# Lab: 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 the [authentication](http://symfony.com/doc/current/security.html) module, to 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**](http://softuni.bg/downloads/svn/soft-tech/Jan-2018/Software-Technologies-Mar-2018/10.%20PHP-Blog-Basic-Functionality/10.%20PHP-Blog-Basic-Functionality-Blog-Skeleton.zip) **which you can use and start from chapter IV.**

Този документ дефинира пълен пътеводител за създаване на приложение за Blog с Symfony Framework, от създаване на рамката чрез модула за удостоверяване до създаването на CRUD около Entity of Doctrine.  
Уверете се, че сте инсталирали XAMPP, HeidiSQL и сте добавили основната PHP папка към променливата за пътната среда.  
Главите от I до III са за напреднали потребители. Има скелет, който можете да използвате и да започнете от глава IV.

# I. Set Up Symfony Project

I. Изграждане на симфоничен проект

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

Симфоничната рамка идва с различни начини за създаване на проект, всички включващи присъствието на проект Symfony. Най-удобният начин е да създадете проект чрез своя IDE. За щастие има няколко приставки за PHPStorm (и други IDEA-базирани IDE's), които помагат при разработването на приложенията със Symfony

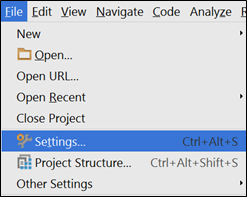
## 1. Install Symfony-related Plugins

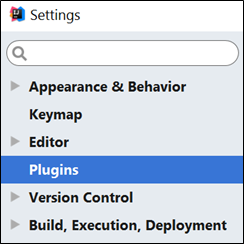
1. Инсталирайте Plugins, свързани с Symfony

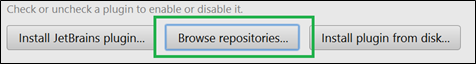
Before we start working on our project, we can make our life easier by **installing** a couple of related **plugins**:

* Go to **[File]** è **[Settings]** è **[Plugins] è [Browse repositories]**:

Преди да започнем да работим по нашия проект, можем да улесним живота си, като инсталираме няколко свързани плъгини:  
• Отидете на [Файл] è [Настройки] è [Plugins] è [Преглед на хранилища]:



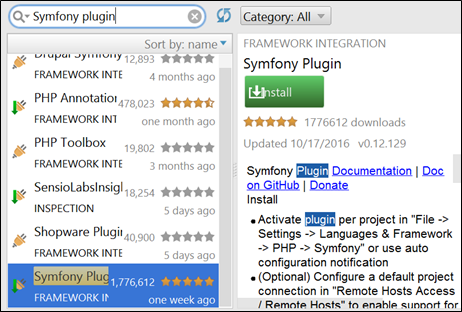


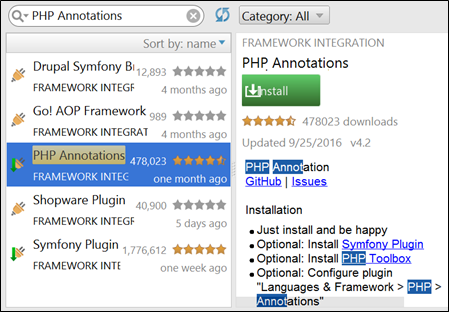


We need to install the following plugins:

1. Symfony Plugin

Трябва да инсталираме следните приставки:  
1. Симфони Plug-in



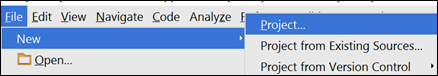
1. 2. PHP Annotations
2. 

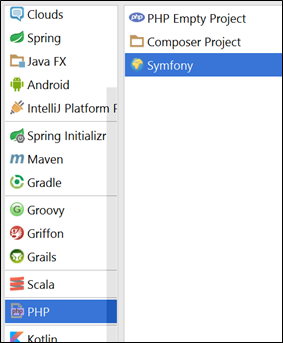
## 2. Create Symfony Project from IDE

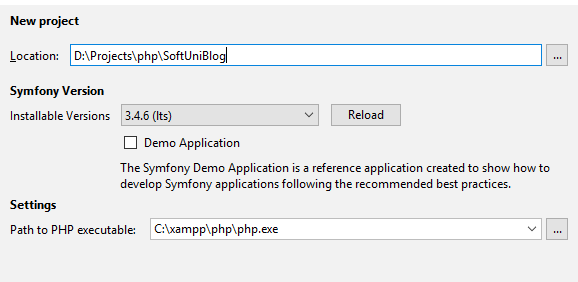
2. Създайте проект Symfony от IDE

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:

След като инсталирате плъгините и рестартирате IDE, ще имате PHP подкатегория (IntelliJ) или директно Symfony one (PHPStorm) в контекстното меню Create Project:





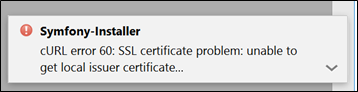


We need to specify the **php executable**, which most probably resides in **c:/xampp/php**

## 3. Check Project Status

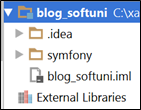
If you have received the following error:

Трябва да определим PHP изпълним файл, който най-вероятно се намира в c: / xampp / php  
3. Проверете състоянието на проекта  
Ако сте получили следната грешка:



And your project looks like this:

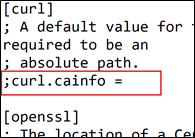
И вашият проект изглежда така:



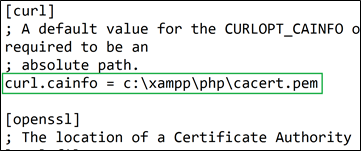
You most probably haven’t created the project properly. This could of possible missing curl.cainfo directive in **php.ini**.

Follow these [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 find the following line

Най-вероятно не сте създали правилно проекта. Това би могло да е от възможна липса на директива curl.cainfo в php.ini.  
Следвайте тези инструкции, САМО ако сте получили грешката по-горе, а другият ще се опитате да прекратите тази стъпка.  
1. Запазете този файл: https://curl.haxx.se/ca/cacert.pem в c: / xampp / php  
2. Редактирайте файла c: /xampp/php/php.ini и намерете следния ред

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



1. Create the project again

4. Създайте отново проекта

## 4. Rename Default Bundle

4. Преименувайте пакета по подразбиране

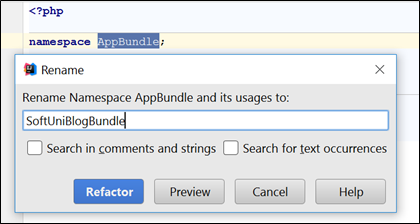
The Default bundle located in src folder is called AppBundle. Rename with the following occurrences to SoftUniBlogBundle, using **[Shift+F6]**:

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

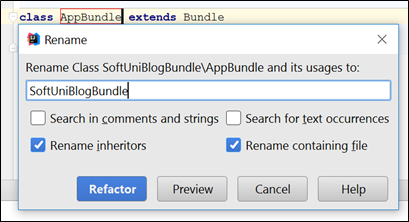
Пакетът по подразбиране, намиращ се в папката src, се нарича AppBundle. Преименувайте следните събития на SoftUniBlogBundle, като използвате [Shift + F6]:  
1. папка src / AppBundle  
2. SRC / AppBundle / AppBundle.php

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

3. Директивата за пространството на имената в src / AppBundle / AppBundle.php



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



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

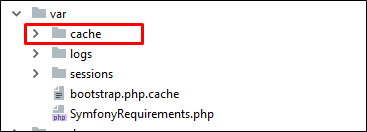
Променете случая в app / config / routing.yml и в SoftUniBlogBundle:



Change the occurrence in **composer.json** to SoftUniBlogBundle too.

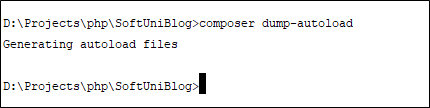
Delete the cache in var folder:

Променете случая в composer.json и в SoftUniBlogBundle.  
Изтриване на кеша в папката var:



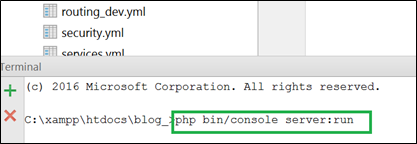
Run the following format in the console:

composer dump-autoload



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

Стартирайте сървъра, като изпълнявате следната команда в папката на проекта

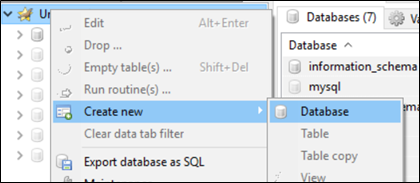


After that, you can see the result at [http://localhost:8000](http://localhost:8000/) J

## 5. Create Database

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

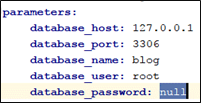
След това можете да видите резултата в http: // localhost: 8000   
5. Създаване на база данни  
Отворете HeidiSQL, свържете се с MySQL потребителския модел и създайте база данни с име "blog"



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



Забележка: Вие също трябва да зададете вашата MySQL база данни корен потребител потребител:



## 6. Setup Layout

6. Настройка на оформлението

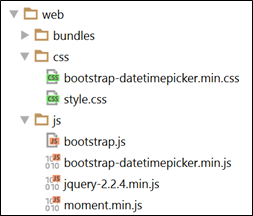
We will need a base layout for all our templates. As we are using **Bootstrap**, we will need its css included in all pages, and the related scripts too. We can download the sample **blog design skeleton** from [here](http://softuni.bg/downloads/svn/soft-tech/Jan-2018/Software-Technologies-Mar-2018/10.%20PHP-Blog-Basic-Functionality/10.%20PHP-Blog-Basic-Functionality-Blog-Design.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 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:

Ще се нуждаем от основно оформление за всички наши шаблони. Тъй като използваме Bootstrap, ще се нуждаем от CSS, включен във всички страници, и свързаните с тях скриптове. Можем да изтеглим примерния скелет за дизайн на блог от тук, където част от нашия JavaScript и CSS е включена. Освен това ще се нуждаем от:  
1. Bootstrap Date Time picker за избор на дати в нашите формуляри  
2. Moment JS за потвърждаване на датите  
Всичките ни стилове и скриптове трябва да включим в нашия проект. Създайте две папки в папката "web", наречена "css" и "js" съответно. В скелета за дизайн на блога в скриптовете на папките можете да намерите файловете jquery и bootstrap.  
Поставете необходимите скриптове и стилове там, завършвайки със следната структура:



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:

Тогава трябва да използваме тези стилове и скриптове, като настроим базово оформление в app / resources / views / base.html.twig.  
Настройте базовото оформление, както желаете, или използвайте следното:

*{#*  
 *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*](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;** 2018 - 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**>

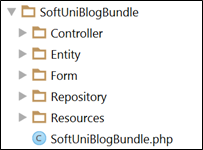
# II. Symfony Base Project Overview

II. Symfony Основен преглед на проекта

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](http://symfony.com/doc/current/bundles.html), which can be treated as a plugin later. A **bundle** often has **Controllers**, **Entities** and related components (e.g. **Repositories**, **Forms**, **Commands**…)

Symfony е модулна корпоративна уеб-рамка, която се предлага с поддръжка на солиден доставчик, система за пакети, корпоративни механизми и е най-подходяща за MVC архитектура.  
Първоначално проектът е снабден с основен пакет, който по-късно може да се третира като приставка. Пакетът често включва контролери, субекти и свързани компоненти (например хранилища, формуляри, команди ...)



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

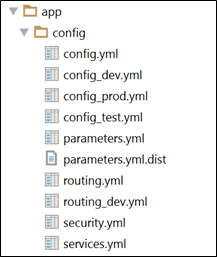
Стандартните шаблони (изгледи) се намират в папката (приложението) и обикновено са разделени в папка, наречена след имената на контролерите.



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 configuration files for the [Doctrine](http://www.doctrine-project.org/) connection are defined among [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.

Де факто стандартният View Engine в Symfony е Twig.  
Базовата конфигурация на проекта се поставя в app / config, където конфигурационните файлове за връзката Доктрина са дефинирани в Управление на сигурността, Правила за маршрутизиране, регистриране на Услуги и т.н.



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**.

Много е важно файлът parameters.yml.dist да съдържа същите клавиши като тези в parameters.yml, тъй като инсталирането на нов пакет ще изтрие неизползваните двойки.

# III. User Authentication

III. Потребителско удостоверяване

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.

Symfony разполага с много мощна система за управление на сигурността, където общата работа за проверка на разрешенията на потребителите и изпращане на заявката е добре извлечена, но конфигурацията може да е объркваща. В стъпката по-долу ще настроим регистрация и процес на вход и достъп до защитено съдържание.

## 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](http://symfony.com/doc/current/doctrine/console.html) 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:

Нашите потребители трябва да се съхраняват в базата данни. Това означава, че се нуждаем от таблица "потребители". Тъй като таблиците се представят като обекти в парадигмата "Mapping Object / Relation Mapping", трябва да създадем обект, който представлява тази таблица. Класовете (обектите), които представляват таблици, се наричат Модели и Единици.  
В де факто, стандартен ORM в Symfony, наречен Doctrine, тези обекти се наричат Entities.  
Да дефинираме правилата си за потребител:  
• Трябва да има уникално име за вход, да речем имейл  
• Трябва да има парола  
• Трябва да има пълно име, да речем fullName  
Доктрината идва с удобен конзолен инструмент за управление на базата данни и създаване на обекти. Нека използваме "Доктрина", за да създадем лице, наречено "Потребител", като използваме съветника за генериране на обекти. За да направите това, трябва да отворите терминален прозорец в основната директория на проекта и да въведете следната команда:

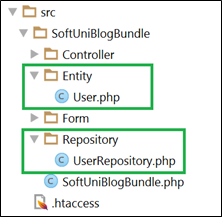
|  |
| --- |
| php bin/console doctrine:generate:entity |

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.

Това ще ни накара да въведем име на обект. Предприятията се представят с пакета, към който трябва да принадлежат. Нашият пакет се нарича SoftUniBlogBundle (основното име е AppBundle), така че ще въведете в SoftUniBlogBundle: User (или AppBundle: User, ако вашият пакет се нарича AppBundle).  
След това ще ни поискат свойствата (полетата) на потребителския обект. Както казахме по-горе, ще има имейл, парола и fullName, всички от тях са текстови полета (низове). Имейлът трябва да е уникален, така че когато бъдете подканени за уникалност там, въведете "true" вместо просто да кликнете върху enter (което по подразбиране е невярно)  
Когато последното поле (fullName) е създадено и се появи подкана за друг, просто кликнете върху enter, за да излезете от съветника. Това ще създаде потребителското лице и съответния UserRepository.



## 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”

Както казахме, Symfony идва с няколко конфигурационни файла, един от които се нарича security.yml. Трябва да посочим няколко неща, като например:  
• Как ще бъде шифрована паролата и на коя единица  
• Кой обект ще бъде използван за потребителите и кое от неговите полета ще бъде полето за потребителско име (например имейл, потребителско име и т.н.)  
• Където ще се намира формулярът за вход (име на маршрут)  
• Къде ще се публикува този формуляр за вход  
По-долу има файл security.yml, който има следната конфигурация:  
• Пакетът се нарича SoftUniBlogBundle  
• Потребителското звено се нарича Потребител и полето му за потребителско име се нарича имейл  
• Формулярът за вход ще бъде достъпен и публикуван в "security\_login"  
• След успешно влизане, потребителят ще бъде пренасочен към "blog\_index"

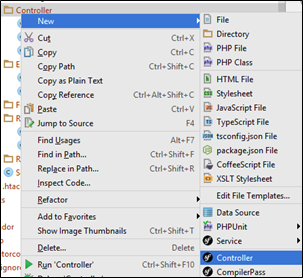
**security:**  
 **encoders:**  
*# Our user class and the algorithm we'll use to encode passwords*  
 *# http://symfony.com/doc/current/book/security.html#encoding-the-user-s-password*  
**SoftUniBlogBundle\Entity\User:** bcrypt  
  
 **providers:**  
*# in this example, users are stored via Doctrine 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 }*

## 3. Формуляр за вход. 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.

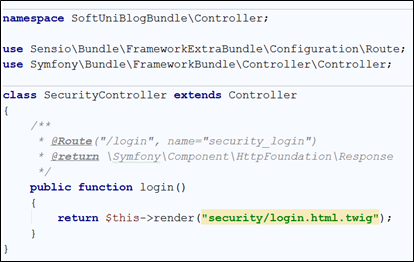
Let’s call our Controller “SecurityController“:

За да създадете формуляр за вход, трябва да създадем така наречения контролер, който ще слуша по този маршрут (който по-горе наричаме "security\_login") и ще направим изгледа с формуляра за вход, когато някой отива в / login маршрута.  
Нека се обадим на нашия контролер "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**)

След това се нуждаем от метод (който ще наричаме "login ()"), който слуша на маршрута "/ login" и прави изглед (нека го насочим към файл за вход.html.twig, който се намира в папката за защита)



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] J

Цветът на жълтия фон в името на изгледа ни показва, че все още нямаме този изглед. Можем лесно да го създадем, като щракнете върху [Alt + Enter] 



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

Преди да объркаме всички оформления (които сме настроили и ще използваме в следващите глави), ние просто ще създадем обикновен формуляр за вход без стилове.  
Трябва да дефинираме маркер <form>, който се публикува в маршрута security\_login. Twig, за щастие, осигурява пътека за функции (), който използва имена на маршрути и генерира URL адреси от тях



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 J)

Формулата е наречена "удостоверяване", защото по-късно ще използваме това име, за да генерираме CSRF означение  
Symfony сигурност изисква потребителското име (което е имейл в нашия случай) и паролите за парола да бъдат наименувани съответно \_username и \_password  
Трябва да дефинираме тези две текстови полета (или полето за парола за типа парола )



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

И поле за CSRF Token, използвайки помощния метод на Twig csrf\_token (), който приема името на формуляра.



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

Сега отварянето на http: // localhost: 8000 / login трябва да направи този формуляр за вход



Not the most beautiful login form J But still it’s there! J

Не най-красивата форма за вход  Но все пак е там! 

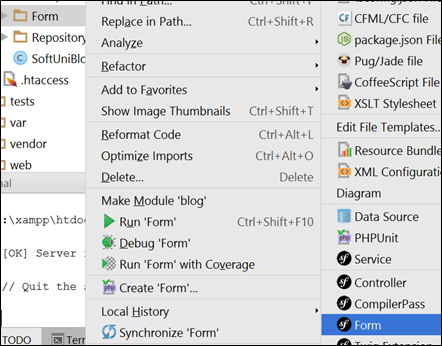
4. Register Form Формуляр за регистрация

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](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:

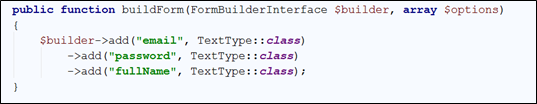
Какво представлява формулярът за вход без потребители - нищо. За да имаме потребители, се нуждаем от регистрационен формуляр. По аналогия отворете вече генерирания DefaultController или създайте нов (например UsersController) и действие, което слуша на "register".  
Това ще направи формата по същия начин, но също така трябва да се справи с тази форма.  
За да може даден формуляр да работи с дадено предприятие, той се нуждае от съответния FormType. Преди да можем да продължим да създаваме регистърното действие, трябва да създадем тип на формуляра. Създайте папка "Form" в src / SoftUniBlogBundle. След това създайте формуляр тип, както следва:



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.

Да го наречем UserType.  
В метода на скеле "buildForm ()" трябва да дефинираме двойките - полетата на обекта и техните съответни типове във формата. Всичките ни три полета са текстови типове, така че ще използваме текстов тип от пространството на имена Symfony \ Component \ Form \ Extension \ Core \ Type \ TextType.



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

Връщайки се към метода за регистрация на контролера, сега можем да създадем форма на UserType.

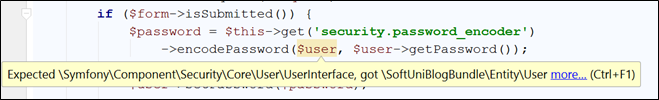


We have said here: Create a form of user type and after it’s submitted fill the $user object.

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

Тук казахме: Създайте форма на потребителски тип и след подаването му попълнете потребителския обект $.  
След това трябва да посочим метода - след като формулярът бъде изпратен и всички потвърждения са предадени (например текстовете са попълнени), запазете потребителското лице в базата данни.  
Има един възможен проблем - паролата ще стане ясно в ПБ. За щастие в конфигурацията за сигурност сме регистрирали доставчик на шифроване, за да можем да използваме този доставчик за кодиране на паролата и след това да я изпратим в базата данни



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.

Енкодерът работи само върху обекти на UserInterface и нашите потребители не са един. Това, от което се нуждаем, е да отидем до потребителското звено и да го направим да изпълни интерфейса UserInterface.

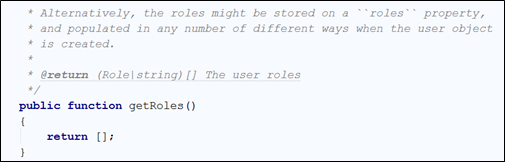


Then implement all of the missing methods with [ALT+ENTER].

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:

След това въведете всички липсващи методи с [ALT + ENTER].  
Можете да оставите повечето празни (автоматично генерирани), но някои от тях трябва да бъдат попълнени.  
Първият метод е getRoles (). Той трябва да върне масив от роли (може да е празен), но не и нула:



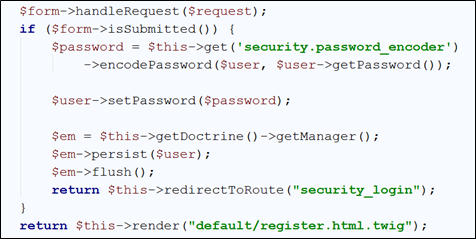
The other one is the getUsername() method, which will be used for authentication. We need to return our $email field in it, because that’s our username:

Другата е методът getUsername (), който ще се използва за удостоверяване. Трябва да върнем полето за имейл домейн в него, защото това е нашето потребителско име:



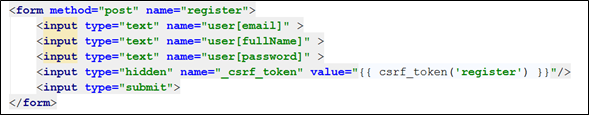
Now going back to the registration action, the error is gone. We can safely set the encoded password to the user object and persist it via [EntityManager](http://www.doctrine-project.org/api/orm/2.5/class-Doctrine.ORM.EntityManager.html) to the database

След като се върнахме към действието за регистрация, грешката е изчезнала. Можем безопасно да зададем кодираната парола на потребителския обект и да го задържим чрез EntityManager към базата данни



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 J

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:



IV. Creating Articles Създаване на статии

## 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](https://softuni.bg/downloads/svn/soft-tech/May-2017/Software-Technologies-July-2017/05.%20Software-Technologies-PHP-Blog-Basic-Functionality/05.%20Software-Technologies-PHP-Blog-Basic-Functionality-PHP-MySQL-Blog-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).

## 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:

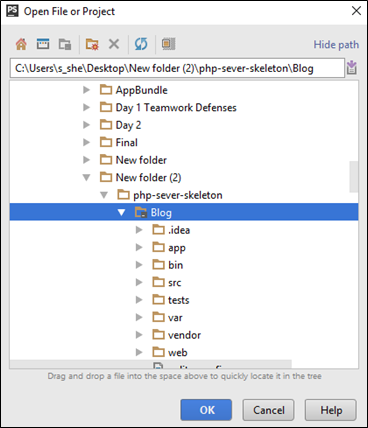
For this step, we will open the project with **PhpStorm** or **IntelliJ** Idea. Starting from the home screen, click on “**Open**”:

0. Стартирайте MySQL (само ако сте тук от самото начало)  
Прескочете тази стъпка, ако сте преминали през горните глави III.  
Ако все още четете:  
Изтеглете скелета на проекта, извлечете го по най-краткия път, който можете да направите, напр. в c: \ project.  
Преди да започнем да използваме нашия блог, трябва да създадем база данни.  
1. Отворете проекта (само ако сте направили стъпка 0.)  
Прескочете тази стъпка, ако сте преминали през горните глави III.  
Ако все още четете:  
За тази стъпка ще отворим проекта с PhpStorm или IntelliJ Idea. Започвайки от началния екран, кликнете върху "Отвори":



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

Намерете папката на скелета, която ви дадохме, и изберете папката "Блог" от извлечената папка (например 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.

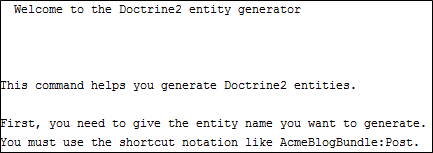
## 2. Create the Article Entity

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](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**:

След като кликнете върху "OK", проектът трябва да започне да се зарежда и индексира. След няколко секунди / минути в зависимост от вашия компютър, вие ще можете да работите с проекта.  
2. Създайте Entity Член  
Отворете терминала или командния ред (CMD) в основната папка на проекта за блога. Нека моделираме статиите си. Това означава, че ще създадем дефиниращите свойства на една статия. За да направим това, трябва да създадем учебно заведение. Нашата организация ще опише какво ще съхраняваме в нашата база данни. Следната команда ще стартира съветника за генератор на обекти:

**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:

Сега трябва да изберем подходящо име за нашата организация. Използвайте следното име:

**SoftUniBlogBundle:Article**

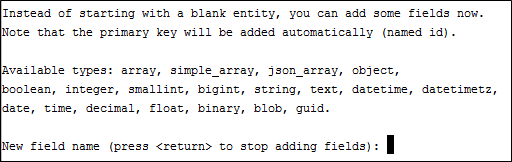
The result should be the following:

Резултатът трябва да бъде следният:



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

Просто натиснете [Enter]. Сега трябва да дефинираме свойствата на нашата организация. трябва да видите това:



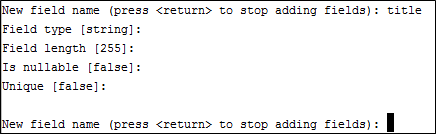
Our **first** **field** will be the “**title**” of our article. Just write “**title**” and press **[Enter]**. You should see this:

Първото ни поле ще бъде "заглавието" на нашата статия. Просто напишете "заглавие" и натиснете [Enter]. Трябва да видите това:



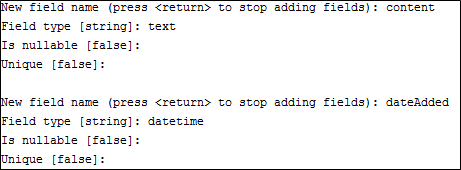
Press **[Enter]**. You should see “**Field length [255]**”. Press ‘**Enter**’ again. You will be asked if you want to make the field **nullable**. Press **[Enter]**. Finally, you will be asked to make your field **unique**. Just press **[Enter]** one more time. Now you should see this:

Натиснете [Enter]. Трябва да видите "Дължина на полето" [255] ". Натиснете "Enter" отново. Ще бъдете попитани дали искате да направите полето неутрално. Натиснете [Enter]. И накрая, ще бъдете помолени да направите полето си уникално. Просто натиснете [Enter] още веднъж. Сега трябва да видите това:



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

Подобно на това, трябва да създадем още 2 полета за "съдържание" и "dateAdded". Ето как ги създаваме:



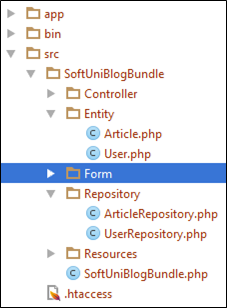
Finally, press ‘**Enter**’ one more time to close the wizard. You should see this:

Накрая натиснете "Enter" още веднъж, за да затворите съветника. Трябва да видите това:



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

Да видим проекта в 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.

## 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](http://www.mysqltutorial.org/mysql-primary-key/) for our table. On top of our entity we should see something that looks like a comment:

Добре, доктрината е създала "член" и "ArticleRepository", която е специален тип класа. Неговата задача е да управляваме данните си и да опростим работата си с базата данни.  
3. Добавете резюме към обекта на члена  
Нека да влезем в субекта "Член", който Доктрината създаде в предишната стъпка. То трябва да съдържа всички полета, създадени от терминала, плюс едно специално поле "id". Това е основният ключ за нашата маса. На горната част на нашата организация трябва да видим нещо, което прилича на коментар:



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

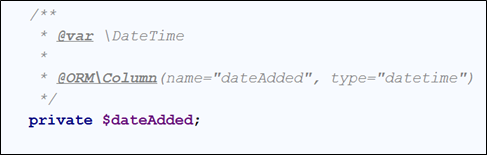
Това обаче не е просто коментар. Това е анотация. По-конкретно, това е анотация на Учение. Тя разказва доктрината как ще бъдат наречени таблиците и полетата в базата данни. На пръв поглед виждаме пояснението

|  |
| --- |
| \* @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**”.

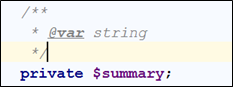
Now we need to create some fields, that will not get saved into the database. Find the “$dateAdded” field. You should see something like this:

Това определя името на нашата таблица в базата данни. Имената на таблиците в базата данни трябва да бъдат множествено. Поради тази причина преименувайте таблицата на "статии".  
Сега трябва да създадем няколко полета, които няма да бъдат записани в базата данни. Намерете полето "$ dateAdded". Трябва да видите нещо подобно:



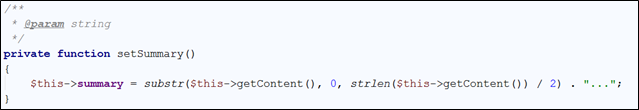
Below that, first add a new private field called “summary”. It will hold the short summary of the article:

По-долу първо добавете ново частно поле, наречено "резюме". Той ще съдържа кратко резюме на статията:



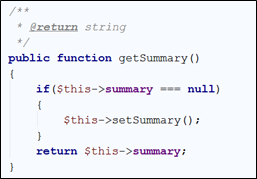
Then we need to create [Mutator and Accessor](http://www.refulz.com/mutator-and-accessor-methods-in-php/) (Getter and Setter) **methods** for the summary. Let’s first start with the **mutator**. Its job is to **set the value** of the summary to **half of the article content**. The code should look like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

След това трябва да създадем методи за Mutator и Accessor (Getter и Setter) за резюмето. Нека първо да започнем с мутатора. Неговата задача е да зададе стойността на резюмето на половината от съдържанието на статията. Кодът трябва да изглежда така:



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**:

Сега трябва да създадем аксесоара. То трябва просто да върне запазената стойност в обобщената променлива. Ако обаче обобщението е празно, то трябва да се обади на мутатора, за да зададе стойността:



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

## 1. 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:

Свършихме с обобщената променлива, но все още имаме по-голяма променлива за изпълнение.  
4. Създайте връзка между потребителя и статията  
Дойдохме до мястото, където трябва да свързваме всеки потребител с неговите статии. За да направим това, ще създадем още две полета в обекта "Член". Точно под частното поле за резюмето, което сме създали в предишната стъпка, създайте ново частно поле, наречено "authorId". Използвайки това поле, всяка статия ще знае кой е нейният автор:



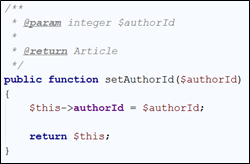
We’ve come to the part where we must connect each user with his articles. To do that, we will create 2 more field in the “**Article**” entity. Just below the private summary field, that we’ve created in the previous step, create new private field called “authorId”. Using that field, each article will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

Дойдохме до мястото, където трябва да свързваме всеки потребител с неговите статии. За да направим това, ще създадем още две полета в обекта "Член". Точно под частното поле за резюмето, което сме създали в предишната стъпка, създайте ново частно поле, наречено "authorId". Използвайки това поле, всяка статия ще знае кой е нейният автор:



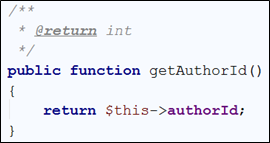
You have probably noticed that we’re going to **save this field in the table** using the @ORM annotation. This will **create a column in the table**, which will keep integer, representing a user. Similarly, to the summary, we need to create **getter and setter** methods for this field. Again, we’re starting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

Вероятно сте забелязали, че ще запазим това поле в таблицата с помощта на @ORM анотацията. Това ще създаде колона в таблицата, която ще запази цяло число, представляващо потребител. По същия начин, за резюмето, трябва да създадем методи за получаване и създаване в тази област. Отново започваме с мутатора:



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

Едно специално нещо, което трябва да отбележим тук, е, че мутаторът всъщност връща обекта, който се променя. Сега просто създайте аксесоара:



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**”:

Свършихме с автора, но връзката все още не е готова. За да може нашата статия действително да има автор, трябва да декларираме частно поле от тип "Потребител":



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](https://en.wikipedia.org/wiki/One-to-many_(data_model)) 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** Userentity.

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

Още нови неща! Използваме 2 нови пояснения. Първата е анотацията "ManyToOne". Много от взаимоотношенията представляват връзка "Един към много" от страна на "много". В нашия случай ще използваме "една към много отношения", за да кажем на програмата, че един потребител ще има много статии. Тъй като работим с члена на член, казваме Доктрината, че много от нашите статии ще отговарят на един потребител. Параметрите "inversedBy" разказват доктрината, че потребителското звено ще има частно поле, наречено "статии", което ще запази всички статии на един потребител. Другата анотация е "JoinColumn", която казва Учение, как ще свържем полетата в нашите обекти. Нашето пояснение казва Учение, че полето "authorId" в нашата статия ще съответства на полето "id" от User entity.  
Сега трябва да създадем създателя на полето за автора:



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:

Това е всичко, ние сме готови с членство обект за тази стъпка. Трябва да направим връзката "един към много" от страна на потребителското лице. Точно под частното поле "парола", създайте следното поле:



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:

Това поле ще бъде от тип ArrayCollection, което ще запази всички текущи потребителски публикации. Както можете да видите, ние дефинираме една към много взаимоотношения с члена на статията, използвайки полето за автор, което сме създали по-рано. За това поле няма да създаваме сетер като предишните. Първо, трябва да създадем поглъщащото устройство:



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:

Настройващото устройство обаче ще бъде малко по-различно. То трябва да добави нова публикация към текущите потребителски публикации. За да направите това, трябва да напишете следния код:

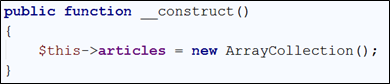


We’re done with the connection for now. Later **we will update the database** schema.

## 5. Set Default Values for our Entities

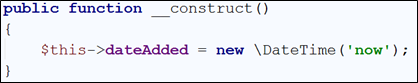
Our next job is to create the so-called [constructors](http://php.net/manual/en/oop4.constructor.php) for our entities. The constructors are special methods that are called **each time a new object from the entity is created**. Let’s start with the User entity. Its constructor should be the following:

В момента свършихме с връзката. По-късно ще актуализираме схемата на базата данни.  
5. Задайте стойности по подразбиране за нашите Entities  
Нашата следваща работа е да създадем така наречените конструктори за нашите организации. Конструкторите са специални методи, които се наричат всеки път, когато се създава нов обект от предприятието. Нека да започнем с потребителското звено. Неговият конструктор трябва да бъде следният:



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

Всеки път, когато създаваме нов потребител, той получава празен масив от статии. Членът от друга страна трябва да изглежда така:



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).

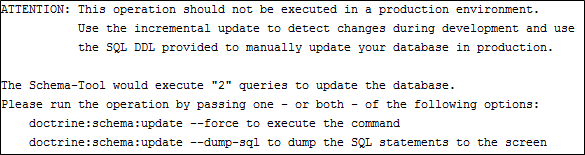
## 6. Updating the DB with our Article Entity

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](http://docs.doctrine-project.org/en/latest/). **Open** a **Terminal**/**CMD** in the project **root** **folder**. Let’s write the following command:

Всеки път, когато се създава нова статия, този конструктор ще добави текущото време.  
Ние сме готови с тази част, сега можем да актуализираме базата данни (схема).  
6. Актуализиране на ПБ с нашия член  
Има много начини да обновите или създадете таблиците, от които се нуждаем. Първата е да ги създадете ръчно. Това ще отнеме много време и затова няма да го направим по този начин. Ние ще ги създадем с помощта на Учение. Отворете терминал / CMD в основната директория на проекта. Нека да напишем следната команда:

|  |
| --- |
| 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:

Това основно ни казва, че правим операция, която не е безопасна. За да го направим, трябва да принудим Учение да изпълни нашата заповед. За да направим това, трябва да добавим "-force" след нашата предишна команда:

|  |
| --- |
| 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:

Ако разгледаме DB в HeidiSQL, ще видим, че е създадена таблицата "статии":

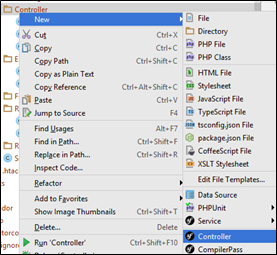


We are ready, to start making our blog.

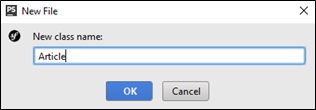
## 7. Creating the Article Controller

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:

Ние сме готови да започнем да правим нашия блог.  
7. Създаване на контролер на статии  
Сега трябва да създадем клас, който да контролира как статиите ще бъдат разглеждани, създадени, редактирани и изтривани. Ще го създадем в папката "Контролер". Ако използвате PhpStorm или IntelliJ IDEA и имате инсталиран приставката Symfony, трябва да го видите, когато щракнете с десния бутон върху папката Controller:

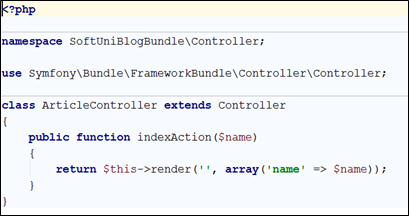


Give it the name ArticleController:



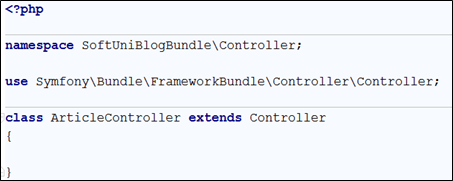
We have just created an ArticleController in the **Controller** folder, that looks like this:

Току-що създадохме ArticleController в папката Controller, който изглежда така:



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

Изтрийте метода indexAction, няма да имаме нужда от него. Трябва да сме доволни от следния резултат:

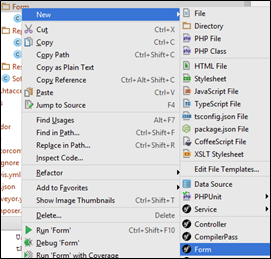


We have a controller, but we need **form template**.

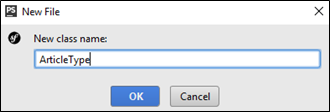
## 8. Creating the Article Type Form

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**:

Имаме контролер, но имаме нужда от формата.  
8. Създаване на формуляр за типовете статии  
Следващата ни стъпка е да създадем шаблон за формуляри, който ще попълваме всеки път, когато създаваме или редактираме статия. За да създадете този формуляр, просто щракнете с десния бутон на мишката върху папката Form и изберете новата форма:



Give it the name “ArticleType”:



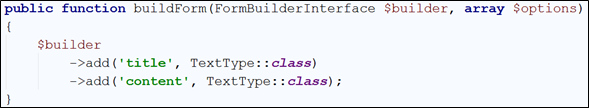
We should receive something like this:

Трябва да получим нещо подобно:



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:

Може да забележите, че имаме 2 празни функции. "BuildForm" ще създаде нашата форма и "configureOptions" ще каже на нашия формуляр, че тя е за членството. Нека да започнем с създателя на формуляра:

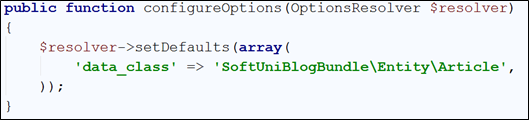


It’s a pretty simple form. It should only contain **title** and **content** fields, both of type text. You should use specific using for the “TextType” to work. If you have another one **ending in \TextType** already imported – delete it and add:

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

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

Нека да създадем логиката за функцията "configureOptions":



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:

Стойността по подразбиране за нашия резолтор казва на контролера, че ще използва формуляра, в какъв вид обект да съхрани датата от нашия формуляр. Това е.  
9. Прилагане на функцията за създаване на статии  
Върнете се в контролера на статиите, трябва да създадете нова функция. Ще го наречем "createAction" и ще създадем няколко анотации за него:



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:

Нека да започнем от първата анотация. Той казва на нашия проект, че функцията ще получи един параметър от тип Request. Ще запазим искането за друго време. Втората анотация е по-интересно. Тя определя "маршрут". Маршрутът представлява URL адреса, който съответства на настоящия метод. В този случай функцията ще бъде извикана, когато отидем в http: // localhost: 8000 / article / create. Всеки път, когато използваме този URL адрес, рутерът ще се обади на нашата функция. За да опростим пренасочването между нашите страници, даваме по-просто име като "article\_create". Третата анотация е да се уверите, че само влезлите в системата потребители ще използват нашата функция. Без него всеки гост ще може да създаде нова статия и ние не искаме това. Окончателният параметър указва, че нашата функция ще върне отговор. Ще поговорим по-късно. За да могат тези пояснения да работят правилно, уверете се, че използвате правилния внос:

**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:

Сега нека да напишем някакъв истински код. Във функцията напишете следното:



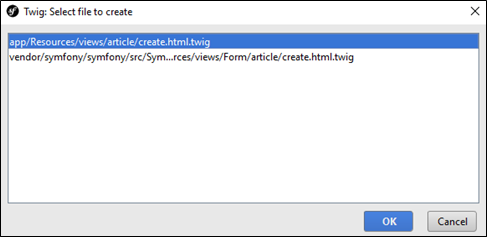
What is this code doing? It’s simple – it **creates a new article**. Then it **creates** a **new form** from the template we’ve created earlier and tells the **form** that it **should** **fill our new article**. Finally, it **sends the form to a** **view** that we are going to **render** on the screen.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

Какво прави този код? Това е просто - създава нова статия. След това създава нов формуляр от създадения по-рано шаблон и казва на формуляра, че трябва да попълни новата ни статия. Накрая, тя изпраща формуляра на изгледа, който ще представим на екрана. Наклон означава рисуване. Симфони използва Twig. Twig е шаблон за двигател, чиято работа е да показваме данните си по-лесен начин, отколкото да създадем HTML от себе си. Важната част тук е, че все още нямаме такъв шаблон и ни казва "PhpStorm" (IntelliJ IDEA), като направим жълт правоъгълник над името на нашия шаблон. За да го създадете, просто кликнете върху името на шаблона и след това натиснете [Alt] + [Enter]. Това ще отвори контекстно меню, в което се обаждате, кажете на IDE да създаде шаблона за вас:

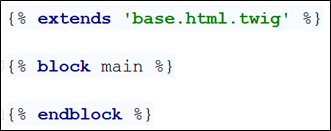


Then you need to choose the first option:

След това трябва да изберете първата опция:



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

Поздравления, гледате празен шаблон. Напишете следния код:

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:

Този код прави 3 неща. Първата е да се "разшири" съществуващ шаблон. Какво означава това? Това означава, че сме създали основния дизайн на блога за вас, включително всички стилове и скриптове, от които може да се нуждаете. Вече можете да използвате отново този основен шаблон във всички шаблони, които ще създадете. Второто изречение замества блок, наречен "основен" в базовия шаблон. Това означава, че всичкият HTML код в основния шаблон за "основния" блок ще бъде заменен с кода, който ще напишеш за секунда.  
Само защото не искаме да се фокусирате върху HTML и Twig, ще дадем целия код, който трябва да напишете в главния блок:

**<div class="container body-content span=8 offset=2"**>

|  |
| --- |
| <**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:

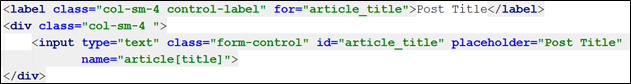
Нека обаче да обясним няколко части от този шаблон.  
Първата част, която ще обсъдим, е:



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:

Използваме някои CSS класове, тази част, с която трябва да сте запознати. Наистина интересните части са атрибутите на действие и метод на нашата форма. Първо, ще говорим за метода. Този атрибут определя какъв тип искане ще използваме. За да опростим нещата, нека скоро да обясним исканията. Искането, което ще използваме, е "POST". Това означава, че искаме да изпратим данни на някакво място. В нашия случай той казва на HTTP протокола, че искаме да изпратим заглавието и съдържанието си на място в нашия блог. Другият тип заявка, от която се интересуваме, е "GET". Той казва на HTTP, че искаме да получим някои данни от някъде. Има и други видове искания, но сега няма да ви притесняваме. Нека да поговорим за действието. Атрибутът за действие дефинира от / до къде искаме да получим / поставим нашите данни. Помнете ли името на маршрута, който дадехме нашата функция по-рано? Да, искаме да изпратим заявка за POST с нашето заглавие и съдържание обратно към функцията, която създадохме по-рано. Ще видим как да използваме данните от искането по-късно.  
Втората част от шаблона, която заслужава бърз поглед, е:

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:

Първото нещо, което трябва да забележите, е, че атрибутът за етикета и идентификационния атрибут на входа имат една и съща стойност. Сега погледнете името на атрибута на входа. Тя изглежда като речник стойност. Когато картографирате обектите си в шаблоните за клони, важно е да отбележите, че първата част на името е името на обекта. В квадратните скоби поставяме името на полето от обекта, който ще попълним.  
Друго интересно нещо е:



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:

Това е специален клонка код. Той създава ново невидимо поле в нашата форма, което потвърждава нашата форма. Без него нашата форма няма да работи. Искате да знаете повече, които трябва да проверите за CSRF.  
И накрая, още един специален клон, който видяхме и по-рано:



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:

Тази команда "път" използва името на маршрута и пренасочва към дадения маршрут.  
Достатъчно за шаблоните за сега, нека да започнем блога и да видим дали работи. За да направим това, трябва да отворим терминала / CMD в основната папка на нашия блог или да използваме вградения терминал в PhpStorm (IntelliJ Idea). Не забравяйте да стартирате MySQL, ако не сте го направили. Въведете следната команда:

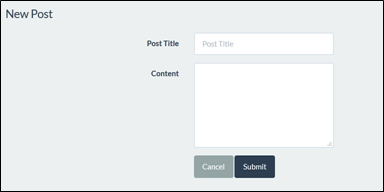
**php bin/console server:run**

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:

Отворете браузъра и отидете на адреса. Трябва да видите почти празна страница. Сега трябва да регистрирате нов потребител и да влезете. След като влезете в URL адреса въведете http: // localhost: 8000 / article / create. Тя трябва да ви пренасочи към форма като тази:



**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:

Попълнете формуляра и кликнете върху "Изпращане". Страницата се опреснява, но ако проверим таблицата в базата данни, тя е празна. Нека да отстраним проблема. Връщайте се към функцията си в контролера на статиите. Проблемът е, че никога не сме използвали данните от нашия формуляр. Добавете към функцията си следния код:



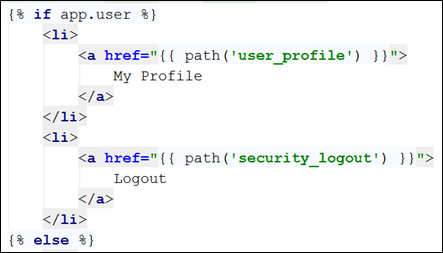
This code **takes the data from request (make sure the imported “use” statement at the beginning of the class is Symfony\Component\HttpFoundation\Request)** and **fills** the **form**. After the form is filled, we check if there is **any data** in the form and if it **is valid**. If everything is okay, 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:

Този код отнема данните от заявката (уверете се, че внесеният израз "use" в началото на класа е Symfony \ Component \ HttpFoundation \ Request) и попълва формуляра. След като формулярът е попълнен, проверяваме дали има данни във формата и дали е валидна. Ако всичко е наред, тогава получаваме в момента регистрирания потребител и го възлагаме като автор на статията. Тогава ние получаваме управителя на предприятието от доктрината и използвайки функцията "persist" добавяме нашата нова статия в базата данни. Накрая наричаме функцията "флъш", която изпраща статията в нашата база данни. След като статията бъде изпратена в базата данни, пренасочваме изгледа към индексната страница на нашия блог.  
Докато променяме кода, отворете основния шаблон:



Find this part of the code:



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

Добавете нов елемент "<li>", който ще се пренасочи към страницата за създаване на статията:



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

Да отидем на нашия блог и да влезем. Сега в дясната страна на навигационната лента ще видите новия бутон:



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:

Нека се опитаме да създадем нова статия. След натискане на бутона "Изпращане" трябва да пренасочим към началната страница. Да видим дали имаме нещо ново в базата данни. В HeidiSQL отворете таблицата с статиите:



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.

Хури, направихме го! Сега можем да създадем статии. Проблемът е, че не можем да ги видим в нашия блог. Нека да приложим това.

# 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**:

V. Списък на статиите  
1. Вписване на всички статии  
Хайде да отидем до домашния контролер. Когато го отворите, ще намерите функция, наречена "indexAction". Единствената му работа в момента е да се обади на индексния изглед, без никакви данни. Ще променим това. Напишете следния код в началото на функцията:



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

Това ще ни даде всички статии от базата данни. Нека ги предадем на изгледа. Редактирайте изявлението за връщане така:

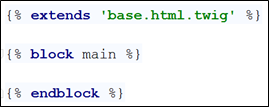


We’re done here, go to the view, and examine it:

Преминахме тук, преминете към изгледа и я разгледайте:



You should see this:



In the main block, write the following code:

В главния блок напишете следния код:

|  |
| --- |
| <**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:

Има няколко ключови моменти, които искаме да разгледаме. Първият е:



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:

Това е проста линия на foreach в клонка. Той ще пресече масива от статии, които сме изпратили на изгледа през контролера. Също така има заключително изявление по няколко реда по-долу:



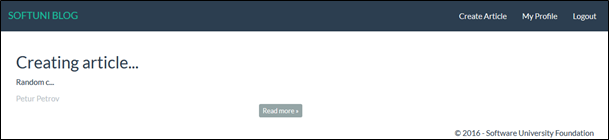
Between those two rows, there are a couple of twig calls. The first one is:

Между тези два реда има няколко клонки. Първият е:

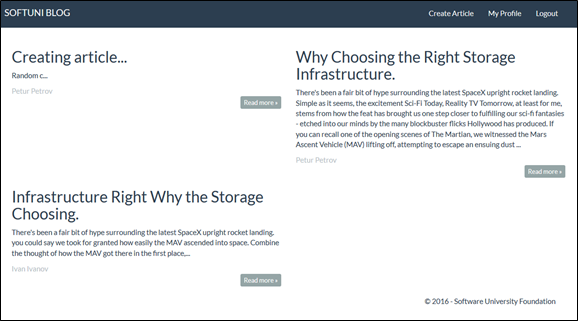


This will print the title for each article. We have the same thing for the summary and author of the article.

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

Това ще отпечата заглавието за всяка статия. Имаме едно и също нещо за резюмето и автора на статията.  
Засега нека да започнем блога и да видим какво имаме:

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.

## 2. Showing Single Article

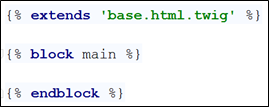
To implement the single article page, we need to go back to the article controller. Create the following method:

Изглежда добре. Проблемът е, че ако натиснете "Прочетете повече" нищо не се случва. Трябва да поправим това.  
2. Показване на един член  
За да внедрим страницата с една статия, трябва да се върнем към контролера на статиите. Създайте следния метод:



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:

Нека да разгледаме поясненията. Анотацията на маршрута има къдрави скоби ('{', '}') и някои параметри в тях. Това е параметърът, който изисква функцията. Всичко останало е стандартно. Ако погледнем функцията, можем да видим, че търсим конкретен идентификатор в базата данни. Този ред ще ни даде само статията с дадения id. След това го изпращаме до изгледа. Създайте изгледа, както направихме по-рано. Генерираният изглед ще съдържа основната структура, за която вече сме запознати:



Write the following code in the main block:

Напишете следния код в главния блок:

<**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.

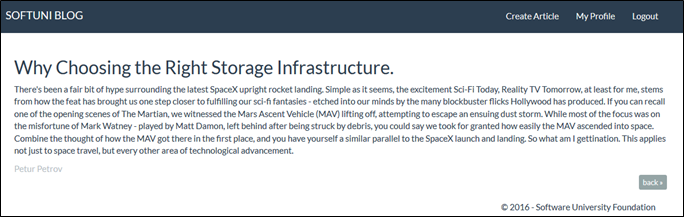
Let’s start the blog and see if it works. The answer is no, it doesn’t. Right now, the button read more doesn’t redirect to the right route. Let’s go back to the index view and find this piece of code:

Този код е много прост, като единствената разлика от предишната е, че имаме само една статия и ние отпечатваме съдържанието вместо обобщението.  
Нека да започнем блога и да видим дали работи. Отговорът е не, не. В момента бутонът "Прочетете повече" не пренасочва към правилния маршрут. Нека да се върнем към индексния изглед и да намерим това парче код:



Change it to: Променете го на:



Let’s try it now: Нека да опитаме сега:Congratulations! If everything works okay, you’ve just created a very simple blog system where users can log in and post articles.

Поздравления! Ако всичко работи добре, току-що сте създали много проста блог система, в която потребителите могат да влизат и да публикуват статии.