Lab: Blog - PHP and Symfony

This document defines a complete walkthrough of creating a **Blog** application with the Symfony Framework, from setting up the framework through the authentication module, to creating a **CRUD** around Doctrine entities.

Make sure you have installed XAMPP, HeidiSQL and added PHP root folder to the path environment variable.

Chapters from I to III are for advanced users. There's a skeleton which you can use and start from chapter IV.

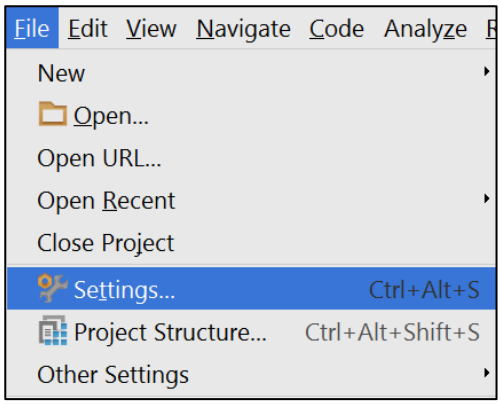
I. Set Up Symfony Project

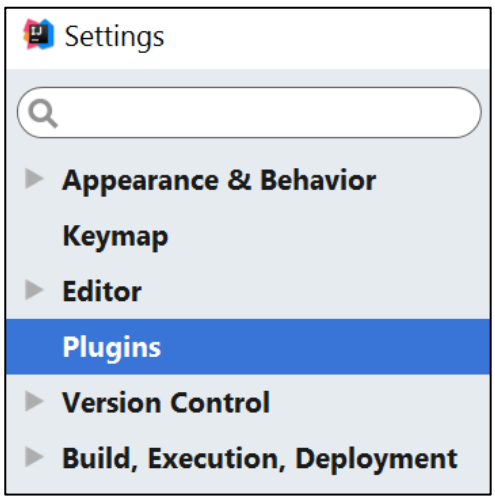
Symfony framework comes with various ways of creating a project, all of them involving the presence of Symfony project. The most convenient way is to **create a project via your IDE**. Luckily there are several **plugins** for **PHPStorm** (and the other **IDEA**-based IDE's) which help developing application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

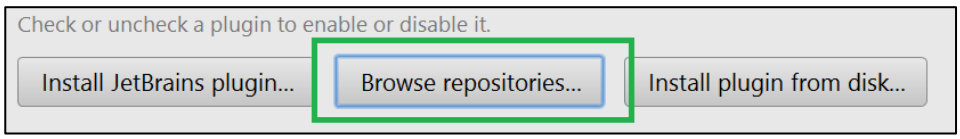
1. Install Symfony-related Plugins

Before we start working on our project, we can make our life easier by **installing** a couple of related **plugins**:

- Go to **[File] ➔ [Settings] ➔ [Plugins] ➔ [Browse repositories]**:



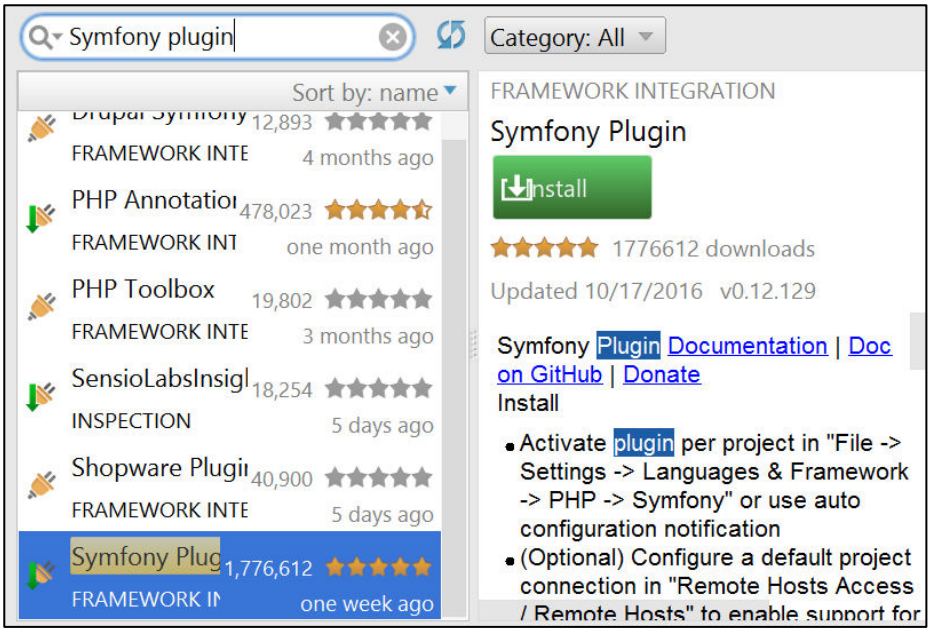




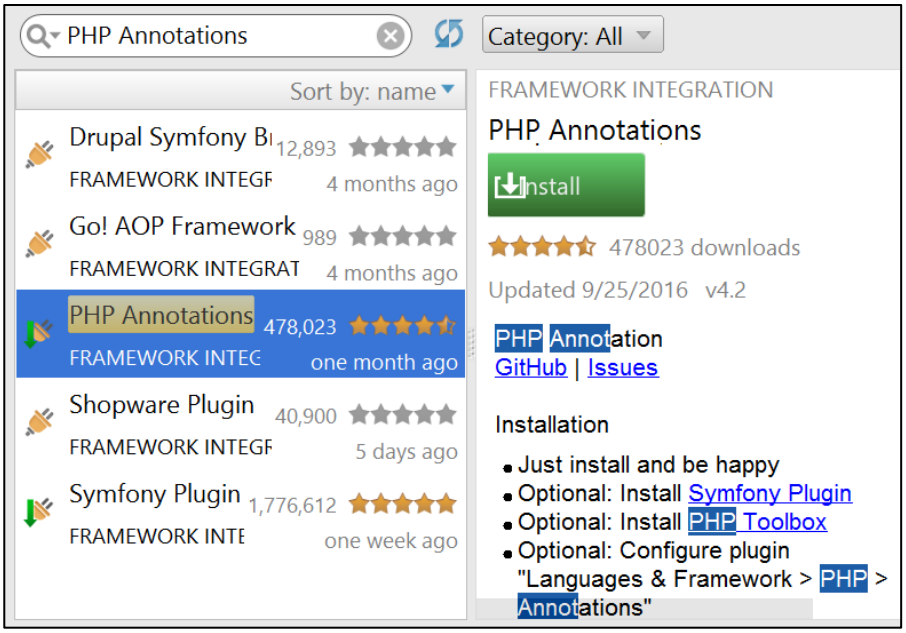
We need to install the following plugins:

© Software University Foundation (softuni.org). This work is licensed under the CC-BY-NC-SA license.
Follow us:

1. Symfony Plugin

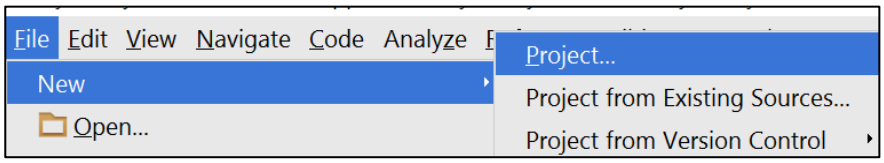


2. PHP Annotations

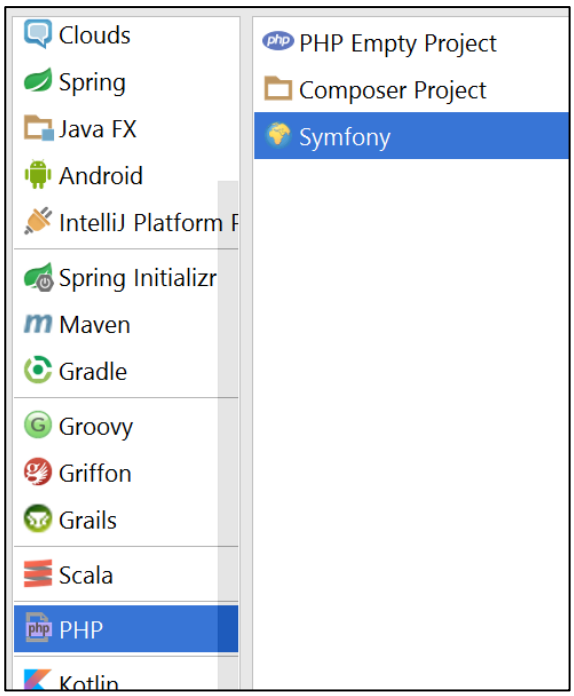


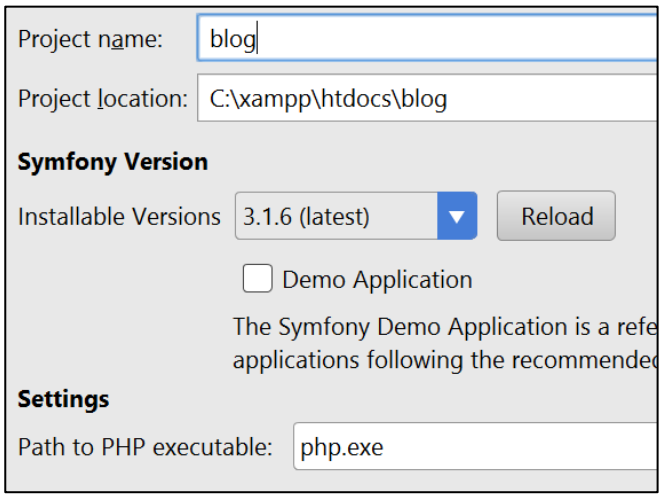
2. Create Symfony Project from IDE

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

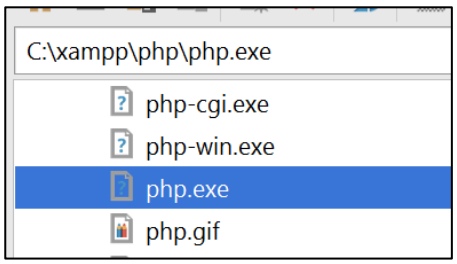


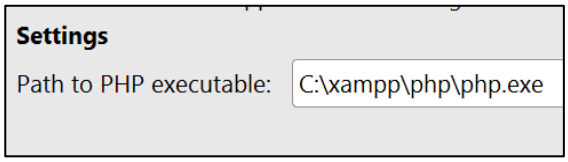
© Software University Foundation (softuni.org). This work is licensed under the CC-BY-NC-SA license.
Follow us:





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

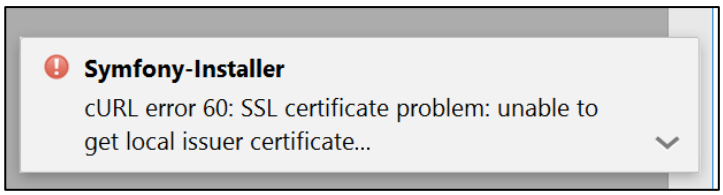




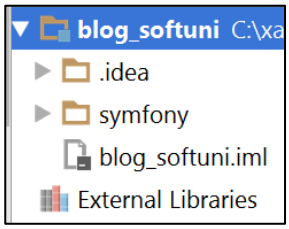
3. Check Project Status

If you have received the following error:

© Software University Foundation (softuni.org). This work is licensed under the CC-BY-NC-SA license.
Follow us:



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 these [instructions] **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 find the following line

```
[curl]
; A default value for
required to be an
; absolute path.
;curl.cainfo =

[openssl]
; The location of a Ce
```

3. And make it: "**`curl.cainfo = c:\xampp\php\cacert.pem`**"

```
[curl]
; A default value for the CURLOPT_CAINFO o
required to be an
; absolute path.
curl.cainfo = c:\xampp\php\cacert.pem

[openssl]
; The location of a Certificate Authority
```

4. Create the project again

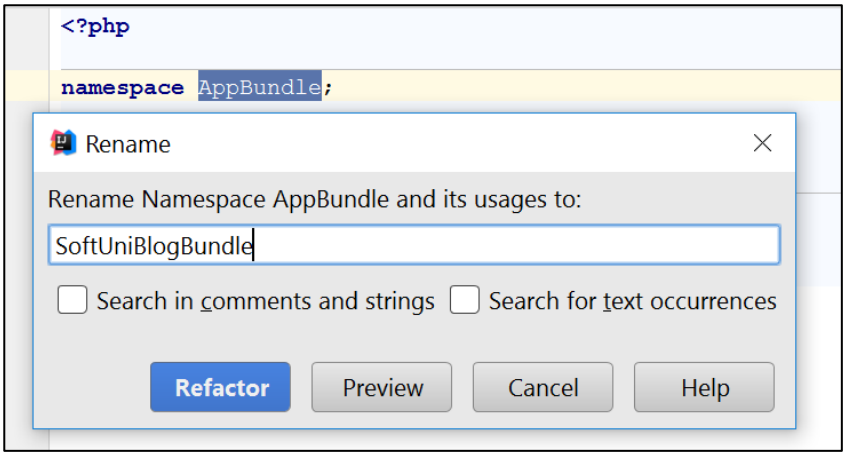
4. Rename Default Bundle

The Default bundle located in **`src`** folder is called **`AppBundle`**. Rename with the following occurrences to **`SoftUniBlogBundle`**, using **[Shift+F6]**:

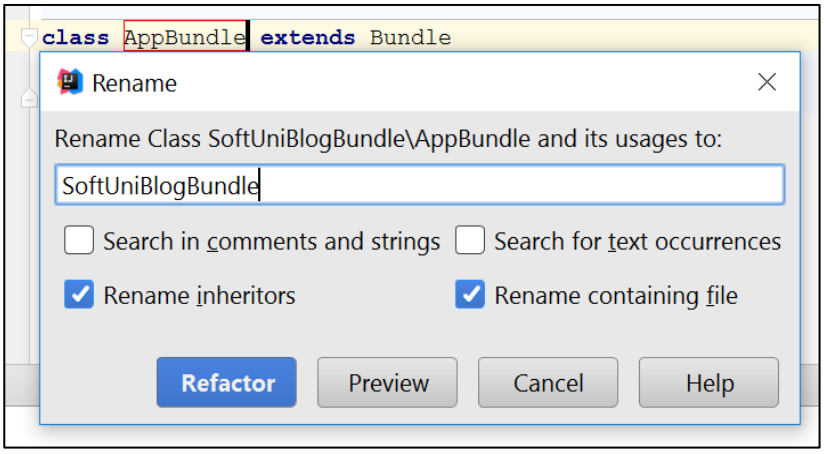
1. **`src/AppBundle`** folder
2. **`src/AppBundle/AppBundle.php`**

© Software University Foundation (softuni.org). This work is licensed under the CC-BY-NC-SA license.
Follow us:

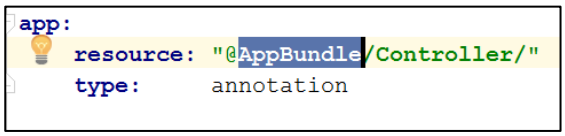
3. The namespace directive in **`src/AppBundle/AppBundle.php`**



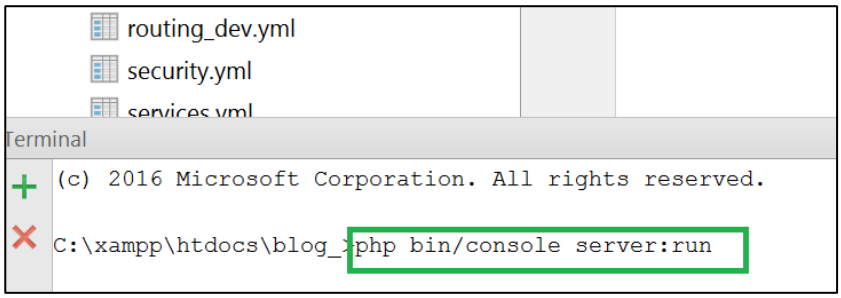
4. The classname in **`src/AppBundle/AppBundle.php`**



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



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

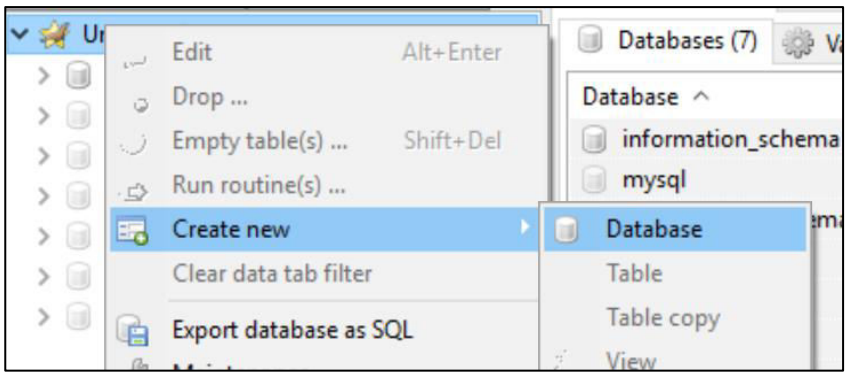


After that, you can see the result at http://localhost:8000 ☺

5. Create Database

Open HeidiSQL, connect to the MySQL instance and create a database named "**`blog`**"

© Software University Foundation (softuni.org). This work is licensed under the CC-BY-NC-SA license.
Follow us:



And change the database name in **`app/config/parameters.yml`** to "**`blog`**"

```
# This file is auto-generate
parameters:
    database_host: 127.0.0.1
    database_port: null
    database_name: blog
    database_user: root
```

*Note: you also need to specify your **MySQL** database **root user password**:*

```
parameters:
    database_host: 127.0.0.1
    database_port: 3306
    database_name: blog
    database_user: root
    database_password: null
```

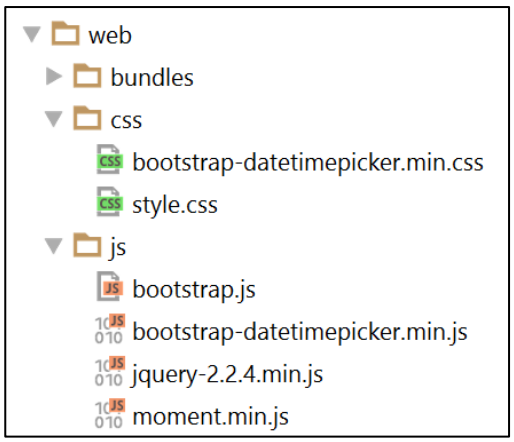
6. Setup Layout

We will need a base layout for all 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, where part of our **JavaScript** and **CSS** is included. In addition, we will need:

1. Bootstrap Date Time picker for choosing dates in our forms
2. Moment JS for validating dates

All our styles and scripts we need to include to our project. Create two folders in the "**`web`**" folder called "**`css`**" and "**`js`**" respectively. In the **blog design skeleton** in the folder scripts you can find the **`jquery`** and **`bootstrap`** files.

Place the needed scripts and styles there, ending up 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`**.

© Software University Foundation (softuni.org). This work is licensed under the CC-BY-NC-SA license.
Follow us:

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>
```

© Software University Foundation (softuni.org). This work is licensed under the CC-BY-NC-SA license.

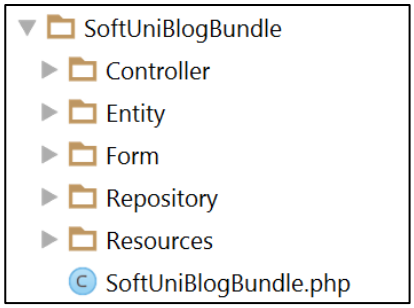
Follow us:

```
{% 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>
```

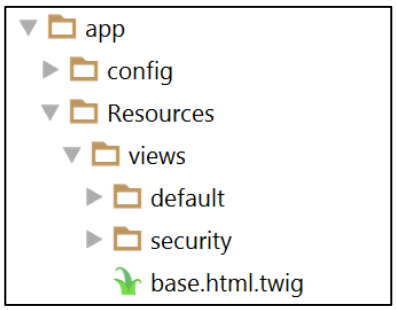
II. Symfony Base Project Overview

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

Initially the project comes with a main bundle, which can be treated as a plugin later. A **bundle** often has **Controllers**, **Entities** and related components (e.g. **Repositories**, **Forms**, **Commands**…)



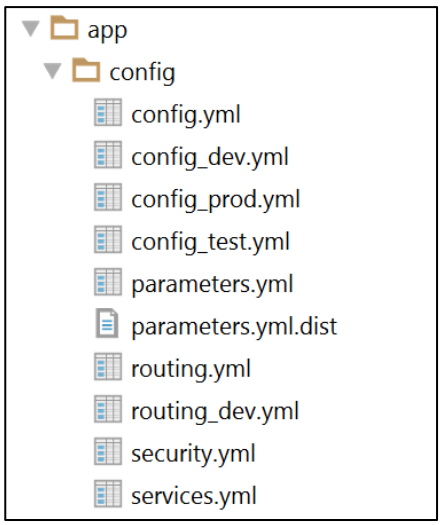
Standard templates (**views**) reside in the application folder (**app**) and are usually separated in a folder named after the **controller names**.



The de-facto standard **View Engine** in Symfony is Twig.

The base **configuration** of the project is placed in **`app/config`**, where configuration files for the Doctrine connection are defined among Security management, Routing rules, registering Services and so forth.

© Software University Foundation (softuni.org). This work is licensed under the CC-BY-NC-SA license.
Follow us:



It's very important that the **`parameters.yml.dist`** file contains the **same** keys as the ones in **`parameters.yml`**, since installing a new bundle will **delete unused pairs**.

III. 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 could be confusing. In the walkthrough below, we will setup a **registration and login process** and accessing **secured** content.

1. Creating User Entity

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

In the de-facto, standard **ORM** in Symfony, called **Doctrine**, these objects are called **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 for managing the database and creating entities. Let's use Doctrine to create an entity called **`User`**, using the **entity generation wizard**. To do this, we need to **open** a **terminal window** in the **project root directory** and type the following command:

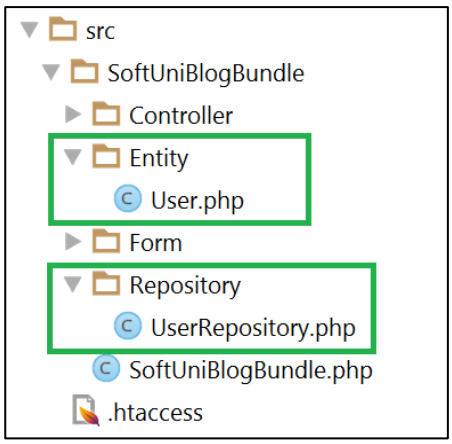
```
php bin/console doctrine:generate:entity
```

This will prompt us to enter an **entity name**. Entities are **prefixed** with the **bundle** they should belong to. Our bundle is called **`SoftUniBlogBundle`** (the default name is **`AppBundle`**), so we'll type in **`SoftUniBlogBundle:User`** (or **`AppBundle:User`**, if your 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). The **email should be unique**, so **when you are prompted for uniqueness there, type** "**true**" instead of just clicking enter (which defaults to **false**)

When the last field (**`fullName`**) is created and you are prompted for another one, just click **enter** to exit the wizard. This will create the **`User`** entity and its corresponding **`UserRepository`**.

© Software University Foundation (softuni.org). This work is licensed under the CC-BY-NC-SA license.
Follow us:



2. Setting Up Security Configuration

As we have said, Symfony comes with a couple of configuration files, one of which is called **`security.yml`**. We need to specify a few things, such as:

- How the password will be **encrypted** and on **which entity**
- **Which** entity will be used for **users** and which of its **fields** will be the **username field** (e.g. **email**, **username**, etc.)
- **Where the login form will be located** (route name)
- Where this **login form will post to**

Below is a **`security.yml`** file, which has the following configuration:

- The bundle is called **`SoftUniBlogBundle`**
- The **user** entity is called **`User`**, and its username field is called **`email`**
- The login form will be accessed and posted to "**`security_login`**"
- After a successful login, the user will be redirected 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
```

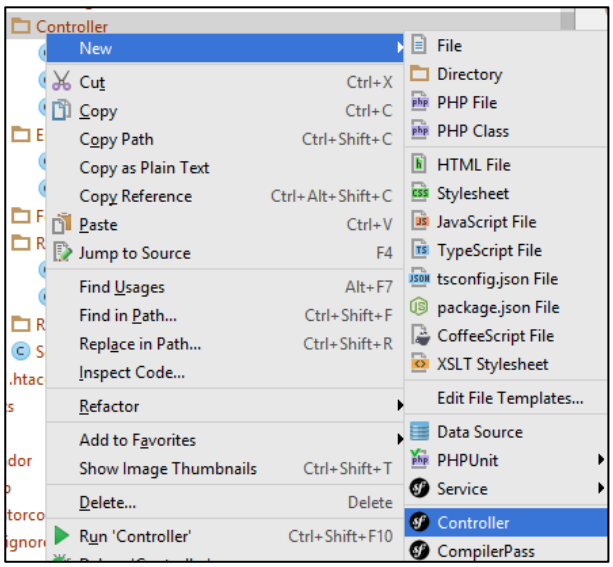
© Software University Foundation (softuni.org). This work is licensed under the CC-BY-NC-SA license.
Follow us:

```
            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 }
```

3. Login Form

To create a login form, we need to create a so-called **`Controller`** which will **listen on** this **route** (which above we called "**`security_login`**") and render the **`View`** with the login form when someone goes to the **`/login`** route.

Let's call our Controller "**`SecurityController`**":



Then we need a method (which we will call "**`login()`**"), which listens on the "**`/login`**" **route** and renders a view (let's point it to a **`login.html.twig`** file, which resides in the **`security`** **folder**)

```
namespace SoftUniBlogBundle\Controller;

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

class SecurityController extends Controller
{
    /**
     * @Route("/login", name="security_login")
     * @return \Symfony\Component\HttpFoundation\Response
     */
    public function login()
    {
        return $this->render("security/login.html.twig");
    }
}
```

The yellow background color in the view name tells us we don't have that view yet. We could easily create it by clicking **`[Alt+Enter]`** ☺

© Software University Foundation (softuni.org). This work is licensed under the CC-BY-NC-SA license.
Follow us:

```
return $this->render("security/login.html.twig");
```

Create template

Convert string literal to HEREDOC

Before messing with any 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 the **`security_login`** route. **Twig**, fortunately, provides a function **`path()`** that uses route names and generates URLs from them

```
<form name="authenticate" action="{{ path('security_login') }}" method="post">
```

The form is named "**authenticate**" because we will use this name later to generate a CSRF Token

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

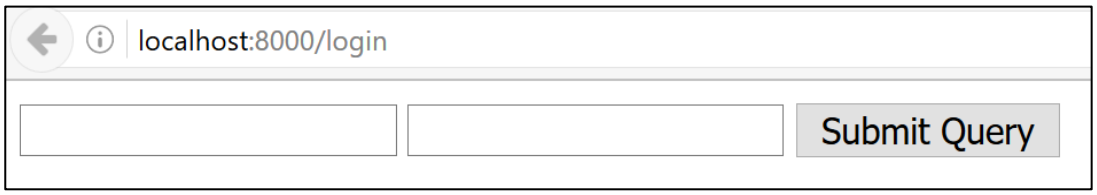
We need to define these two text fields (or **password** field for the password type ☺)

```
<input type="text" name="_username" >
<input type="text" name="_password" >
```

And a field for the CSRF Token using the Twig's helper method **`csrf_token()`** which accepts the form name.

```
    <input type="hidden" name="_csrf_token" value="{{ csrf_token('authenticate') }}"/>
    <input type="submit">
</form>
```

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



Not the most beautiful login form ☺ But still it's there! ☺

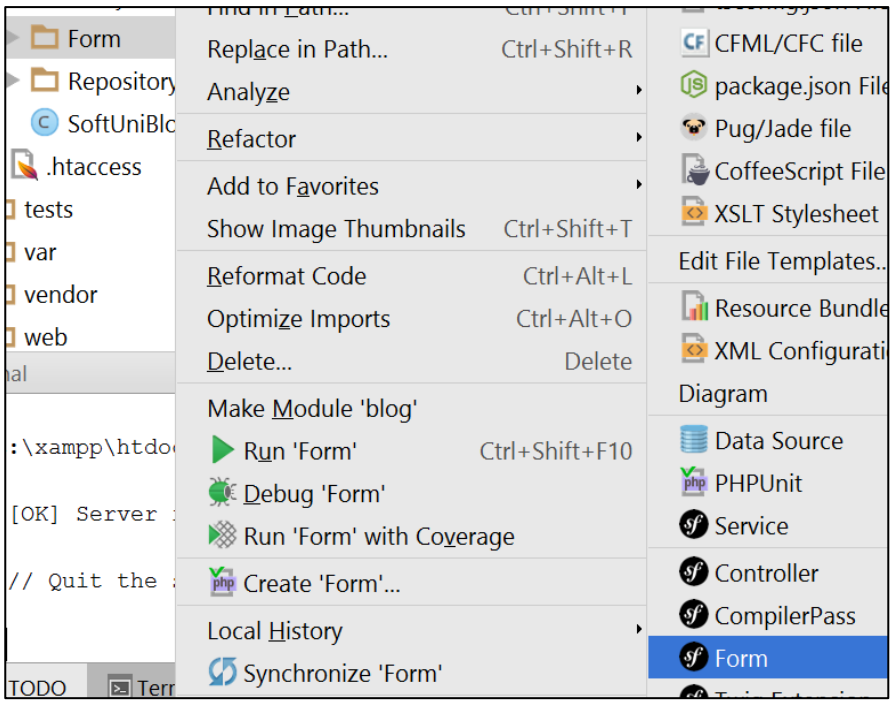
4. Register Form

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 for a form to work with an entity, it needs a corresponding FormType. 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:

© Software University Foundation (softuni.org). This work is licensed under the CC-BY-NC-SA license.
Follow us:



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 our three fields are text types, so we will use a **`TextType`** from the **`Symfony\Component\Form\Extension\Core\Type\TextType`** namespace.

```
public function buildForm(FormBuilderInterface $builder, array $options)
{
    $builder->add("email", TextType::class)
        ->add("password", TextType::class)
        ->add("fullName", TextType::class);
}
```

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

```
/**
 * @Route("register")
 * @param Request $request
 * @return \Symfony\Component\HttpFoundation\Response
 */
public function registerAction(Request $request)
{
    $user = new User();
    $form = $this->createForm(UserType::class, $user);
```

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

© Software University Foundation (softuni.org). This work is licensed under the CC-BY-NC-SA license.
Follow us:

```
if ($form->isSubmitted()) {
    $password = $this->get('security.password_encoder')
        ->encodePassword($user, $user->getPassword());
```

Expected \Symfony\Component\Security\Core\User\UserInterface, got \SoftUniBlogBundle\Entity\User more... (Ctrl+F1)

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.

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

/**
 * User
 *
 * @ORM\Table(name="user")
 * @ORM\Entity(repositoryClass="SoftUniBlogBundle\Repository\UserRepository")
 */
class User implements UserInterface
{
```

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:

```
    * Alternatively, the roles might be stored on a ``roles`` property,
    * and populated in any number of different ways when the user object
    * is created.
    *
    * @return (Role|string)[] The user roles
    */
   public function getRoles()
   {
       return [];
   }
```

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:

```
   /**
    * Returns the username used to authenticate the user.
    *
    * @return string The username
    */
   public function getUsername()
   {
       return $this->email;
   }
```

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 to the database

© Software University Foundation (softuni.org). This work is licensed under the CC-BY-NC-SA license.

Follow us:

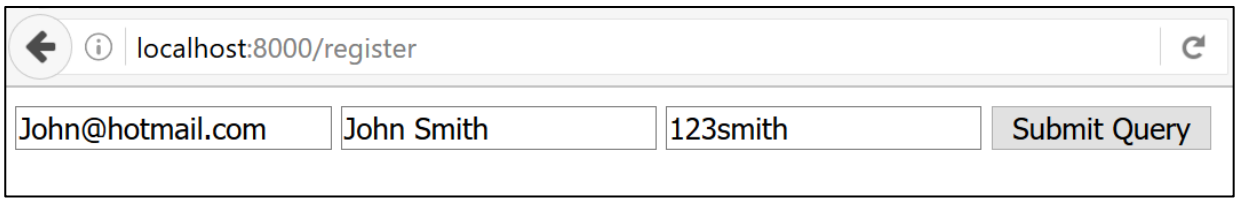
```
$form->handleRequest($request);
if ($form->isSubmitted()) {
    $password = $this->get('security.password_encoder')
        ->encodePassword($user, $user->getPassword());

    $user->setPassword($password);

    $em = $this->getDoctrine()->getManager();
    $em->persist($user);
    $em->flush();
    return $this->redirectToRoute("security_login");
}
return $this->render("default/register.html.twig");
```

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]:**

```
<form method="post" name="register">
    <input type="text" name="user[email]" >
    <input type="text" name="user[fullName]" >
    <input type="text" name="user[password]" >
    <input type="hidden" name="_csrf_token" value="{{ csrf_token('register') }}"/>
    <input type="submit">
</form>
```

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



IV. Creating Articles

0. 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, 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. We will use MySQL, 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:

© Software University Foundation (softuni.org). This work is licensed under the CC-BY-NC-SA license.
Follow us:

```
Diese Eingabeforderung nicht waehrend des Running beenden
Please dont close Window while MySQL is running
MySQL is trying to start
Please wait  ...
MySQL is starting with mysql\bin\my.ini (console)
2016-11-01  7:51:04 1556 [Note] mysql\bin\mysqld (mysqld 10.1.13-MariaDB) starting as process 1372 ...
2016-11-01  7:51:04 1556 [Note] InnoDB: Using mutexes to ref count buffer pool pages
2016-11-01  7:51:04 1556 [Note] InnoDB: The InnoDB memory heap is disabled
2016-11-01  7:51:04 1556 [Note] InnoDB: Mutexes and rw_locks use Windows interlocked functions
2016-11-01  7:51:04 1556 [Note] InnoDB: Memory barrier is not used
2016-11-01  7:51:04 1556 [Note] InnoDB: Compressed tables use zlib 1.2.3
2016-11-01  7:51:04 1556 [Note] InnoDB: Using generic crc32 instructions
2016-11-01  7:51:04 1556 [Note] InnoDB: Initializing buffer pool, size = 128.0M
2016-11-01  7:51:04 1556 [Note] InnoDB: Completed initialization of buffer pool
2016-11-01  7:51:04 1556 [Note] InnoDB: Highest supported file format is Barracuda.
2016-11-01  7:51:04 1556 [Note] InnoDB: The log sequence numbers 2607638 and 2607638 in ibdata files do no
g sequence number 2624064 in the ib_logfiles!
2016-11-01  7:51:04 1556 [Note] InnoDB: Database was not shutdown normally!
2016-11-01  7:51:04 1556 [Note] InnoDB: Starting crash recovery.
2016-11-01  7:51:04 1556 [Note] InnoDB: Reading tablespace information from the .ibd files...
2016-11-01  7:51:04 1556 [Note] InnoDB: Restoring possible half-written data pages
2016-11-01  7:51:04 1556 [Note] InnoDB: from the doublewrite buffer...
2016-11-01  7:51:05 1556 [Note] InnoDB: 128 rollback segment(s) are active.
2016-11-01  7:51:05 1556 [Note] InnoDB: Waiting for purge to start
2016-11-01  7:51:05 1556 [Note] InnoDB:  Percona XtraDB (http://www.percona.com) 5.6.28-76.1 started; log
r 2624064
2016-11-01  7:51:05 13432 [Note] InnoDB: Dumping buffer pool(s) not yet started
2016-11-01  7:51:05 1556 [Note] Plugin 'FEEDBACK' is disabled.
2016-11-01  7:51:05 1556 [Note] Server socket created on IP: '::'.
2016-11-01  7:51:05 1556 [Note] mysql\bin\mysqld: ready for connections.
Version: '10.1.13-MariaDB'  socket: ''  port: 3306  mariadb.org binary distribution
```

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.

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

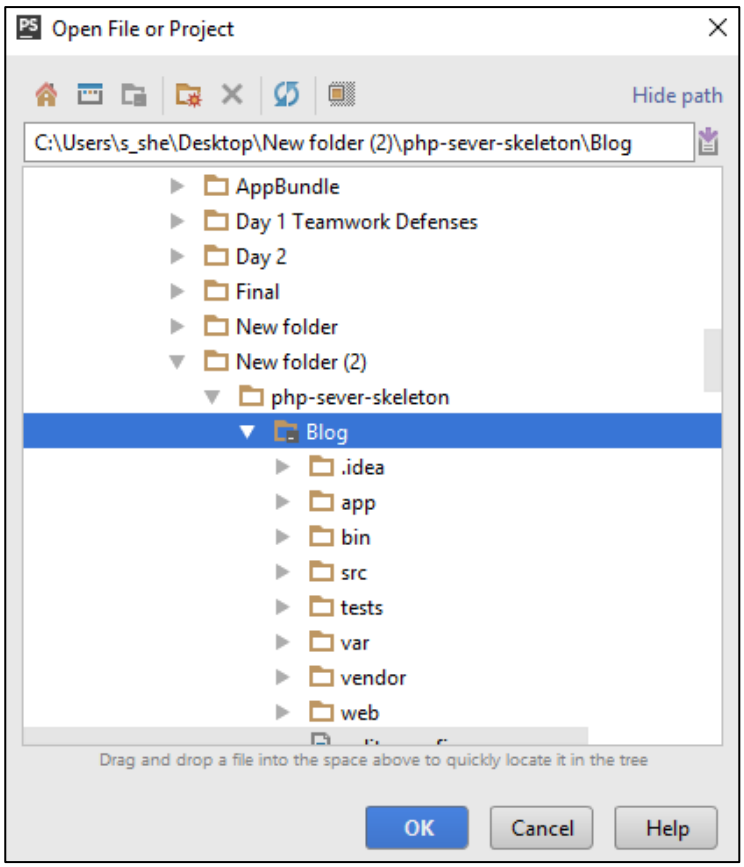
Skip this step if you have gone through the above III chapters.

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**):

© Software University Foundation (softuni.org). This work is licensed under the CC-BY-NC-SA license.
Follow us:



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.

2. 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. 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:

```
 Welcome to the Doctrine2 entity generator



This command helps you generate Doctrine2 entities.

First, you need to give the entity name you want to generate.
You must use the shortcut notation like AcmeBlogBundle:Post.
```

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

```
SoftUniBlogBundle:Article
```

The result should be the following:

© Software University Foundation (softuni.org). This work is licensed under the CC-BY-NC-SA license.
Follow us:

```
Determine the format to use for the mapping information.

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

Just press **[Enter]**. Now we need to **define the properties** for our entity. you should see this:

```
Instead of starting with a blank entity, you can add some fields now.
Note that the primary key will be added automatically (named id).

Available types: array, simple_array, json_array, object,
boolean, integer, smallint, bigint, string, text, datetime, datetimetz,
date, time, decimal, float, binary, blob, guid.

New field name (press <return> to stop adding fields):
```

Our **first field** will be the "**title**" of our article. Just write "**title**" and press **[Enter]**. You should see this:

```
New field name (press <return> to stop adding fields): title
Field type [string]:
```

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:

```
New field name (press <return> to stop adding fields): title
Field type [string]:
Field length [255]:
Is nullable [false]:
Unique [false]:

New field name (press <return> to stop adding fields):
```

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

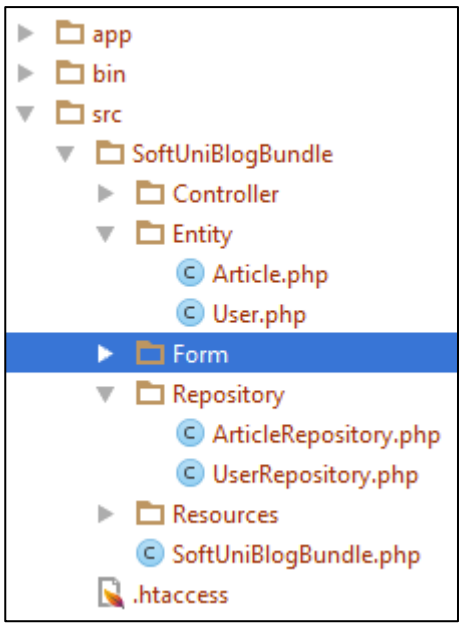
```
New field name (press <return> to stop adding fields): content
Field type [string]: text
Is nullable [false]:
Unique [false]:

New field name (press <return> to stop adding fields): dateAdded
Field type [string]: datetime
Is nullable [false]:
Unique [false]:
```

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

```
 Entity generation


> Generating entity class C:\Users\s_she\Desktop\New folder\BlogSkeleton\src\SoftUniBlogBundle\Entity\Article.php: OK!
> Generating repository class C:\Users\s_she\Desktop\New folder\BlogSkeleton\src\SoftUniBlogBundle\Repository\ArticleRepository.php: OK!


 Everything is OK! Now get to work :).
```

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

© Software University Foundation (softuni.org). This work is licensed under the CC-BY-NC-SA license.
Follow us:



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.

3. 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 for our table. On top of our entity we should see something that looks like a comment:

```
/**
 * Article
 *
 * @ORM\Table(name="article")
 * @ORM\Entity(repositoryClass="SoftUniBlogBundle\Repository\ArticleRepository")
 */
class Article
{
```

However, this is not just a comment. It is an **`annotation`**. More specifically, it is a Doctrine Annotation. 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:

```
    /**
     * @var \DateTime
     *
     * @ORM\Column(name="dateAdded", type="datetime")
     */
    private $dateAdded;
```

© Software University Foundation (softuni.org). This work is licensed under the CC-BY-NC-SA license.
Follow us:

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

```
/**
 * @var string
 */
private $summary;
```

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

```
/**
 * @param string
 */
private function setSummary()
{
    $this->summary = substr($this->getContent(), 0, strlen($this->getContent()) / 2) . "...";
}
```

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**:

```
/**
 * @return string
 */
public function getSummary()
{
    if($this->summary === null)
    {
        $this->setSummary();
    }
    return $this->summary;
}
```

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

4. 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:

```
/**
 * @var int
 *
 * @ORM\Column(name="authorId", type="integer")
 */
private $authorId;
```

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:

© Software University Foundation (softuni.org). This work is licensed under the CC-BY-NC-SA license.
Follow us:

```
/**
 * @param integer $authorId
 *
 * @return Article
 */
public function setAuthorId($authorId)
{
    $this->authorId = $authorId;

    return $this;
}
```

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

```
/**
 * @return int
 */
public function getAuthorId()
{
    return $this->authorId;
}
```

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**":

```
/**
 * @var User
 *
 * @ORM\ManyToOne(targetEntity="SoftUniBlogBundle\Entity\User", inversedBy="articles")
 * @ORM\JoinColumn(name="authorId", referencedColumnName="id")
 */
private $author;
```

More new stuff! We're using 2 new annotations. The first one is the "**`ManyToOne`**" annotation. A **many to one** relationship represents a One To Many relationship from the side of the "**many**". In our case, we will use a "**one to many relationship**" to tell the program that **one user** will have **many articles**. 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 the **`User`** entity 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 "`authorId`" in our article entity will correspond to the "id" field from the `User`** entity.

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

```
/**
 * @param \SoftUniBlogBundle\Entity\User $author
 *
 * @return Article
 */
public function setAuthor(User $author = null)
{
    $this->author = $author;

    return $this;
}
```

© Software University Foundation (softuni.org). This work is licensed under the CC-BY-NC-SA license.
Follow us:

And our **getter**:

```
/**
 * @return \SoftUniBlogBundle\Entity\User
 */
public function getAuthor()
{
    return $this->author;
}
```

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:

```
/**
 * @var ArrayCollection
 *
 * @ORM\OneToMany(targetEntity="SoftUniBlogBundle\Entity\Article", mappedBy="author")
 */
private $articles;
```

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:

```
/**
 * @return \Doctrine\Common\Collections\Collection
 */
public function getArticles()
{
    return $this->articles;
}
```

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:

```
/**
 * @param \SoftUniBlogBundle\Entity\Article $article
 *
 * @return User
 */
public function addPost(Article $article)
{
    $this->articles[] = $article;

    return $this;
}
```

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

5. Set Default Values for our Entities

Our next job is to create the so-called constructors 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:

© Software University Foundation (softuni.org). This work is licensed under the CC-BY-NC-SA license.
Follow us:

```
public function __construct()
{
    $this->articles = new ArrayCollection();
}
```

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

```
public function __construct()
{
    $this->dateAdded = new \DateTime('now');
}
```

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).

6. 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. **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:

```
ATTENTION: This operation should not be executed in a production environment.
           Use the incremental update to detect changes during development and use
           the SQL DDL provided to manually update your database in production.

The Schema-Tool would execute "2" queries to update the database.
Please run the operation by passing one - or both - of the following options:
    doctrine:schema:update --force to execute the command
    doctrine:schema:update --dump-sql to dump the SQL statements to the screen
```

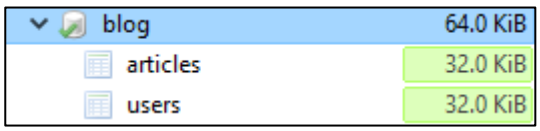
It basically tells us, that we are doing an operation that is not safe. To do it, **we need to force Doctrine** to execute our command. In order to do that we need to add "**--force**" after our previous command:

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

The result of this command should be the following:

```
Updating database schema...
Database schema updated successfully! "2" queries were executed
```

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

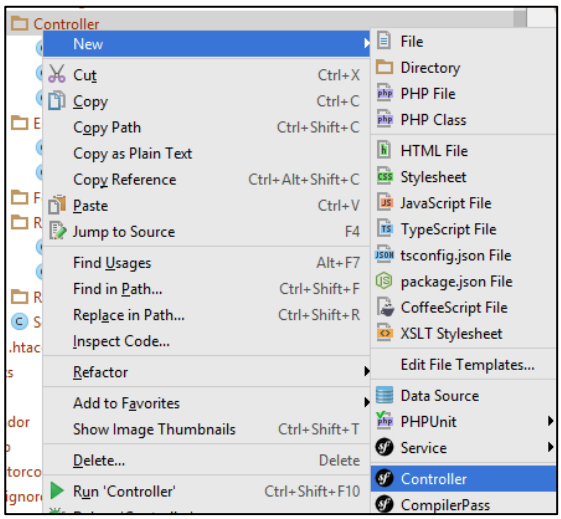


We are ready, to start making our blog.

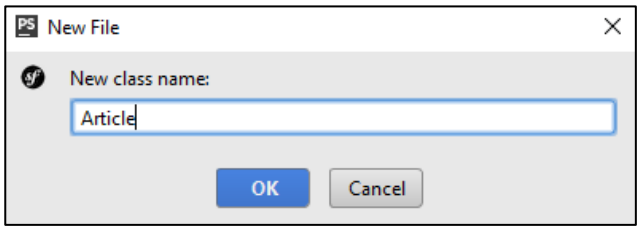
© Software University Foundation (softuni.org). This work is licensed under the CC-BY-NC-SA license.
Follow us:

7. 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 **`ArticleController`**:



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

```php
<?php

namespace SoftUniBlogBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class ArticleController extends Controller
{
    public function indexAction($name)
    {
        return $this->render('', array('name' => $name));
    }
}
```

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

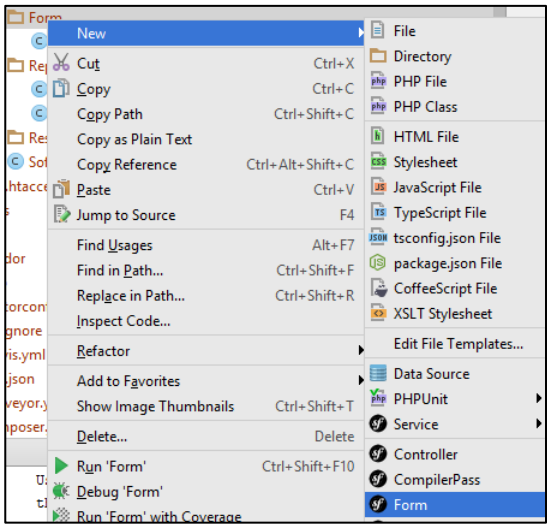
© Software University Foundation (softuni.org). This work is licensed under the CC-BY-NC-SA license.
Follow us:

```php
<?php

namespace SoftUniBlogBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class ArticleController extends Controller
{

}
```

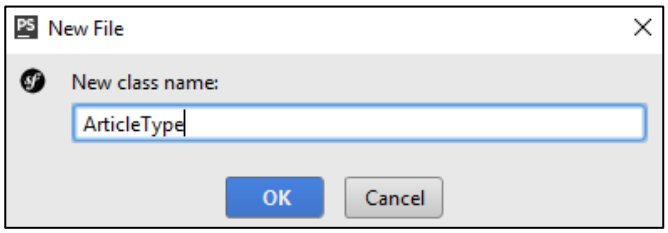
We have a controller, but we need **form template**.

8. 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:

© Software University Foundation (softuni.org). This work is licensed under the CC-BY-NC-SA license.
Follow us:

```php
<?php

namespace SoftUniBlogBundle\Form;

use ...

class ArticleType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {

    }

    public function configureOptions(OptionsResolver $resolver)
    {

    }

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

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:

```php
public function buildForm(FormBuilderInterface $builder, array $options)
{
    $builder
        ->add('title', TextType::class)
        ->add('content', TextType::class);
}
```

It's a 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:

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

Let's create the logic for our "**configureOptions**" function:

```php
public function configureOptions(OptionsResolver $resolver)
{
    $resolver->setDefaults(array(
        'data_class' => 'SoftUniBlogBundle\Entity\Article',
    ));
}
```

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.

9. 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:

© Software University Foundation (softuni.org). This work is licensed under the CC-BY-NC-SA license.

Follow us:

```
/**
 * @param Request $request
 *
 * @Route("/article/create", name="article_create")
 * @Security("is_granted('IS_AUTHENTICATED_FULLY')")
 *
 * @return \Symfony\Component\HttpFoundation\RedirectResponse
 *
 */
public function create(Request $request)
{

}
```

Let's start from the first annotation. It tells our project that the function will receive **one parameter** of type Request. We will save what request is for some other time. The second annotation is more interesting. It defines a "Route". 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:

```
$article = new Article();
$form = $this->createForm(ArticleType::class, $article);

return $this->render('article/create.html.twig',
    array('form' => $form->createView()));
```

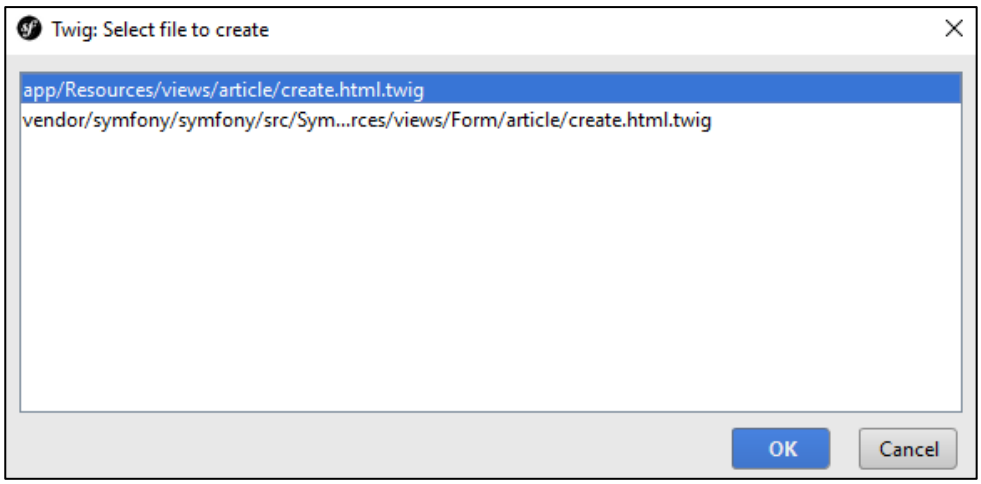
What is this code doing? It's simple – it **creates a new article**. Then it **creates** a **new form** from the template we've created earlier and tells the **form** that it **should fill our new article**. Finally, it **sends the form to a view** that we are going to **render** on the screen. Render means draw. Symfony uses Twig. Twig is a templating engine, 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:

```
>render('article/create.html.twig',
rm' => $form->createVie
```

Create template

Then you need to choose the first option:

© Software University Foundation (softuni.org). This work is licensed under the CC-BY-NC-SA license.
Follow us:



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

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

{% block main %}

{% endblock %}
```

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>
```

© Software University Foundation (softuni.org). This work is licensed under the CC-BY-NC-SA license.
Follow us:

```
</div>
```

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

The first part we are going to discuss is:

```
<form class="form-horizontal" action="{{ path('article_create') }}" method="POST">
```

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 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 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:

```
<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>
```

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:

```
{{ form_row(form._token) }}
```

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.

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

```
<a class="btn btn-default" href="{{ path('blog_index') }}">Cancel</a>
```

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
```

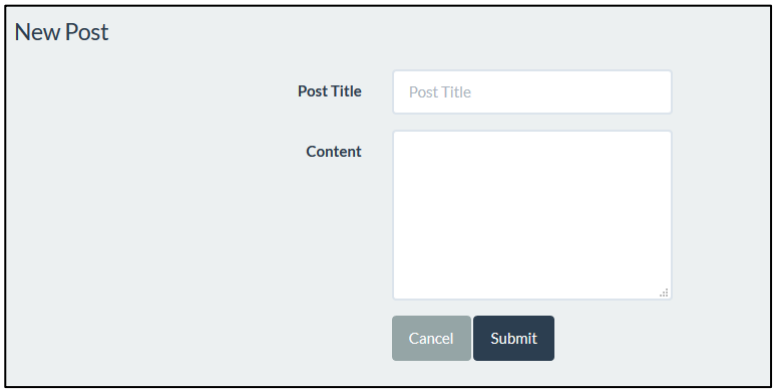
If everything works, you should see this:

```
[OK] Server running on http://127.0.0.1:8000

// Quit the server with CONTROL-C.
```

© Software University Foundation (softuni.org). This work is licensed under the CC-BY-NC-SA license.

Follow us:

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:

```
public function create(Request $request)
{
    $article = new Article();
    $form = $this->createForm(ArticleType::class, $article);

    $form->handleRequest($request);

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

        $article->setAuthor($this->getUser());
        $em = $this->getDoctrine()->getManager();
        $em->persist($article);
        $em->flush();

        return $this->redirectToRoute('blog_index');
    }

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

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:

© Software University Foundation (softuni.org). This work is licensed under the CC-BY-NC-SA license.
Follow us:

Find this part of the code:

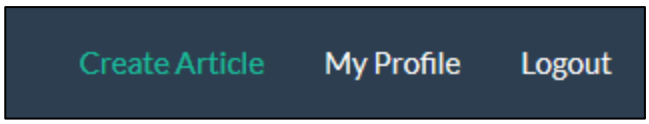
```
{% if app.user %}
    <li>
        <a href="{{ path('user_profile') }}">
            My Profile
        </a>
    </li>
    <li>
        <a href="{{ path('security_logout') }}">
            Logout
        </a>
    </li>
{% else %}
```

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

```
{% if app.user %}
    <li>
        <a href="{{ path('article_create') }}">
            Create Article
        </a>
    </li>
    <li>
        <a href="{{ path('user_profile') }}">
            My Profile
        </a>
    </li>
    <li>
        <a href="{{ path('security_logout') }}">
            Logout
        </a>
    </li>
{% else %}
```

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

© Software University Foundation (softuni.org). This work is licensed under the CC-BY-NC-SA license.
Follow us:



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. In **HeidiSQL**, open the **articles** table:

id	title	content	dateAdded	authorId
1	Creating article...	Random content...	2016-11-01 09:42:58	1

Hooray, we did it! Now we can create articles. The problem is that we can't see them on our blog. Let's implement that.

V. Listing Articles

1. 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**:

```
$articles = $this->getDoctrine()->getRepository(Article::class)->findAll();
```

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

```
return $this->render('blog/index.html.twig', ['articles' => $articles]);
```

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

You should see this:

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

{% block main %}

{% endblock %}
```

In the main block, write the following code:

© Software University Foundation (softuni.org). This work is licensed under the CC-BY-NC-SA license.
Follow us:

```
<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:

```
{% for article in articles %}
```

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:

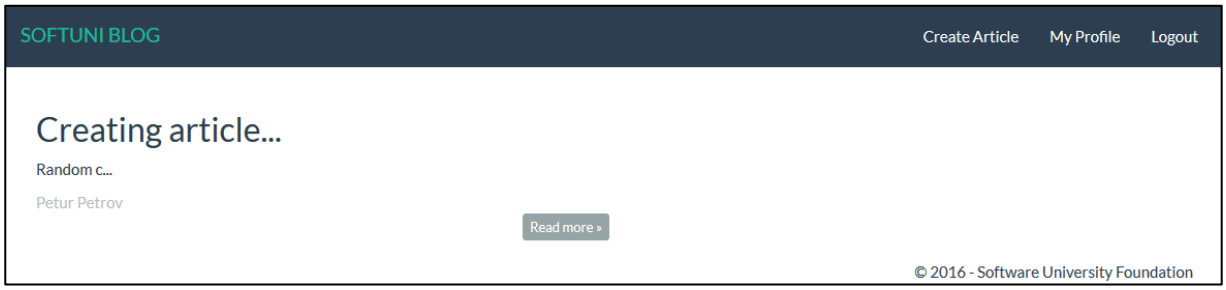
```
{% endfor %}
```

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

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

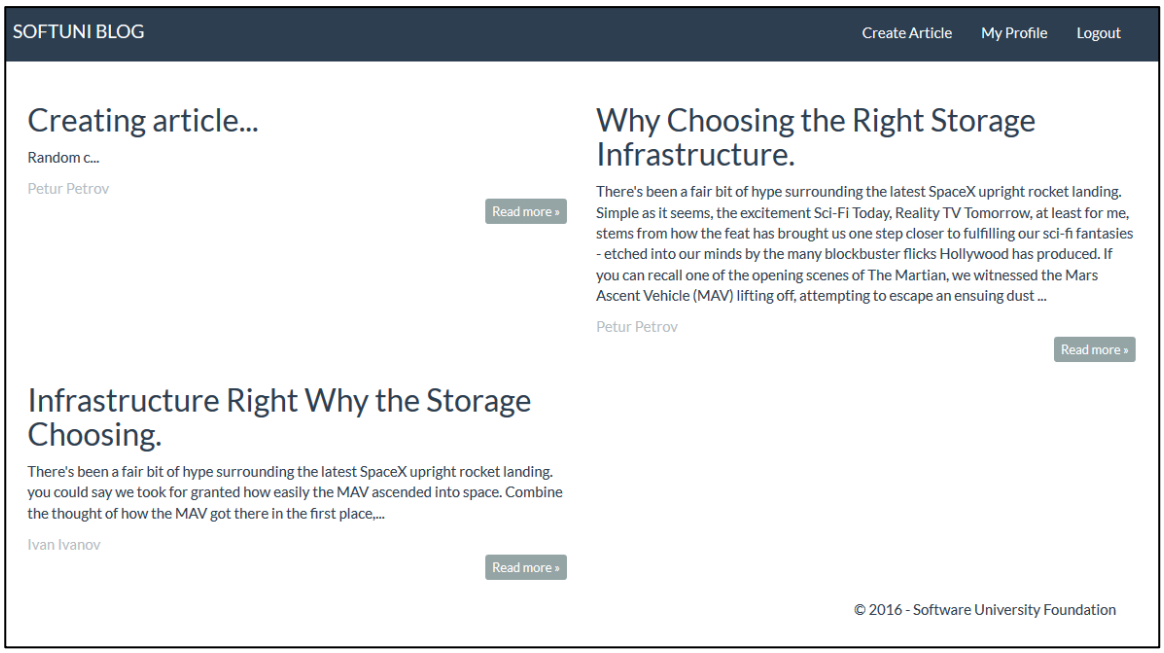
This will print the title for each article. We have the same thing for the summary and author of the article.

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



It works! Let's create few more articles:

© Software University Foundation (softuni.org). This work is licensed under the CC-BY-NC-SA license.
Follow us:



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

2. Showing Single Article

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

```
/**
 * @Route("/article/{id}", name="article_view")
 * @param $id
 * @return \Symfony\Component\HttpFoundation\Response
 */
public function viewArticle($id)
{
    $article = $this->getDoctrine()->getRepository(Article::class)->find($id);

    return $this->render('article/article.html.twig', ['article' => $article]);
}
```

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:

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

{% block main %}

{% endblock %}
```

Write the following code in the main block:

```
<div class="container body-content">
    <div class="row">
        <div class="col-md-12">
            <article>
```

© Software University Foundation (softuni.org). This work is licensed under the CC-BY-NC-SA license.
Follow us:

```
                <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.

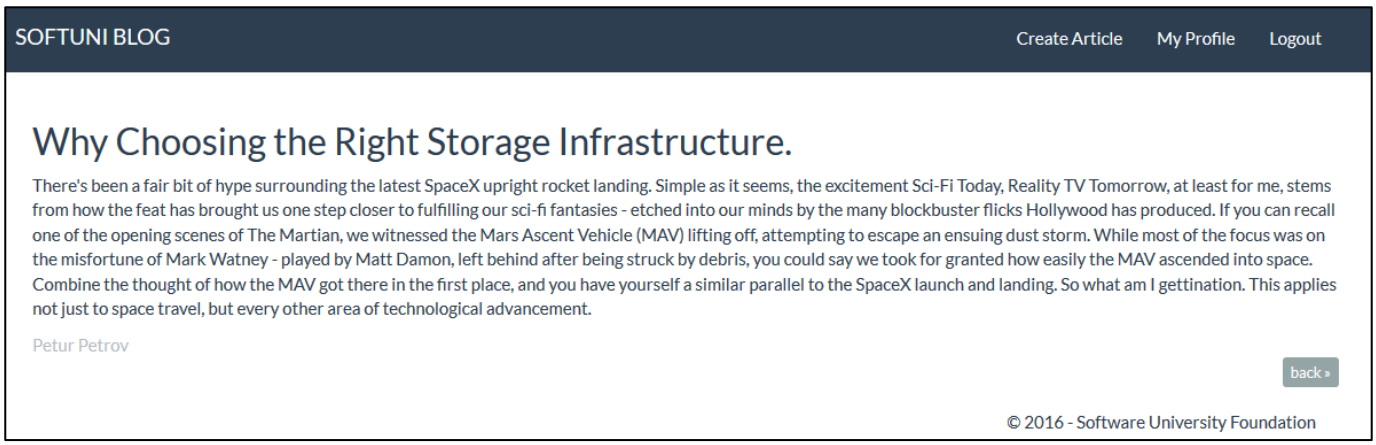
Let's start the blog and see if it works. The answer is no, it doesn't. Right now, the button read more doesn't redirect to the right route. Let's go back to the index view and find this piece of code:

```
<a class="btn btn-default btn-xs" href="#">Read more &raquo;</a>
```

Change it to:

```
<a class="btn btn-default btn-xs"
   href="{{ path('article_view', {'id': article.id}) }}">Read more &raquo;</a>
```

Let's try it now:



Congratulations! If everything works okay, you've just created a very simple blog system where users can log in and post articles.

© Software University Foundation (softuni.org). This work is licensed under the CC-BY-NC-SA license.
Follow us: