join our tags to a string (string.Join(", ", tags) in C#). In the end of the function we will **send the string to our view**:



If the form is **submitted** and **valid**, we should do the **same thing** we've done in the **create** function:



Now **you can edit** your **tags**. If you want to **delete the article** you only need to **display them in the view**.

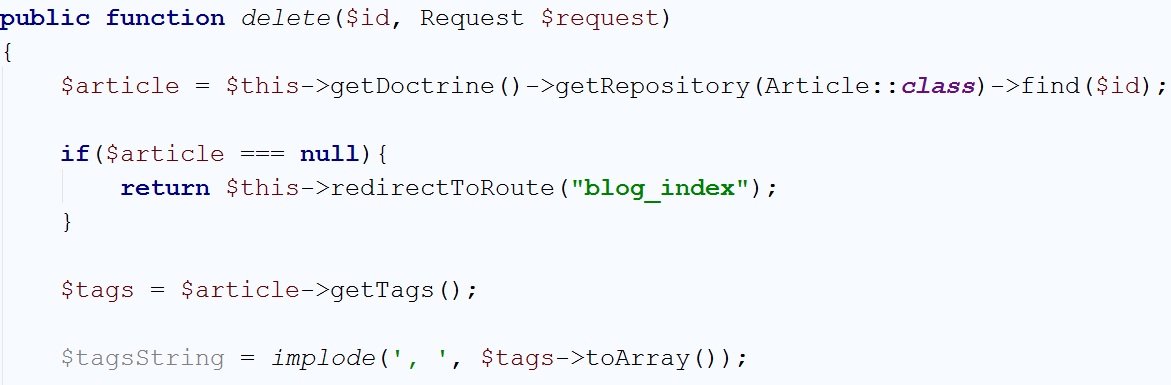
Go to the "article/delete" view and find the following line:



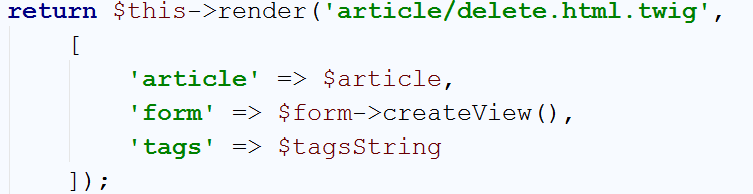
Above it, add the following code:

|  |
| --- |
| <**div class="form-group"**>  <**label class="col-sm-4 control-label" for="article\_tags"**>Tags</**label**>  <**div class="col-sm-4 "**>  <**input type="text" class="form-control" id="article\_tags" name="tags" value="**{{ tags }}**" disabled**>  </**div**> </**div**> |

In the delete() function join the tags like you've done in the **edit**:



Then simply **send the string to the view**:



We don't need to do anything else. The **only thing left** now is to **search all articles with a given tag**.

## Searching by Tags

Here, you should print **each article with its tags** **and create hyperlink for each tag**. Those hyperlinks will lead to a **new view** that will **render only the articles that contain that tag**.

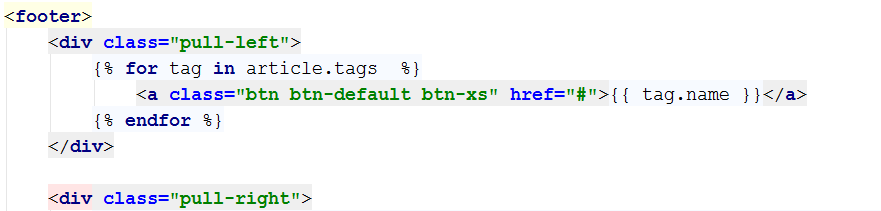
First of all, we should display **all tags for each article**. We will do that in the **single article view** first.



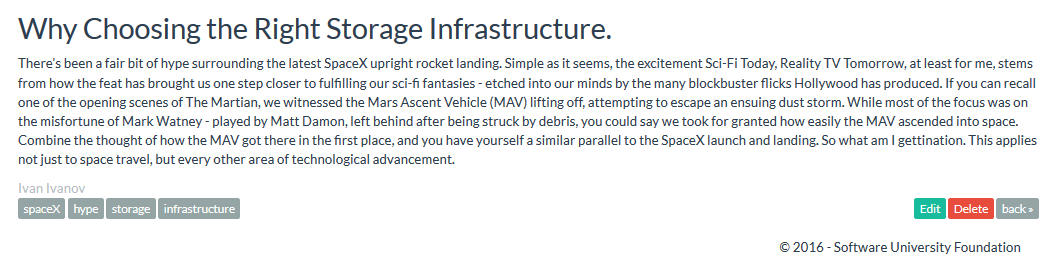
You should be familiar with this part of the code. In the beginning of the footer we will add new <div> that will contain our tags:

|  |
| --- |
| <**div class="pull-left"**>  {% **for** tag **in** article.tags %}  <**a class="btn btn-default btn-xs" href="#"**>{{ tag.name }}</**a**>  {% **endfor** %} </**div**> |

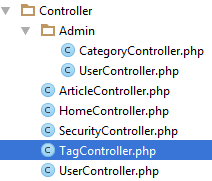
The end result should look like this:



If we open an article now, we should see something like this:



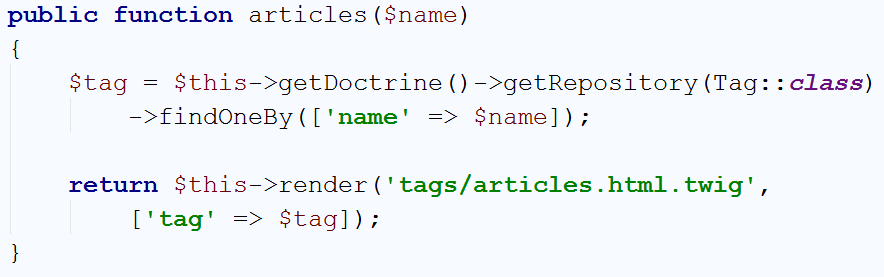
In the **lower left part** of the article you can see the tags. The links are **not working** for now, but we will fix that. First, let's create new TagController:



Delete the function that’s inside. We will **define our own function**:



The **code inside** will be really simple:

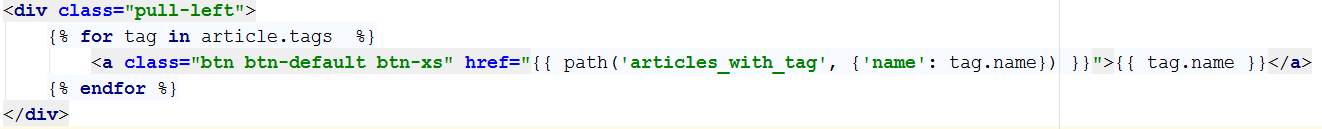


Basically, it means get the **tag** and give it to the view.

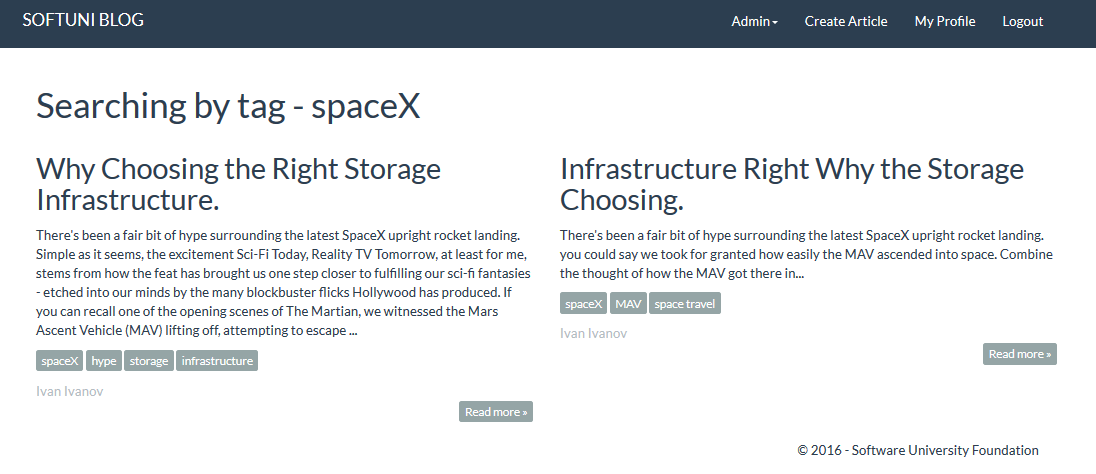
Our view will have the following code:

|  |
| --- |
| {% **extends 'base.html.twig'** %}  {% **block** main %}  <**div class="container body-content"**>  <**h1**>Searching by tag - {{ tag.name }}</**h1**>  <**div class="row"**>  {% **for** article **in** tag.articles %}  <**div class="col-md-6"**>  <**article**>  <**header**>  <**h2**>{{ article.title }}</**h2**>  </**header**>   <**p**>  {{ article.summary }}  </**p**>  <**p**>  {% **for** tag **in** article.tags %}  <**a class="btn btn-default btn-xs"  href="**{{ path(**'articles\_with\_tag'**, {**'name'**: tag.name}) }}**"**>{{ tag.name }}</**a**>  {% **endfor** %}  </**p**>  <**small class="author"**>  {{ article.author }}  </**small**>   <**footer**>  <**div class="pull-right"**>  <**a class="btn btn-default btn-xs"  href="**{{ path(**'article\_view'**, {**'id'**: article.id}) }}**"**>Read more **&raquo;**</**a**>  </**div**>  </**footer**>  </**article**>  </**div**>  {% **endfor** %}  </**div**>  </**div**> {% **endblock** %} |

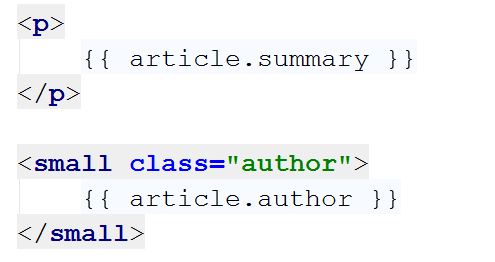
We will make a foreach loop for **all articles** with the given tag and **print them all**. Let's **fix the tag link** in our **single article view** now:



Let's see what we get when we **click** **on** **a** **tag**:



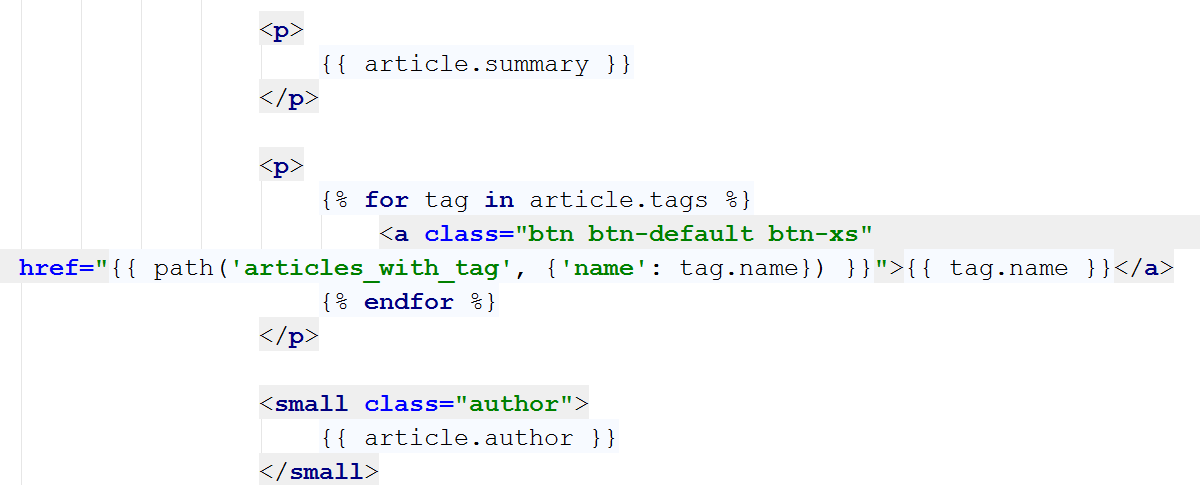
We see **all of the articles with the given tag**, with the **rest of their tags**. We can **implement that in the category** **listing** as well. To do that simply open the "article/list" view and find:



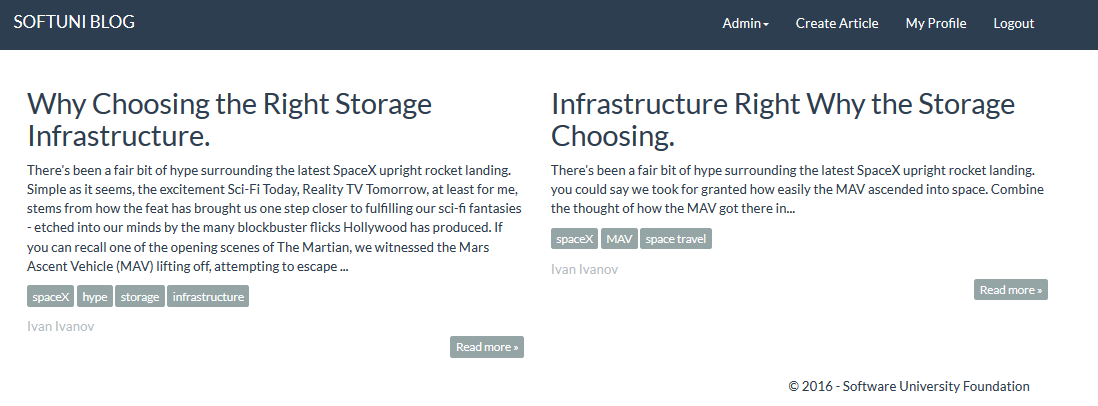
After the summary write the following code:

|  |
| --- |
| <**p**>  {% **for** tag **in** article.tags %}  <**a class="btn btn-default btn-xs" href="**{{ path(**'articles\_with\_tag'**, {**'name'**: tag.name}) }}**"**>{{ tag.name }}</**a**>  {% **endfor** %}  </**p**> |

The result should be this:



Let's see the listings in a random category:



This is everything. We want to **congratulate** **you** if you got to this point and **thank you** for your patience.

We hope that **you've** **enjoyed working** with **PHP, Symfony, Doctrine** and **MySQL**. Farewell, friends! ☺