# **Blog: Java and Spring**

This document defines a complete walkthrough of creating a **Blog** application with the [Spring](http://spring.io/) Framework, from setting up the framework through [authentication](http://projects.spring.io/spring-security/) module, ending up with creating a **CRUD** around [Doctrine](http://hibernate.org/orm/) entities.

**Chapters from III to V are for advanced users. There’s a** [**skeleton**](http://softuni.bg/downloads/svn/soft-tech/May-2017/Software-Technologies-July-2017/10.%20Software-Technologies-Java-Blog-Basic-Functionality/10.%20Software-Technologies-Java-Blog-Skeleton.zip) **which you can use and start from chapter VI, after you set up the JDK in chapter II.**

Този документ дефинира пълен пътеводител за създаването на приложение за блогове с пролетната рамка, от създаването на рамката чрез модула за удостоверяване, завършвайки с създаването на CRUD около доктрините.  
Глави от III до V са за напреднали потребители. Има скелет, който можете да използвате и да започнете от глава VI, след като създадете JDK в глава II.

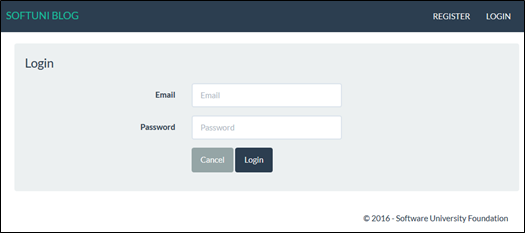
# I. What Will You Create

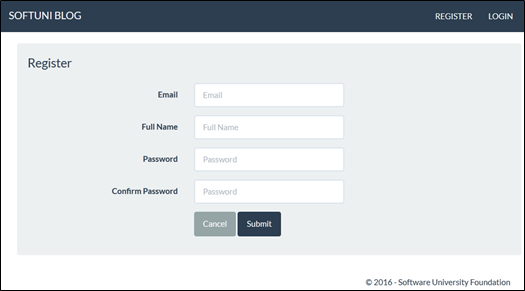
In the end of the whole tutorial, you should have blog which supports the following functionality:

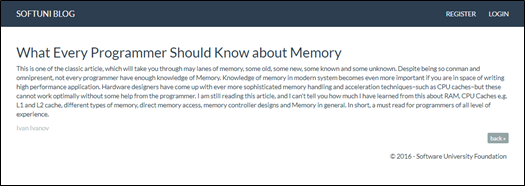
* **Register** and **Login** Users
* **Create** Articles
* **Edit** Articles
* **Delete** Articles
* **Responsive Design** using **Bootstrap**
* **More…**

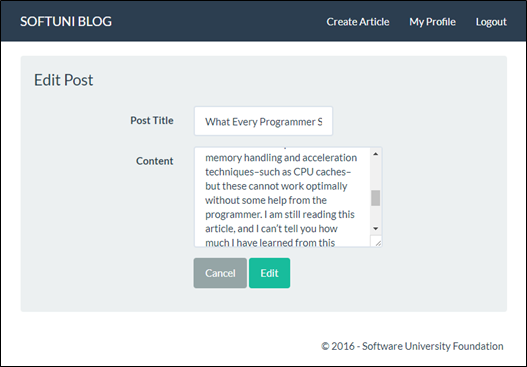
**Pictures:**

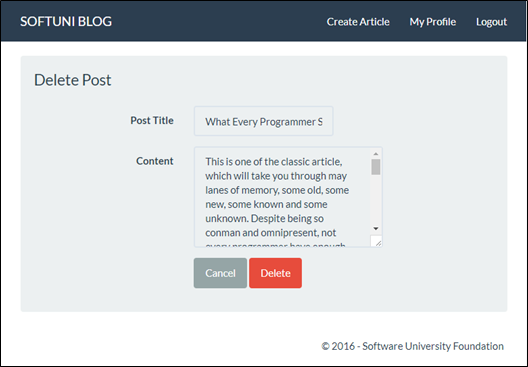














# II. Set Up JDK and IntelliJ Idea Configuration

Before we start you need to download the [Java Development Kit](http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html), also known as **JDK**. Download the "**Java SE Development Kit 8u(latest version number)**". After downloading it, install it **without changing the installation directory**. That will install it in the "**Program Files**" folder if you are on **Windows**.

Преди да започнете, трябва да изтеглите комплекта за разработка Java, известен също като JDK. Изтеглете "Java SE Development Kit 8u (най-новата версия на версията)". След като го изтеглите, инсталирайте го, без да променяте инсталационната директория. Това ще го инсталирате в папката "Програмни файлове", ако сте в Windows.

### Using the Skeleton

If you are using the skeleton and see something like this:

Използване на скелета  
Ако използвате скелета и вижте нещо подобно:

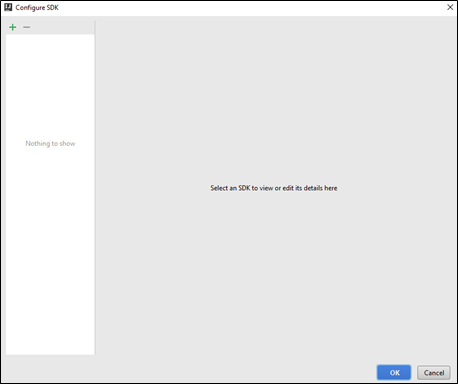
You should set-up the SDK. Click on "**Setup SDK**". You should see this screen:

Трябва да настроите SDK. Кликнете върху "Настройка SDK". Трябва да видите този екран:



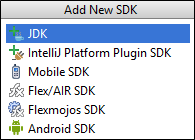
Click on "**Configure**" and see if you receive this screen:

Кликнете върху "Конфигуриране" и вижте дали ще получите този екран:



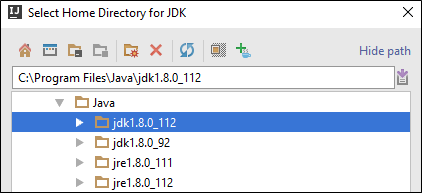
Click on the **green plus sign** in the top left corner of the window and choose **JDK**:

Кликнете върху зеления знак плюс в горния ляв ъгъл на прозореца и изберете JDK:



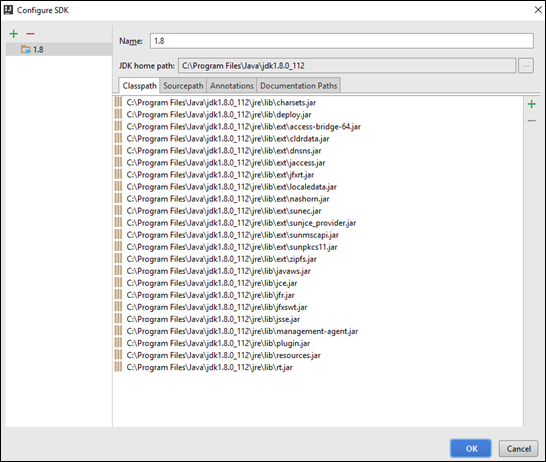
Then **locate your JDK**, it should be in the "**Program Files**" **folder** if you're using **windows**:

След това намерете JDK, той трябва да е в папката "Program Files", ако използвате Windows:



After you click "**OK**", you should see this screen:

След като кликнете върху "OK", трябва да видите този екран:



That is everything, your **JDK is now configured**.

## 1. Creating New Project

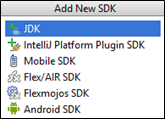
In **IntelliJ** **Idea** **Ultimate**, you should see this when you try to create a new project:

Това е всичко, вашият JDK вече е конфигуриран.  
1. Създаване на нов проект  
 В IntelliJ Idea Ultimate, трябва да видите това, когато се опитате да създадете нов проект:



Click on "**New**". From the drop-down choose **JDK**:

Кликнете върху "Ново". От падащото меню изберете JDK:



Then **locate your JDK**, it should be in the "**Program Files**" **folder** if you're using **windows**:

След това намерете JDK, той трябва да е в папката "Program Files", ако използвате Windows:



Click "**OK**" and you are **ready to create your project**.

# III. Create Spring Project

## 1. Using Spring Initializr

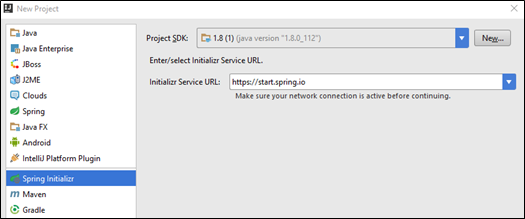
Setting Spring projects without any help is usually a time-consuming thing to do. That is because you need to search the internet for each module that you want to install. This is not always easy and thankfully there are tools that make our life easier. One of the tools is **Spring Initializr**. There is a [web version](http://start.spring.io/), but we are not going to use it. We are going to use the built-in tool in **IntelliJ Idea Ultimate** (not **Community**). In the start page click on "**Create New Project**":

Кликнете върху "OK" и сте готови да създадете своя проект.  
III. Създайте пролетен проект  
1. Използване на Spring Initializer  
Настройването на пролетните проекти без никаква помощ обикновено е отнемащо време. Това е така, защото трябва да търсите в интернет за всеки модул, който искате да инсталирате. Това не винаги е лесно и за щастие има инструменти, които улесняват живота ни. Един от инструментите е Spring Initializr. Има уеб версия, но няма да я използваме. Ще използваме вградения инструмент в IntelliJ Idea Ultimate (не Общността). В началната страница кликнете върху "Създаване на нов проект":



In the newly opened window, on the **left side**, you should see "**Spring Initializr**" as a **project type**:

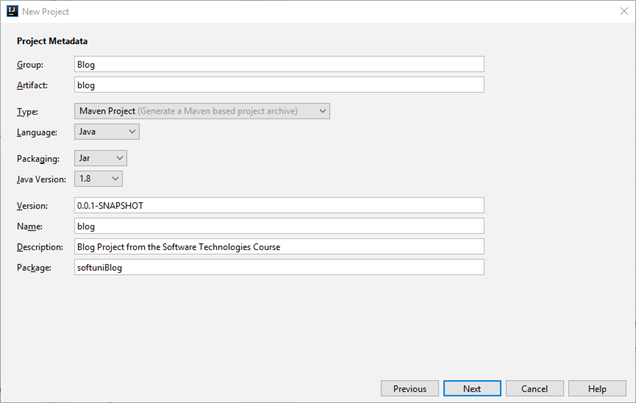
В отворения прозорец от лявата страна трябва да видите "Spring Initializr" като тип проект:



If your "Project SDK" field is empty refer to **chapter 0**.

Click on "**Next**":

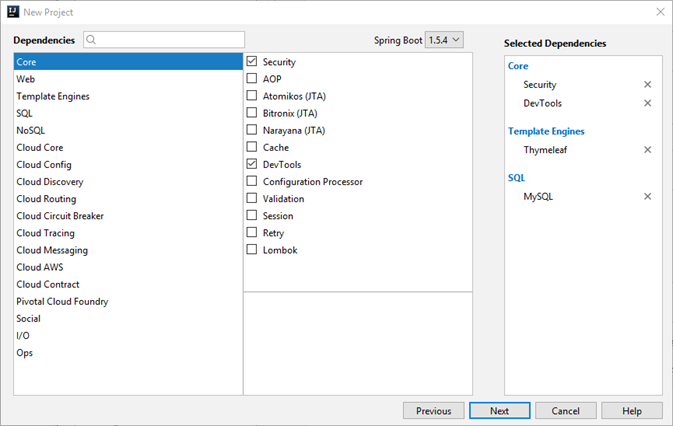
Ако полето "SDK на проекта" е празно, вижте глава 0.  
 Кликнете върху "Напред":



Use the **values** from the **picture** **above**. Now you will see **all of the things** that we **can** **include** in **our project**. We want to include only the following:

* From **Core** choose:
  + **Security**
  + **DevTools**
* From **Template Engines** choose:
  + **Thymeleaf**
* From SQL choose:
  + **MySQL**

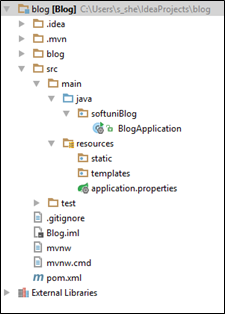
You should have something like this:



Click "**Next**" and on the final page click "**Finish**".

After few seconds, you should have project structure like this one:

Кликнете върху "Напред" и на последната страница кликнете върху "Завършване".  
След няколко секунди трябва да имате структура на проекта като тази:



We will explain the project structure in the next chapter, but first we need to import something.

## 2. Import Additional Dependencies

Now we are going to open the file called "**pom.xml**". It contains **all of the modules** that **we've selected earlier using the Spring Initializr**, but they are not enough. In the file search for this section:

Ще обясним структурата на проекта в следващата глава, но първо трябва да импортираме нещо.  
2. Импортирайте допълнителни зависимости  
Сега ще отворим файла "pom.xml". Той съдържа всички модули, избрани по-рано, с помощта на Spring Initializr, но те не са достатъчни. В търсенето на файл за този раздел:



We want to **include** **additional dependency**, that will help us later. Before we continue, if you see the **following window**:

Искаме да включим и допълнителна зависимост, която ще ни помогне по-късно. Преди да продължим, ако видите следния прозорец:



Click on "**Enable Auto-Import**". It is **really important** and if you miss this step, the **project might not work** as **you would expect**. Now that we've got this out of the way, we can import the **following dependency**:

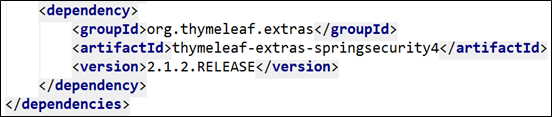
Кликнете върху "Активиране на автоматичното импортиране". Това е наистина важно и ако пропуснете тази стъпка, проектът може да не работи както бихте очаквали. Сега, след като сме излезли от пътя, можем да импортираме следната зависимост:

<**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-data-jpa</**artifactId**>  
 </**dependency**>

<**dependency**>  
 <**groupId**>org.thymeleaf.extras</**groupId**>  
 <**artifactId**>thymeleaf-extras-springsecurity4</**artifactId**>  
 <**version**>2.1.2.RELEASE</**version**>  
 </**dependency**>

**Insert** this at the **bottom of the dependencies section**, and you have this:

Поставете това в долната част на секцията за зависимости и имате следното:

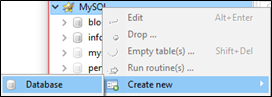


This will give us some **additional commands** that we are going to use in the **following chapters**.

## 3. Create the Database Connection

The last thing we are going to do in this chapter is create the DB connection. For database, we are going to use **MySQL**, the **same** **DB** we've used in the **PHP Blog**. That means that you will need to have **XAMPP** [installed](https://softuni.bg/downloads/svn/soft-tech/May-2016/Software-Technologies-June-2016/06.%20Software-Technologies-XAMPP/06.%20Software-Technologies-XAMPP-Exercises.docx). Now you need to start the **MySQL module** in XAMPP and open [HeidiSQL](http://www.heidisql.com/download.php?download=installer). Again, you should be familiar with **Heidi** from the **PHP Blog**. We should create **new database**. After you are **connected to MySQL** with **Heidi** and you see the **homepage**, you should **right-click** on the connection name:

Това ще ни даде някои допълнителни команди, които ще използваме в следващите глави.  
3. Създайте връзка към базата данни  
Последното нещо, което ще направим в тази глава, е създаването на DB връзка. За база данни ще използваме MySQL, същия DB, който използвахме в блога на PHP. Това означава, че ще трябва да имате инсталиран XAMPP. Сега трябва да стартирате MySQL модула в XAMPP и да отворите HeidiSQL. Отново, трябва да сте запознати с Heidi от блога на PHP. Трябва да създадем нова база данни. След като сте свързани с MySQL с Heidi и виждате началната страница, трябва да кликнете с десния бутон върху името на връзката:

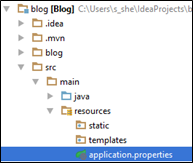


Use the following values: Използвайте следните стойности:



That's it, you've created the database. Now we need to create the connection with our project. Find the file "**application.properties**":

Това е, че сте създали базата данни. Сега трябва да създадем връзка с нашия проект. Намерете файла "application.properties":



The file should be **empty at the moment**. Add the following code:

Файлът трябва да е празен в момента. Добавете следния код:

|  |
| --- |
| *# Database connection with the given database name* **spring.datasource.url** = **jdbc:mysql://localhost:3306/java\_blog?createDatabaseIfNotExist=true&useSSL=false**  **# Username and password**  **spring.datasource.username** = **root**  **spring.datasource.password** =   # Show or not log for each sql query  **spring.jpa.show-sql** = **true**  **# Hibernate ddl auto (create, create-drop, update): with "update" the database**  **# schema will be automatically updated accordingly to java entities found in**  **# the project**  **# Using "create" will delete and recreate the tables every time the project is started**  **spring.jpa.hibernate.ddl-auto** = **update**  **# Naming strategy**  **spring.jpa.hibernate.naming.strategy** = **org.hibernate.cfg.ImprovedNamingStrategy**  **# Allows Hibernate to generate SQL optimized for a particular DBMS**  **spring.jpa.properties.dialect** = **org.hibernate.dialect.MySQL5InnoDBDialect**  **#Turn off Thymeleaf cache**  **spring.thymeleaf.cache** = **false** |

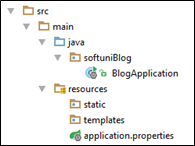
Our connection is done. We will test it later.

Our **project is ready** now, so we can take a look around in the next chapter.

# IV. Reviewing the Project Structure

There is only one folder we're interested at. That is the "**src**" folder. That folder will **contain all of the files** we are **going to create**. Let's take a look:

Нашата връзка е свършена. Ще го тестваме по-късно.  
Нашият проект вече е готов, за да можем да разгледаме в следващата глава.  
IV. Преглед на структурата на проекта  
Има само една папка, от която се интересуваме. Това е папката "src". Тази папка ще съдържа всички файлове, които ще създадем. Нека да разгледаме:



It contains main folder, which is then **separated into 2 different** folders. The first one is the "**java**" folder. This folder contains our **blog package**. **Inside** that **package,** we are going to **create** our **entities**, **controllers**, **configurations**, etc. The other folder is called "**resources**". It contains one file that **creates the connection** with our **database**. There are two other folders named "**static**" and "**templates**". As you've probably have figured it out by now, the "**templates**" folder will contain the **templates** for our [templating engine](https://en.wikipedia.org/wiki/Template_processor). The "**static**" folder will contain the **stylesheets** and **javascripts** **we are going to use** in our project. We will see how are we going to use that in the next chapters.

Тя съдържа основна папка, която след това се разделя на 2 различни папки. Първата е папката "java". Тази папка съдържа пакета ни за блогове. Вътре в този пакет ще създадем нашите структури, контролери, конфигурации и т.н. Другата папка се нарича "ресурси". Той съдържа един файл, който създава връзката с нашата база данни. Има две други папки, наречени "статични" и "шаблони". Както вероятно сте разбрали досега, папката "шаблони" ще съдържа шаблоните за нашия шаблонен двигател. Папката "static" ще съдържа стилове и javascript, които ще използваме в нашия проект. Ще видим как ще го използваме в следващите глави.

# V. Spring Security

At the moment, you **cannot use your project**. Why? Because we've **imported the dependency for Spring**, that gives us the **authentication module**, but we haven’t configured it, yet. To do that we will create User **entity** using Hibernate. Then we are going to tell **Spring Security** what to use from our entity. Finally, we will setup the **configuration** that will **allow us to login**. This **module** will **give us** the **user authorization** as well.

V. Пролетната сигурност  
В момента не можете да използвате вашия проект. Защо? Тъй като сме импортирали зависимостта за Пролетта, това ни дава модула за удостоверяване, но все още не сме го конфигурирали. За да направим това, ще създадем потребителски обект, използвайки хибернация. Тогава ще кажем на Spring Security какво да използваме от нашата организация. И накрая, ще настроим конфигурацията, която ще ни позволи да влезем. Този модул ще ни даде и разрешението на потребителя.

## 1. Creating the User Entity

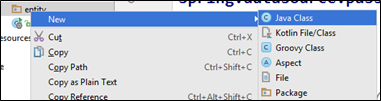
In our "java/softuniBlog" package create a new package called "entity":

1. Създаване на потребителското лице  
В пакета "java / softuniBlog" създайте нов пакет, наречен "entity":



This package will contain all of our entities – **users**, **articles**, **roles**, **tags**, **categories**, etc. We will start by **creating new Java** **class** called "User":

Този пакет ще съдържа всички наши обекти - потребители, статии, роли, тагове, категории и т.н. Ще започнем с създаването на нов клас Java наречен "Потребител":



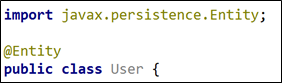
By default, it should look something similar to this:

По подразбиране трябва да изглежда нещо подобно на това:



Let's start with the first annotation:

Нека да започнем с първата анотация:



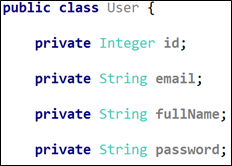
This [annotation](https://en.wikipedia.org/wiki/Java_annotation) means that the User class will become [entity](http://stackoverflow.com/questions/2550197/whats-the-difference-between-entity-and-class) that will get **saved into our database**. The next annotation is going to **define the table name** in our **database**:

Тази анотация означава, че потребителският клас ще стане обект, който ще бъде запазен в нашата база данни. Следващата анотация ще дефинира името на таблицата в нашата база данни:



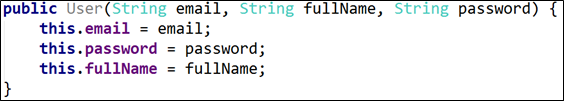
This looks **very similar** to the **Doctrine** entities that you've created for the **PHP** blog. Create the following private fields:

Това изглежда много подобно на ученията, които сте създали за блога на PHP. Създайте следните частни полета:



That is the information that we will keep in the database for our user. **ID**, which will be the **unique key**, **email**, **name** and **password**. The next thing that we are going to **create** is **constructor**, which should **help** us **with** the **user creation** later on:

Това е информацията, която ще съхраняваме в базата данни на нашия потребител. Идентификационен номер, който ще бъде уникален ключ, имейл, име и парола. Следващото нещо, което ще създадем, е конструкторът, който по-късно трябва да ни помогне при създаването на потребителя:



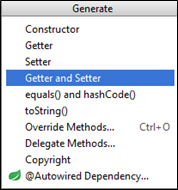
**Spring Security** will need **second constructor** in order to provide us with useful features. **It** **should be empty**:

Spring Security ще има нужда от втори конструктор, за да ни предостави полезни функции. То трябва да е празно:



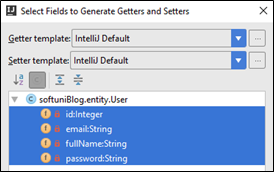
Now we need [getters and setters](http://java.about.com/od/workingwithobjects/a/accessormutator.htm). You should already be familiar with them. If you are curious why are we doing that, you can read more [here](https://www.tutorialspoint.com/java/java_encapsulation.htm). There is a **simple way to create them** in **IntelliJ Idea**. If you press "[Alt + Insert]", you should see that context menu:

Сега имаме нужда от купувачи. Вече трябва да сте запознати с тях. Ако сте любопитни защо правим това, можете да прочетете повече тук. Има един лесен начин да ги създадете в IntelliJ Idea. Ако натиснете "[Alt + Insert]", трябва да видите това контекстно меню:



Choosing the "**Getter and Setter**" option will **open new window**. You should select **all private fields** from there:

Избирането на опцията "Getter and Setter" ще отвори нов прозорец. Трябва да изберете всички частни полета оттук:



When you **click** "**OK**", you should **receive this code**:

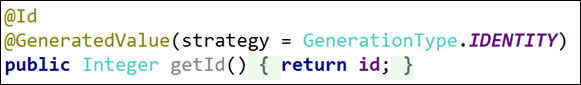
Когато кликнете върху "OK", трябва да получите този код:



It might be **formatted in a different way**, but the **result should be the same**.

Now we need to create our annotations. Let's start with the getId() getter. We want the id to be **generated automatically**. Place the following annotations:

Тя може да бъде форматирана по различен начин, но резултатът трябва да бъде същият.  
Сега трябва да създадем нашите пояснения. Да започнем с getId () getter. Искаме ID да се генерира автоматично. Поставете следните пояснения:



The "@Id" annotation tells [Hibernate](http://hibernate.org/orm/) that **this field will be the primary key** for our **database**. The second annotation makes the **field generated automatically**, without us doing anything. The next annotations are really similar. Анотацията "@Id" казва, че Hibernate е това поле, което ще бъде първичният ключ за нашата база данни. Втората анотация прави полето генерирано автоматично, без да правим нищо. Следващите анотации са наистина подобни.

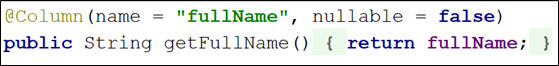
**Email**:



**Password**:



**Name**:



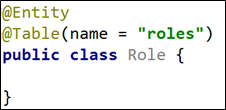
In all three of them, we are **defining** the **column name** and we are **making them non-nullable**. That means they can't contain null value. For the **password** field, we are limiting the **max length to 60 symbols**. Finally, we are telling **Hibernate** that the **Email** **should** **be** **unique** for every user.

Our user is **almost modelled**. But we need to **give him a role**. In order to do that, we need to **create new entity**.

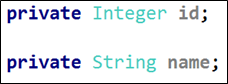
## 2. Creating the Role Entity

Create new class in the "entity" package that will be called "Role" and should have the **following annotations**:

Във всичките три от тях дефинираме името на колоната и ги правим ненулируеми. Това означава, че те не могат да съдържат нулева стойност. За полето за парола ограничаваме максималната дължина до 60 символа. И накрая, казваме на Hibernate, че имейлът трябва да е уникален за всеки потребител.  
Нашият потребител е почти моделиран. Но трябва да му дадем роля. За да направим това, трябва да създадем нов обект.  
2. Създаване на Entity Роля  
Създайте нов клас в пакета "entity", който ще се нарича "Роля" и трябва да има следните пояснения:

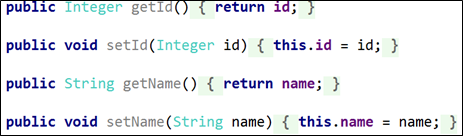


The next thing is the private fields: Следващото нещо е частните полета:



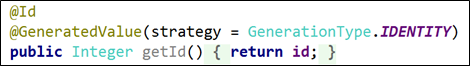
The name will be in the following format "ROLE\_\*". Then, we have to create the getters and setters:

Името ще бъде в следния формат "ROLE\_ \*". След това трябва да създадем получателите и създателите:

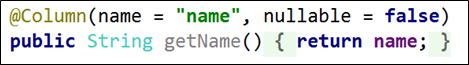


Now we need the **annotations** for our fields. As we did with the **user**, the id should be **auto-generated**:

Сега имаме нужда от поясненията за нашите полета. Както направихме с потребителя, идентификационният номер трябва да се генерира автоматично:



And the **name shouldn't** be null: И името не трябва да бъде нищожно:



**This** is the Role **entity**. Now we need to create the relationship between the User and the Role.

## 3. Creating the Role-User Relation

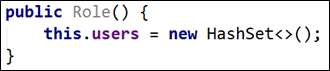
Because we are in the Role entity, let's start the relation from there. Our relation will be of type Many-to-Many. That means that **many users can have many roles**. In order to do that relation, we need to **create a collection of users** in our Role entity. That field will **contain only unique users** and will tell us **which users are having the current role**. It should look like that:

Това е ролевата единица. Сега трябва да създадем връзката между потребителя и ролята.  
3. Създаване на връзката роля-потребител  
Тъй като сме в ролята на Роля, нека започнем връзката оттам. Нашата връзка ще бъде от тип Много до много. Това означава, че много потребители могат да имат много роли. За да направим тази връзка, трябва да създадем колекция от потребители в нашата роля на Ролята. Това поле ще съдържа само уникални потребители и ще ни каже кои потребители имат настоящата роля. Трябва да изглежда така:



To use it, **similar to every other collection**, we need to **initialize** it using **constructor**:

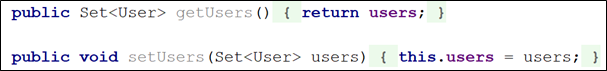
За да го използваме, подобно на всяка друга колекция, трябва да я инициализираме с конструктор:



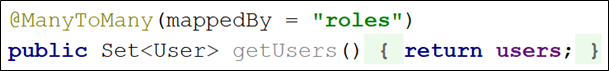
You can read more about the HashSet [here](https://docs.oracle.com/javase/7/docs/api/java/util/HashSet.html).

We also need the **getter and setter** for the field:

Можете да прочетете повече за HashSet тук.  
Нуждаем се също така от getter и setter за полето:



And the annotation will be this: Анотацията ще бъде следната:



This means that in the User entity we need to create **private field** called "**roles**" that will create the relation.

We should jump to the User entity now.

## 4. Creating the User-Role Relation

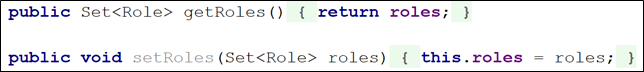
As we've said, we should create private field in the User entity:

Това означава, че в потребителското лице трябва да създадем частно поле, наречено "роли", което ще създаде връзката.  
Сега трябва да преминем към потребителската единица.  
4. Създаване на връзка между потребител и роли  
Както вече казахме, трябва да създадем частно поле в потребителската единица:



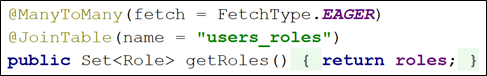
This will keep the **unique roles each user** **has**. Create the **getter** and **setter** now:

Това ще запази уникалните роли, които всеки потребител има. Създайте поглъщащото устройство и устройството за настройка сега:



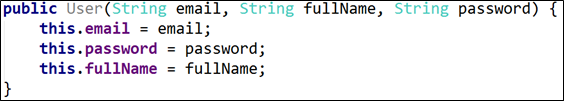
Now you need to **add** the following **annotations**:

Сега трябва да добавите следните пояснения:



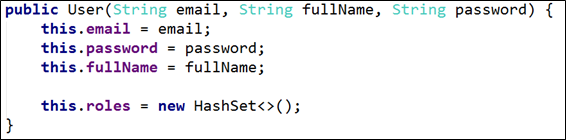
There is something new we are using here. In our "@ManyToMany" annotation we are telling that our "fetch" will be of type "EAGER". It basically means that we want the **roles to be loaded** **together** with the **user**. **Usually that will happen** when we want to **use the roles**, but that's an [advanced topic](https://howtoprogramwithjava.com/hibernate-eager-vs-lazy-fetch-type/). The other annotation will create the **joining table** for our relation and will **name it** "users\_roles". Let's take a look at our constructor now:

Има нещо ново, което използваме тук. В нашето "@ManyToMany" анотация ние казваме, че нашето "донасяне" ще бъде от типа "EAGER". Всъщност това означава, че искаме ролите да се зареждат заедно с потребителя. Обикновено това ще се случи, когато искаме да използваме ролите, но това е напреднала тема. Другата анотация ще създаде свързващата таблица за връзката ни и ще я нарече "users\_roles". Нека сега да разгледаме нашия конструктор:



We are not **assigning default role** when we **create new user**. That's why we need to change the constructor like that:

Не създаваме роля по подразбиране, когато създаваме нов потребител. Ето защо трябва да променим конструктора така:

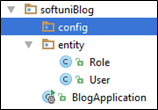


Here we are **initializing our collection**, saving us problems later on.

## 3. Creating UserDetails Implementation

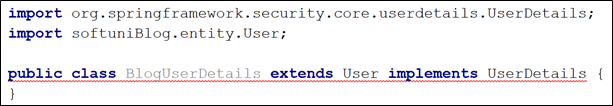
In order to use the **built-in functionality** from **Spring Security** we need to create a **new class**, that will make sure that we are **creating the users** using **the right way**. In the "softuniBlog" package, create a new package called "config":

Тук инициализираме нашата колекция, като ни спести по-късно проблемите.  
5. Създаване на изпълнение на UserDetails  
За да използваме вградената функционалност от Spring Security, трябва да създадем нов клас, който да гарантира, че създаваме потребителите по правилния начин. В пакета "softuniBlog" създайте нов пакет, наречен "config":



Inside of it create a **new class** called "BlogUserDetails". Then change it like that:

Вътре в него създайте нов клас, наречен "BlogUserDetails". След това го променете по следния начин:



**Don't worry** that everything goes **red**, we will take care of that. First, add the following code inside the class:

Не се притеснявайте, че всичко става червено, ще се погрижим за това. Първо, добавете следния код в класа:



We are forced to **override** some of the methods in the "UserDetails" interface. That is not all of them, but before we continue, create two new **private fields** that will keep our **current user and his roles**:

Ние сме принудени да заменим някои от методите в интерфейса "UserDetails". Това не е всичко, но преди да продължим, създайте две нови частни полета, които ще запазят текущия ни потребител и ролите му:



The User is our **entity type** User and the **roles** is a simple **list** collection. And now we need to **create a constructor** for this class. It should look like that:

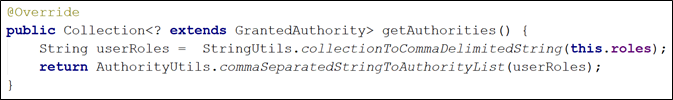
Потребителят е нашият тип потребителски потребител и ролите са проста колекция от списъци. И сега трябва да създадем конструктор за този клас. Трябва да изглежда така:



As you can see we're setting our roles and user fields using the parameters we are taking in the constructor. However, we are doing something else as well. We are using some sort of method called "super()". This is way **more complicated** to explain than it looks so we'll leave it for your future courses (**OOP** **concept** called [inheritance](https://www.tutorialspoint.com/java/java_inheritance.htm)). For now, you can imagine that it **assigns** the **user** **email**, **name** and **password** to our class, using the **constructor** of our [base class](https://www.tutorialspoint.com/java/java_inheritance.htm).

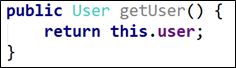
Now we need to override one more method:

Както виждате, настройваме нашите роли и потребителски полета чрез параметрите, които използваме в конструктора. Но правим и нещо друго. Използваме някакъв метод, наречен "супер ()". Това е по-сложно обяснение, отколкото изглежда, така че ще го оставим за вашите бъдещи курсове (понятието OOP, наречено наследство). Засега можете да си представите, че присвоява потребителския имейл, име и парола на нашия клас, използвайки конструктора на базовия ни клас.  
Сега трябва да заличим още един метод:



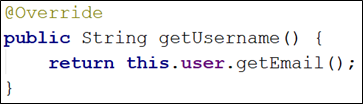
This will get our **roles** (that we currently keep as strings) and **join** **them** into one string (we use the StringUtils class from org.springframework.util). Then it will return collection of [authorities](http://docs.spring.io/spring-security/site/docs/3.0.x/reference/authz-arch.html). The **authorities** in **Spring** are the things we call "**roles**" or "**permissions**". With that our class is almost ready. The only thing left is to create a method that will **return** our **current user**:

Това ще получи нашите роли (които понастоящем запазваме като струни) и ги присъединяваме към един низ (използваме класа StringUtils от org.springframework.util). Тогава тя ще върне събирането на властите. Органите на пролетта са нещата, които наричаме "роли" или "разрешения". С това класът ни е почти готов. Единственото останало нещо е да създадем метод, който да върне настоящия ни потребител:



And **override** the last method, which we need to implement:

И да замени последния метод, който трябва да приложим:

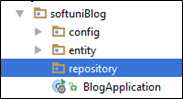


That is all, now we need to **find a way** to get our **users** and **roles** from the **database**.

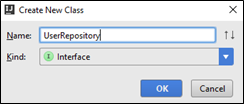
## 6. Creating User Repository

Now, we are not exactly finding a way to get the users. There is a way called "**Repositories**". You can imagine that the [repository](http://docs.spring.io/spring-data/data-commons/docs/1.6.1.RELEASE/reference/html/repositories.html) is our **local access** to the **database**. Using **methods** in our **repositories**, we will **get the entities** from our **database** and **use them locally**. Create a new package called "repository":

Това е всичко, сега трябва да намерим начин да получим нашите потребители и роли от базата данни.  
6. Създаване на потребителско хранилище  
Сега не намираме точно начин да привлечем потребителите. Има начин, наречен "хранилища". Можете да си представите, че хранилището е нашият местен достъп до базата данни. Използвайки методи в нашите хранилища, ние ще получим обектите от нашата база данни и ще ги използваме локално. Създайте нов пакет, наречен "хранилище":



Now we will create UserRepository: Сега ще създадем UserRepository:

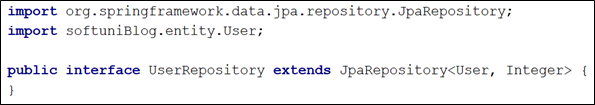


The important thing is that it will **not be a class**. It will be an [interface](http://tutorials.jenkov.com/java/interfaces.html). The interface is a special type, which **can't contain functional methods**. It can **only** **declare them**. You should have this:

Важното е, че няма да е клас. Това ще бъде интерфейс. Интерфейсът е специален тип, който не може да съдържа функционални методи. Тя може само да ги обяви. Трябва да имате следното:



We should quickly change it to: Трябва бързо да го променим на:



This will give us **some methods** that we are going to use later on in our blog, but for now we want to **create the following method** in our **repository**:

Това ще ни даде някои методи, които ще използваме по-късно в нашия блог, но за сега искаме да създадем следния метод в нашето хранилище:

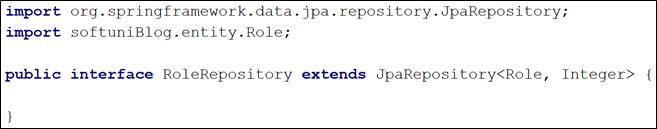


As you can see, this **method is different**. It doesn't have body. Using magic (and [reflection](http://www.javatpoint.com/java-reflection)) **Spring** will find a **user** **by** **his email**. It will use **reflection** to get the **type** of the **repository**, which is our entity "User", then it will get the **table** **name** from the **annotation**. After that, it will split the name of our method into different parts. The first part is "findBy", which means that it will send [SELECT](http://www.w3schools.com/sql/sql_select.asp) **query** to our **database**. Then it will take the **second part** which is "Email" in this case and it will understand that we want to get **user** by a **given** **email** address. The **generated query** will look like this "SELECT id, email, full\_name, password FROM users WHERE email={parameter}". Anyway, let's move on.

## 7. Creating Role Repository

Create a new interface called RoleRepository, that will be the **repository** for our **roles**:

Както можете да видите, този метод е различен. Тя няма тяло. Чрез магия (и отражение) Пролетта ще намери потребител по имейл. Тя ще използва размисъл, за да получи типа хранилище, което е нашето лице "Потребител", след което ще получи името на таблицата от анотацията. След това тя ще раздели името на нашия метод на различни части. Първата част е "findBy", което означава, че ще изпрати заявка SELECT към нашата база данни. След това ще вземе втората част, която е "Email" в този случай и ще разбере, че искаме да получим потребител чрез даден имейл адрес. Генерираната заявка ще изглежда по следния начин: "SELECT id, email, full\_name, парола FROM users WHERE email = {parameter}". Както и да е, да продължим.  
7. Създаване на хранилище за роли  
Създайте нов интерфейс, наречен RoleRepository, който ще бъде хранилището на нашите роли:



Make sure that you've **imported** the right **Role**, because **otherwise** **it won't work**! **Declare a method** like this one:

Уверете се, че сте импортирали правилната роля, защото в противен случай няма да работи! Декларирайте метод като този:



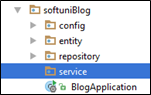
This will get us the **role** with **given name**. It is almost the same as the method in the UserRepository, but the **criteria** and **return types** are **different**.

We are ready with our repositories for now.

## 8. Creating User Service

The next thing we need to implements is the so called "userService". It is used to get **user from the database** and transform it to **Spring Security** User. Create a new package called "service":

Това ще ни даде роля с дадено име. Това е почти същото като метода в UserRepository, но критериите и типовете връщане са различни.  
Засега сме готови с нашите хранилища.  
8. Създаване на потребителско обслужване  
Следващото нещо, което трябва да реализираме, е т.нар. "UserService". Използва се за получаване на потребител от базата данни и трансформирането му в Spring Security User. Създайте нов пакет, наречен "услуга":



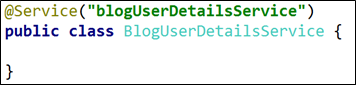
Create a new class called BlogUserDetailsService:

Създайте нов клас, наречен Blog UserDetailsService:



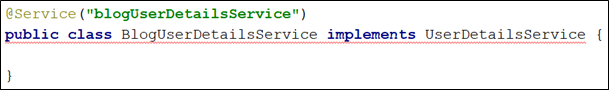
In order to tell **Spring** that **this will be a service**, we need to use the following **annotation**:

За да кажем на пролетта, че това ще бъде услуга, трябва да използваме следното пояснение:



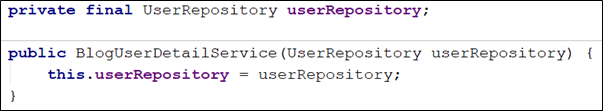
This will **give** our **service** a **name**. Now, we need to change the class like that:

Това ще даде име на нашата услуга. Сега трябва да променим класа така:



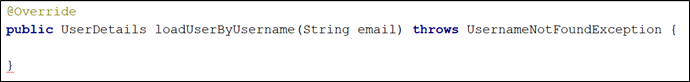
Again, everything **becomes red**, but that's nothing to worry about. We will start by creating a **private** **field** and **constructor** to initialize it:

Отново всичко става червено, но това не е нищо, за което да се притеснявате. Ще започнем с създаването на частно поле и конструктор, за да го инициализираме:



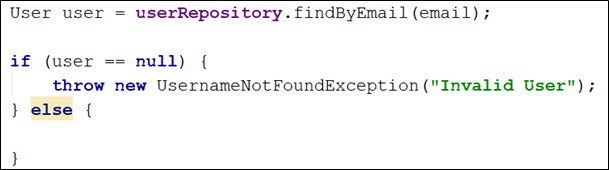
This private field has the "final" keyword, which means that we will **not be able to change it** after **initialization**. Now we need to **override** one of the **base class methods**:

Това частно поле има "крайната" ключова дума, което означава, че няма да сме в състояние да я променим след инициализация. Сега трябва да заличим един от методите на базовия клас:



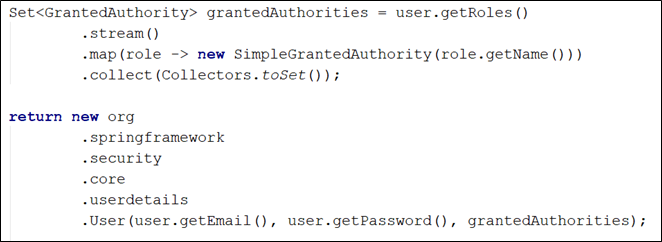
The idea behind this method is to get our **user** and make it object of type UserDetails. This will give us the **ability** to **login** and do other things with our users. We need to **get a user** by a **given email**. If the **user** **does** **not** **exist**, we will **throw exception**:

Идеята зад този метод е да привлечем потребителя и да го направим обект на тип UserDetails. Това ще ни даде възможност да влезем и да правим други неща с нашите потребители. Трябва да получим потребител чрез даден имейл. Ако потребителят не съществува, ще направим изключение:



The case, where the user exist is more interesting:

Случаят, в който потребителят съществува, е по-интересен:



Here we get all of the **user roles** and **create a collection of authorities**. Then we create a new **Spring Security** User with the given **email**, **password** and **authorities**. This is **everything** for **our** **service**, but we are **not** **done**, yet.

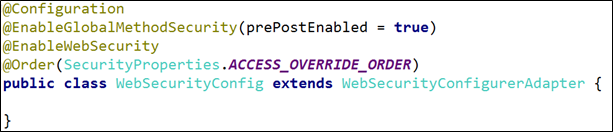
## 9. Creating Web Security Configurer Adapter

We've got to the point, where we need to configure our **Security** module. We should start by creating a new class called "WebSecurityConfig" in the "config" package:

Тук получаваме всички потребителски роли и създаваме колекция от органи. След това създаваме нов потребител на Пролетната сигурност с даден имейл, парола и пълномощия. Това е всичко за нашата услуга, но все още не сме свършили.  
9. Създаване на адаптер за конфигуриране на уеб сигурността  
Ние стигнахме до там, където трябва да конфигурираме нашия модул за сигурност. Трябва да започнем с създаването на нов клас, наречен "WebSecurityConfig" в пакета "config":



Now it will get really messy, really quick: Сега ще се получи наистина разхвърлян, наистина бърз:



Most of those annotations are working together. Links for the different annotations:

* [@Configuration](http://docs.spring.io/spring-framework/docs/4.0.4.RELEASE/javadoc-api/org/springframework/context/annotation/Configuration.html)
* [@EnableGlobalMethodSecurity](http://docs.spring.io/spring-security/site/docs/4.0.4.RELEASE/apidocs/org/springframework/security/config/annotation/method/configuration/EnableGlobalMethodSecurity.html)
* [@EnableWebSecurity](http://docs.spring.io/autorepo/docs/spring-security/4.1.1.RELEASE/apidocs/org/springframework/security/config/annotation/web/configuration/EnableWebSecurity.html)
* [@Order](http://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/core/annotation/Order.html)

Overall, the configuration annotation will tell **Spring** that this is a **configuration class**, and the **rest** **of** **the** **annotations** will **set different settings** for it.

Now, we need to create **private** **field** that will keep our **service**:

Повечето от тези пояснения работят заедно. Връзки за различните пояснения:  
• @Configuration  
• @EnableGlobalMethodSecurity  
• @EnableWebSecurity  
• @Поръчка  
Като цяло, анотацията за конфигурация ще покаже на Spring, че това е конфигурационен клас, а останалите анотации ще определят различни настройки за него.  
Сега трябва да създадем частно поле, което да поддържа услугата ни:



We will need to create the following annotation for it: Ще трябва да създадем следното пояснение за него:



Using that **annotation**, we are telling our class to **initialize the field automatically**. The next thing that we want **Spring** to do **automatically** is to **change the default password encoder** to BCrypt:

Като използваме тази анотация, казваме на нашия клас да инициализира полето автоматично. Следващото нещо, което искаме пролетта да направи автоматично, е да смени стандартния енкодер на паролата на BCrypt:



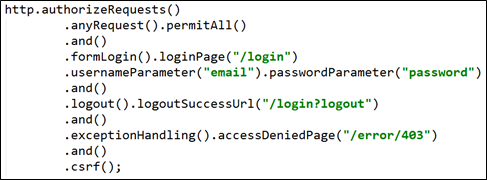
Here we are setting the default userDetailsService to use our **field** and we are setting the passwordEncoder to a **more secure** one.

It's time to create the method that will **take care of the access control**:

Тук настройваме стандартния потребителDetailsService да използва нашето поле и настройваме паролата "Енкодер" на по-сигурна.  
Време е да създадем метода, който ще се погрижи за контрола на достъпа:



This method is going to define **what permissions are needed** to **access** our **blog**. Write the following code: Този метод ще определи какви разрешения са необходими за достъп до нашия блог. Напишете следния код:



This code tells the authentication module, that **every page** can be **accessed** by **every user**. Then it tells us that the **login request** should be expected at the "/login" **route**. The parameter for login will be "email" and the parameter for password will be "password". The **logout** will lead to "/login?logout" and if there is **any error** with the **permissions**, we should **receive view** that tells us that **we don’t have access**.

It’s the beginning of the end guys…

Този код указва на модула за удостоверяване, че всяка страница може да бъде достъпвана от всеки потребител. Тогава ни казва, че заявката за вход трябва да се очаква на маршрута "/ login". Параметърът за вход ще бъде "имейл", а параметърът за парола ще бъде "парола". Изходът ще доведе до "/ login? Logout" и ако има някаква грешка с разрешенията, трябва да получим изглед, който ни показва, че нямаме достъп.  
Това е началото на края на момчетата ...

## 10. Creating Base Layout

Before we give you the **layout code**, let's talk about **layouts**, **templating engines** and more specifically **Thymeleaf**. The idea behind them is to reuse code. Now we want to create the **base layout**, which we will **reuse** for the other pages of our **blog**. **Inside** of that **layout** we will **import** the css and js files. We are going to split it in few different sections. The first section is our "**header**", which contains the **navigation bar** and it will have at least **three different parts**. The first part is this:

10. Създаване на основно оформление  
Преди да ви дадем кода на оформлението, нека да говорим за оформления, шаблони за двигатели и по-специално Thymeleaf. Идеята зад тях е да използват повторно кода. Сега искаме да създадем основното оформление, което ще използваме отново за другите страници на нашия блог. Вътре на това оформление ще импортираме файловете css и js. Ще го разделим на няколко различни раздела. Първият раздел е нашият "header", който съдържа навигационната лента и ще има най-малко три различни части. Първата част е следната:



**Everyone** should see this when they **open** **the** **site**. They will only have the option to **login** and **register**. Once they **login**, they will see **one** of the **following**:

Всеки трябва да види това, когато отвори сайта. Те ще имат само възможност да влязат и да се регистрират. След като влязат в профила си, ще видят едно от следните неща:



Or

Вмъкване на изображение...

The other two parts as you can see for **logged users**. Using **Thymeleaf**, we need to check if whoever is opening our pages is logged in or not. **Take a look** at this code (**don't write** it anywhere):

Другите две части, както можете да видите за регистрирани потребители. Използвайки Thymeleaf, трябва да проверим дали всеки, който отваря страниците ни, е влязъл в профила си или не. Разгледайте този код (не го пишете никъде):



For every link in our navigation menu we are using "sec:authorize". This is coming from **Thymeleaf Security** and gives us the ability to **check** **if** **someone** is **logged in**, **has a specific role** or is just a **guest**.

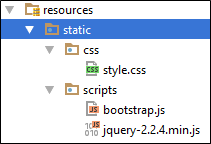
Another section is the "**footer**". It represents this:

За всяка връзка в нашето меню за навигация използваме "sec: authorize". Това идва от Thymeleaf Security и ни дава възможност да проверим дали някой е влязъл в профила си, има определена роля или е просто гост.  
Друг раздел е "долната част". Това представлява:



This is also used on **every single of our pages**. We will also **need scripts**, to **use** **bootstrap**. The final section is called "**main**" and it is unique for every page. It contains the content for any given page. However, three of four **sections can be reused**. First, we need to import our design. In the "resources/static" folder import the scripts and css folders we gave you:

Това се използва и на всеки от нашите страници. Ще ни трябва скриптове, за да използваме bootstrap. Последната секция се нарича "основна" и е уникална за всяка страница. Тя съдържа съдържанието за дадена страница. Въпреки това, три от четирите секции могат да бъдат използвани повторно. Първо, трябва да импортираме нашия дизайн. В папката "resources / static" импортирайте скриптовете и css папките, които ви дадохме:



Now, we will create our base layout. Create a **new HTML file** in the templates package called "base-layout" and leave it there for now:

Сега ще създадем базовото ни разположение. Създайте нов HTML файл в пакета шаблони, наречен "базово оформление", и го оставяйте за момента:



Now, create a **new directory** in the templates package called "fragments". It will contain the fragments(sections), we've talked earlier about:

Сега създайте нова директория в пакета шаблони, наречен "фрагменти". Той ще съдържа фрагментите (раздели), за които говорихме по-рано за:

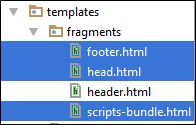


Inside of it, create a **new HTML file** called "header": Вътре в него създайте нов HTML файл, наречен "header":



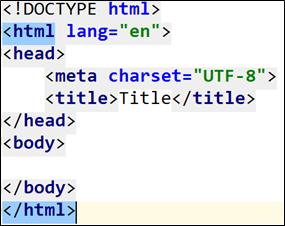
Create another 3 html files called "**footer**", "**head**" and "**scripts-bundle**":

Създайте още 3 html файла, наречени "долен", "глави" и "скриптове-пакет":



Each of the HTMLs we've created should look like this at the moment:

Всеки от създадените от нас HTML файлове трябва да изглежда така:



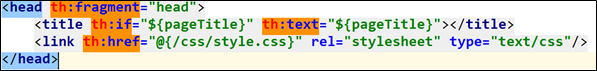
Let's start editing our fragments. The first one is the "head.html". **Delete the existing html** and write the following:

Нека да започнем да редактираме нашите фрагменти. Първата е "head.html". Изтрийте съществуващия html и напишете следното:

<**head th:fragment="head"**>  
 <**title th:if="${pageTitle}" th:text="${pageTitle}"**></**title**>  
 <**link th:href="@{/css/style.css}" rel="stylesheet" type="text/css"**/>  
 </**head**>

Don't be worried if the code is marked in red, or looks like this:

Не се притеснявайте, ако кодът е отбелязан в червено или изглежда така:



The code inspection tells us that **our code is not valid HTML** and that is correct. However, when we import it in our layout, Thymeleaf will validate it. This code **imports** our style.css file and gives us the ability to **dynamically** **change the title** of our blog. Next on the list is the "header.html":

Кодовата проверка ни показва, че кодът ни не е валиден HTML и това е правилно. Въпреки това, когато го внасяме в нашето оформление, Thymeleaf ще го валидира. Този код внася нашия файл style.css и ни дава възможност да променим динамично заглавието на нашия блог. Следващото в списъка е "header.html":

|  |
| --- |
| <**header th:fragment="header"**>  <**div class="navbar navbar-default navbar-static-top" role="navigation"**>  <**div class="container"**>  <**div class="navbar-header"**>  <**a th:href="@{/}" class="navbar-brand"**>SOFTUNI BLOG</**a**>    <**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"**>  <**li sec:authorize="isAuthenticated()"**>  <**a th:href="@{/profile}"**>  My Profile  </**a**>  </**li**>  <**li sec:authorize="isAuthenticated()"**>  <**a th:href="@{/logout}"**>  Logout  </**a**>  </**li**>    <**li sec:authorize="isAnonymous()"**>  <**a th:href="@{/register}"**>  REGISTER  </**a**>  </**li**>  <**li sec:authorize="isAnonymous()"**>  <**a th:href="@{/login}"**>  LOGIN  </**a**>  </**li**>  </**ul**>  </**div**>  </**div**>  </**div**>  </**header**> |

It is a much larger piece of code that represent **our navigation bar**. We will explain most of the Thymeleaf code later on. The only thing that is important at the moment is the "th:href" tag. It is a Thymeleaf hyperlink, that uses the Thymeleaf syntax to redirect us to the other pages, instead of html. Next on the list is the "footer.html" file:

Това е много по-голямо парче код, което представлява нашата навигационна лента. По-късно ще обясним по-голямата част от кода на Thymeleaf. Единственото важно нещо в момента е тагът "th: href". Това е хипервръзка Thymeleaf, която използва синтаксиса Thymeleaf, за да ни пренасочи към другите страници, вместо към html. Следва в списъка файлът "footer.html":

|  |
| --- |
| <**footer th:fragment="footer"**>  <**div class="container modal-footer"**>  <**p**>**&copy;** 2016 - Software University Foundation</**p**>  </**div**>  </**footer**> |

It is a really simple HTML, defining the footer of our blog. Finally let's edit the "scripts-bundle.html":

Това е наистина прост HTML, определящ долния колонтитул на нашия блог. Накрая нека редактираме "scripts-bundle.html":

|  |
| --- |
| <**script** [**th:src="@{/scripts/jquery-2.2.4.min.js}"**></**script**](mailto:th:src="@{/scripts/jquery-2.2.4.min.js}"></script)>  <**script** [**th:src="@{/scripts/bootstrap.js}"**></**script**](mailto:th:src="@{/scripts/bootstrap.js}"></script)> |

Its job is to import the 2 **JavaScript** files we are going to use. As you can see, all of our fragments have a specific role in our design, but let's combine them together in our "base-layout.html":

Неговата задача е да импортираме 2 JavaScript файла, които ще използваме. Както можете да видите, всички наши фрагменти имат специфична роля в нашия дизайн, но нека ги комбинираме в "base-layout.html":

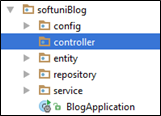
|  |
| --- |
| <!DOCTYPE **HTML**>  <**html xmlns="http://www.w3.org/1999/xhtml" xmlns:th="http://www.thymeleaf.org" xmlns:sec="http://www.thymeleaf.org/thymeleaf-extras-springsecurity4"**>   <**head th:include="fragments/head" th:with="pageTitle='SoftUni Blog'"**></**head**>   <**body**>   <**header th:include="fragments/header"**></**header**>   <**main th:include="${view}"**></**main**>   <**footer th:include="fragments/footer"**></**footer**>   <**span th:include="fragments/scripts-bundle"**></**span**>   </**body**>  </**html**> |

This will be the only complete and valid HTML file. However, it is not just a normal HTML file. It imports Thymeleaf and Thymeleaf Security, which will be of great use. As you can see our **head** **tag** uses something called "th:include". This will replace our current <head> tag with the html file called "head" from our **fragments** **folder**. Then it will give our blog the title "**SoftUni Blog**". In our body tag, we have exactly 4 lines of code. The header **tag** that will be **replaced** **by** the "fragments/header" file. The footer and the span tags that will be **replaced** by **our** **other** **fragments**. There is something strange. Our <main> tag includes some file called "${view}" that we've never created. Not exactly. This is a **variable** in Thymeleaf that we need to **send to our view**. The **variable should be called** "view" and it should **contain the path** to the **html file** that **we** **want** to **load** here. We will do that using our **controllers**.

## 11. Creating User Controller

Create a new package called "controller":

Това ще бъде единственият пълен и валиден HTML файл. Това обаче не е само нормален HTML файл. Той внася Thymeleaf и Thymeleaf Security, които ще бъдат от голяма полза. Както виждате, главният ни маркер използва нещо, наречено "th: include". Това ще замени нашия текущ маркер <head> с html файла, наречен "head" от папката с фрагменти. Тогава той ще даде нашия блог заглавието "SoftUni Blog". В нашия маркер на тялото имаме точно 4 реда код. Знакът на заглавката ще бъде заменен от файла "фрагменти / заглавка". Буквите на долната и долната част, които ще бъдат заменени от другите ни фрагменти. Има нещо странно. Нашият <main> маркер включва файл, наречен "$ {view}", който никога не сме създавали. Не точно. Това е променлива в Тилемеаф, която трябва да изпратим на нашето мнение. Променливата трябва да се нарича "изглед" и тя трябва да съдържа пътя към HTML файла, който искаме да заредим тук. Ще направим това с помощта на нашите контролери.  
11. Създаване на потребителски контролер  
Създайте нов пакет, наречен "контролер":



Now create new class called UserController: Сега създайте нов клас, наречен UserController:



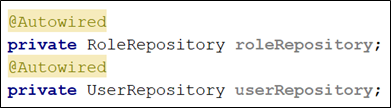
This class will **register new users**, **login** the old ones, **show us the profile page**, etc. That's why we will add the following annotation:

Този клас ще регистрира нови потребители, ще влезе в старите, ще ни покаже страницата на потребителския профил и т.н. Затова ще добавим следната анотация:



That way we are telling **Spring** that this class can **define routes** and that will **take care of actions** related with **our entities**. First let's create private fields for our repositories:

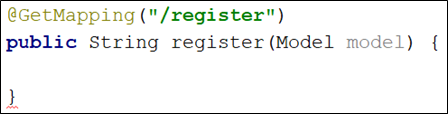
По този начин казваме на пролетта, че този клас може да определи маршрути и ще се погрижи за действия, свързани с нашите субекти. Първо да създадем частни полета за нашите хранилища:



We are using the "@Autowired" annotation again, to tell **Spring** to **initialize those fields**.

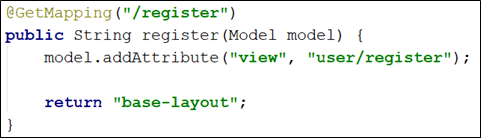
First, we need to be able to create users. Create the following method:

Използваме отново "@Autowired" пояснението, за да кажем на Spring да инициализира тези полета.  
Първо, трябва да можем да създаваме потребители. Създайте следния метод:



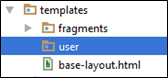
We are using the "@GetMapping" annotation. This annotation defines that the type of [request](http://www.w3schools.com/TAGS/ref_httpmethods.asp) we are going to **process in our method** is "**GET**". That means that if **someone sends data** (i.e. user data), this method won't be called. **This** **method** will **only be called** if someone **tries to open the page** that is **hidden** **behind** the **route**. The model **parameter** will be used to **send data to our view**. Now we need to **return the view**:

Използваме пояснението "@GetMapping". Тази анотация определя, че типът искане, което ще преработим в нашия метод, е "GET". Това означава, че ако някой изпрати данни (т.е. потребителски данни), този метод няма да бъде извикан. Този метод ще бъде извикан само ако някой се опита да отвори страницата, скрита зад маршрута. Параметърът на модела ще бъде използван за изпращане на данни в нашия изглед. Сега трябва да върнем изгледа:

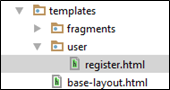


Some of may say "But hey, you are **returning a string**, **not** a **view**" and yes you will be right. Spring however, will **take** **that** **string** and **search** for our **view**. The Model object works with **key-value pairs** just like a **dictionary** (Map in Java). You can see that we are using the addAttribute() method to tell our view, that the variable "view" should be replaced by "user/register". Now we need to **create the view**. Create a **new folder** in the templates package called "user":

Някои от тях могат да кажат "Но хей, вие връщате низ, а не гледка" и да, ще бъдете прав. Пролетта обаче ще вземе този низ и ще търси изгледа ни. Обектът Model работи с двойки ключ-стойност точно като речник (Map in Java). Можете да видите, че използваме метода addAttribute (), за да кажем на нашето изложение, че променливата "view" трябва да бъде заменена с "user / register". Сега трябва да създадем гледката. Създайте нова папка в пакета шаблони, наречен "потребител":



Now create **new HTML file** called register: Сега създайте нов HTML файл, наречен регистър:



In this file, we will only have our **register form**:

В този файл ще имаме само нашия регистрационен формуляр:

|  |
| --- |
| <**main**>  <**div class="container body-content span=8 offset=2"**>  <**div class="well"**>  <**form class="form-horizontal" th:action="@{/register}" method="post"**>  <**fieldset**>  <**legend**>Register</**legend**>  <**div class="form-group"**>  <**label class="col-sm-4 control-label" for="user\_email"**>Email</**label**>  <**div class="col-sm-4 "**>  <**input class="form-control" type="email" id="user\_email" placeholder="Email" name="email" required="required"**/>  </**div**>  </**div**>  <**div class="form-group"**>  <**label class="col-sm-4 control-label" for="user\_fullname"**>Full Name</**label**>  <**div class="col-sm-4 "**>  <**input class="form-control" type="text" id="user\_fullname" placeholder="Full Name" name="fullName" required="required"**/>  </**div**>  </**div**>  <**div class="form-group"**>  <**label class="col-sm-4 control-label" for="user\_password\_first"**>Password</**label**>  <**div class="col-sm-4"**>  <**input type="password" class="form-control" id="user\_password\_first" placeholder="Password" name="password" required="required"**/>  </**div**>  </**div**>  <**div class="form-group"**>  <**label class="col-sm-4 control-label" for="user\_password\_second"**>Confirm Password</**label**>  <**div class="col-sm-4"**>  <**input type="password" class="form-control" id="user\_password\_second" placeholder="Password" name="confirmPassword" required="required"**/>  </**div**>  </**div**>  <**div class="form-group"**>  <**div class="col-sm-4 col-sm-offset-4"**>  <**a class="btn btn-default" th:href="@{/}"**  >Cancel</**a**>  <**input value="Submit" type="submit" class="btn btn-primary"**/>  </**div**>  </**div**>  </**fieldset**>  </**form**>  </**div**>  </**div**>  </**main**> |

You should be familiar with this code, so **let's see if it works**.

## 12. Starting the Project for the First Time

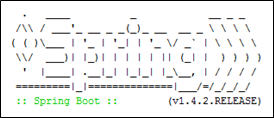
In the **top-right side** of **IntelliJ Idea** you should see this:

Трябва да сте запознати с този код, така че нека да видим дали работи.  
12. Стартиране на проекта за първи път  
В горната дясна част на IntelliJ Idea трябва да видите следното:



Click on the green arrow ( ) and soon you should see something like this:

Кликнете върху зелената стрелка () и скоро ще видите нещо подобно:



Wait until you see this: Изчакайте, докато видите това:



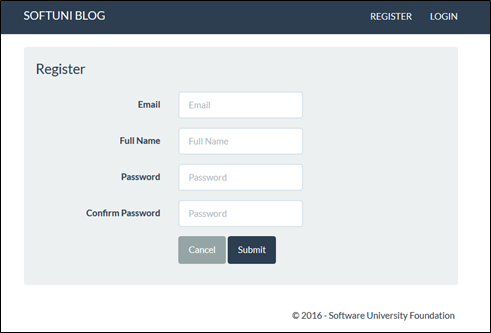
The last row means that the **server is running**. Open <http://127.0.0.1:8080/> and see what you get. You should **receive this error message**:

Последният ред означава, че сървърът работи. Отворете http://127.0.0.1:8080/ и вижте какво получавате. Трябва да получите това съобщение за грешка:



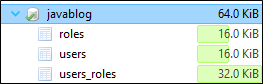
That is normal, **we** **will fix it later**. Try to open <http://127.0.0.1:8080/register>. You should see this:

Това е нормално, ще го поправим по-късно. Опитайте се да отворите http://127.0.0.1:8080/register. Трябва да видите това:



Woah, it works. All of the buttons give us error currently, but this is fine. Our view is rendered and that was what we were trying to do. If you check the database, you can see that we have this:

Woah, това работи. Всички бутони ни дават грешка в момента, но това е добре. Нашето мнение е представено и това се опитвахме да направим. Ако проверите базата данни, можете да видите, че имаме това:



Stop the blog using the  icon.

**Before we continue** with the **user register**, we should create a **home view** to fix the error we've received earlier.

## 13. Creating Home Contoller

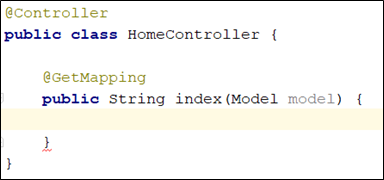
In the controller package create a new class called "HomeController":

Спрете блога, като използвате иконата.  
Преди да продължим с регистъра на потребителите, трябва да създадем начален изглед, за да поправим грешката, която сме получили по-рано.  
13. Създаване на контур на дома  
В пакета на контролера създайте нов клас, наречен "HomeController":



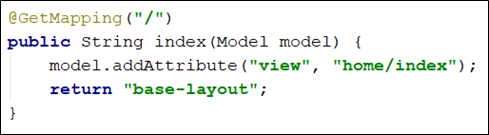
This controller will **list all of our articles** later on, but for now it will just **return an empty page**. Create a new function called "index":

Този контролер ще публикува списък на всичките ни статии по-късно, но за сега просто ще върне празна страница. Създайте нова функция, наречена "индекс":



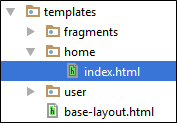
It will catch the **default routing to our blog**. Inside, we should simply **return the desired view**:

Той ще улови маршрута по подразбиране в нашия блог. Вътре трябва просто да върнем желания изглед:



We should create a **new folder** called "home" for our "index" view:

Трябва да създадем нова папка, наречена "дом" за нашия "индекс" изглед:



**Delete everything from the file** and **leave it empty**. When we create our articles, we will edit it.

If we **start the application** and **visit the home page**, we should see this:

Изтрийте всичко от файла и го оставете празно. Когато създаваме статиите си, ще ги редактираме.  
Ако стартираме приложението и посетим началната страница, трябва да видим следното:

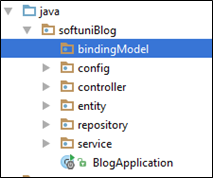


When you click on the **register button**, the hyperlink should lead you to the **register form**. That is everything for our HomeController at this point in time.

## 14. Finishing the User Register

Before we go back to the UserController we need to do something else. In the softuniBlog package create a new package called "bindingModel":

Когато кликнете върху бутона за регистрация, хипервръзката ще ви отведе до регистрационната форма. Това е всичко за нашия HomeController в този момент.  
14. Завършване на регистъра на потребителите  
Преди да се върнем в UserController, трябва да направим нещо друго. В пакета softuniBlog създайте нов пакет, наречен "bindingModel":

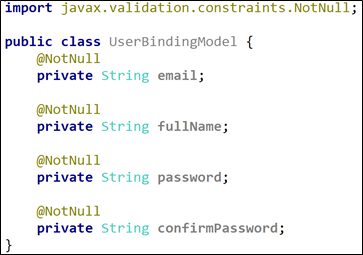


Create a new class called UserBindingModel: Създайте нов клас, наречен UserBindingModel:



This class will **take the data** **from** our **register** **form** and we will **use** **it** to **create a new user**. In order to do that the binding model should contain **the exact fields** that **our form has**. Here is how we should start:

Този клас ще вземе данните от нашия регистрационен формуляр и ще го използваме, за да създадем нов потребител. За да направим това, задължителният модел трябва да съдържа точните полета, които има нашата форма. Ето как трябва да започнем:



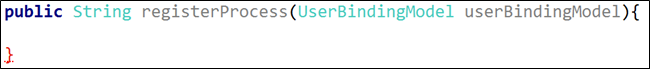
If you check, you will see that these fields have **exactly the same name**, as the **input fields names** in our register form. Let's talk about the **annotation** we are using here. We are saying that those field **cannot be** null or the **user isn’t valid**. We must also create getters and setters for them:

Ако проверите, ще видите, че тези полета имат точно същото име като имената на полетата за въвеждане в нашия регистрационен формуляр. Нека да говорим за анотацията, която използваме тук. Казваме, че тези полета не могат да бъдат нищожни или потребителят не е валиден. Трябва също така да създадем за себе си получатели и създатели:



Let's put this model to good use. Go back to the UserController. Create a new method called "registerProcess":

Нека да използваме този модел добре. Върнете се в UserController. Създайте нов метод, наречен "registerProcess":



This method will have the hard job to **create a new user**. **Spring** will automatically map the form data to our binding model. The only thing that we need to do is **define routing**:

Този метод ще има трудната задача да създаде нов потребител. Пролетта автоматично ще пресметне данните за формуляра в нашия обвързващ модел. Единственото нещо, което трябва да направим, е да определим маршрута:



The PostMapping annotation corresponds to the form method. It means that we will **receive data** from somewhere. The first we want to do in our method is to **check if the passwords match**. If they don't we will **ignore the form submission**:

Появата в PostMapping съответства на метода на формуляра. Това означава, че ще получим данни от някъде. Първото, което искаме да направим в нашия метод, е да проверим дали паролите съвпадат. Ако не го направим, ще пренебрегнем изпращането на формуляра:



The keyword redirect: will change the **current route** **to any given route**. Now we want to create **new password encoder** and **create new object from our user entity** type:

Пренасочването на ключова дума: ще промени текущия маршрут към даден маршрут. Сега искаме да създадем нов енкодер за парола и да създадем нов обект от нашия тип потребителски обект:



Here we use the password encoder to **encode our password**, because we don't want to keep it like a **plain text**. The next thing we want to do – add the default role to our user. To do that we need to go in our User entity and **create a new method** that will **add a new role** to the user:

Тук използваме енкодера за пароли, за да кодираме паролата си, защото не искаме да я държим като обикновен текст. Следващото нещо, което искаме да направим - е да добавим ролята по подразбиране на нашия потребител. За да направим това, трябва да отидем в потребителското ни звено и да създадем нов метод, който ще добави нова роля към потребителя:



Back in our UserController we can't use that method straight away. First, we need to **get the role from our database**:

Обратно в нашия UserController не можем да използваме този метод веднага. Първо, трябва да получим ролята от нашата база данни:

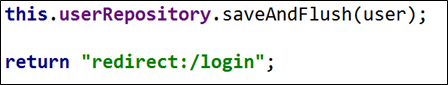


Now we can **add it to our user**: Сега можем да го добавим към нашия потребител:



Finally, we need to **save our user in the database** and **return the login view** that we will create next:

И накрая, трябва да запазим потребителя в базата данни и да върнем изгледа за вход, който ще създадем по-нататък:



Before we test if it works, we want to change something. Maybe you have noticed that **every time** you start the application it **drops the old database** and **creates new one**. **We don't want** **that**. Find your application.properties file. Edit the following line:

Преди да проверим дали работи, искаме да променим нещо. Може би сте забелязали, че всеки път, когато стартирате приложението, тя изпуска старата база данни и създава нова. Ние не искаме това. Намерете файла на application.properties. Редактирайте следния ред:

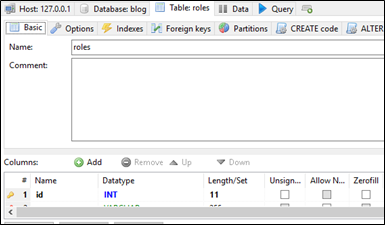


To



Start the application now. Before you open the application, let's create a new role into the database. Open **HeidiSQL**. **Double-click** on the roles table. The main screen should change to this:

Стартирайте приложението сега. Преди да отворите приложението, нека създадем нова роля в базата данни. Отворете HeidiSQL. Щракнете двукратно върху таблицата роли. Основният екран трябва да се промени на следното:



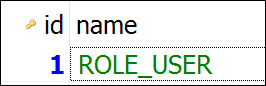
In the navigation, you will see "**Data**" tab. Open it and you should see this:

В навигацията ще видите раздела "Данни". Отворете го и трябва да го видите:



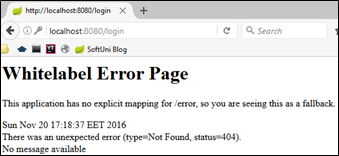
Our database is empty. Click on the "**green plus**" in the **main toolbar**  and you will **be able to enter data** in a **new row**. **Create one role** called "**ROLE\_USER**". It should look like that:

Нашата база данни е празна. Кликнете върху "зеленото плюс" в главната лента с инструменти и ще можете да въвеждате данни в нов ред. Създайте една роля, наречена "ROLE\_USER". Трябва да изглежда така:



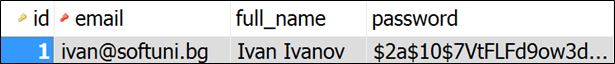
Now we try to register new user. When you **submit the form**, you should **see this error**:

Сега се опитваме да регистрираме нов потребител. Когато изпратите формуляра, трябва да видите тази грешка:



**This is an error yes**, but take a **look at the URL**. We are trying to access "/login" that **doesn't exist**. That means that **our user should be created**. Take a look at the users table in the **database**:

Това е грешка да, но погледнете URL адреса. Опитваме се да осъществим достъп до "/ login", който не съществува. Това означава, че потребителят трябва да бъде създаден. Разгледайте таблицата с потребители в базата данни:

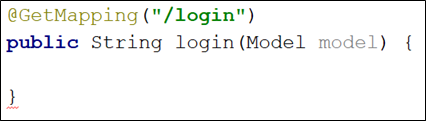


That is **successful registration**! Now we can **create the login**.

## 15. Implement User Login

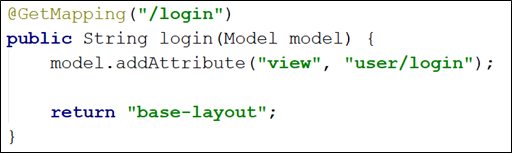
To create a login, we need **only 2 things**. A **method** and a **view**. **Spring Security** **will take care of the rest**. Let's start with the method in our UserController:

Това е успешна регистрация! Сега можем да създадем данните за вход.  
15. Въведете потребителското име  
За да създадете данни за вход, трябва само две неща. Метод и изглед. Spring Security ще се погрижи за останалото. Нека започнем с метода в нашия UserController:



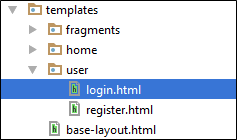
This method will need to **return the login view** and nothing else:

Този метод ще трябва да върне изгледа за вход и нищо друго:



Now we need to create the view. Create a **new html file** in the user folder called "login":

Сега трябва да създадем гледката. Създайте нов html файл в потребителската папка, наречен "вход":



You should delete the existing code and use the following:

Трябва да изтриете съществуващия код и да използвате следното:

|  |
| --- |
| <**main**>  <**div class="container body-content span=8 offset=2"**>  <**div class="well"**>  <**form class="form-horizontal" th:action="@{/login}" method="post"**>  <**fieldset**>  <**legend**>Login</**legend**>  <**div class="form-group"**>  <**label class="col-sm-4 control-label" for="user\_email"**>Email</**label**>  <**div class="col-sm-4"**>  <**input type="email" class="form-control" id="user\_email" placeholder="Email" name="email"**/>  </**div**>  </**div**>  <**div class="form-group"**>  <**label class="col-sm-4 control-label" for="password"**>Password</**label**>  <**div class="col-sm-4"**>  <**input type="password" class="form-control" id="password" placeholder="Password" name="password"**/>  </**div**>  </**div**>    <**input type="hidden" th:name="${\_csrf.parameterName}" th:value="${\_csrf.token}"** />    <**div class="form-group"**>  <**div class="col-sm-4 col-sm-offset-4"**>  <**a class="btn btn-default" th:href="@{/}"**>Cancel</**a**>  <**button type="submit" class="btn btn-primary"**>Login</**button**>  </**div**>  </**div**>  </**fieldset**>  </**form**>  </**div**>  </**div**>  </**main**> |

This should be everything. **Start the blog** and **try to login** with the user you've created previously. You should have this if everything is working:

Това трябва да е всичко. Започнете блога и опитайте да влезете с вече създадения от вас потребител. Трябва да имате това, ако всичко работи:



**The login works**. If you try to **open the user profile** or try **logout** you should **receive errors**. Those are our next targets.

## 16. Implement User Logout

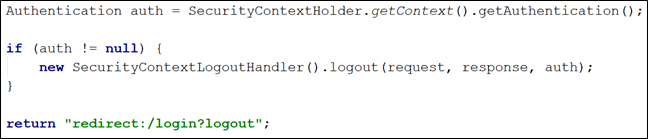
Currently we can login, but **we can't logout**. In our UserController create a new method:

Входът работи. Ако се опитате да отворите потребителския профил или да опитате изход, трябва да получите грешки. Това са следващите ни цели.  
16. Изпълнение на излизане от потребителя  
Понастоящем можем да влезем, но не можем да излезем. В нашия UserController създайте нов метод:



First of all, we are using the "@RequestMapping" annotation. This annotation combines "GET" and "POST" requests (**not** **only**) and we need to specify that we are interested in the "GET" requests only. This method should have the following code:

Първо, използваме пояснението "@RequestMapping". Тази анотация обединява заявките "GET" и "POST" (не само) и трябва да уточним, че се интересуваме само от заявките "GET". Този метод трябва да има следния код:

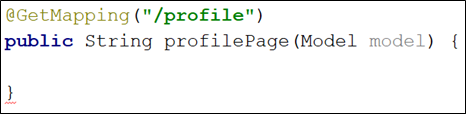


It **checks if there is logged in user** and if there is, it simply **tells the authentication module to logout the user**. Then it **redirects to the login** page again. **The logout is ready**.

## 17. Create User Profile Page

One final thing to create is the **user profile page**. It should give **basic info about the user**. Start by creating a new method in the UserController called "profilePage":

Той проверява дали има потребител влезе и ако има такава, просто казва на модула за удостоверяване да излезе от потребителя. След това отново се пренасочва към страницата за вход. Изходът е готов.  
17. Създаване на страница с потребителски профил  
Едно последно нещо, което трябва да създадете, е страницата на потребителския профил. Тя трябва да дава основна информация за потребителя. Започнете, като създадете нов метод в UserController, наречен "profilePage":



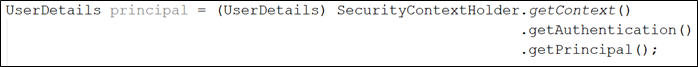
Nothing new for now. First of all, we need to **check** if there is **logged in user**. We **don't want guests** to our blog **to access that page**. We have **many options** to do that, but we are going to use a **feature from Spring**. Add the following **annotation**:

Засега нищо ново. На първо място трябва да проверим дали има потребител. Не искаме посетителите на нашия блог да имат достъп до тази страница. Имаме много възможности да направим това, но ще използваме характеристика от пролетта. Добавете следната анотация:



This annotation automatically check if the **visitor** to our blog **is guest or not**. The page will be **accessed only by logged in users**. **Everyone else** will be **redirected to the login** page. First, we need to get the currently logged in user:

Тази анотация автоматично проверява дали посетителят на нашия блог е гост или не. Страницата ще бъде достъпна само от регистрирани потребители. Всички останали ще бъдат пренасочени към страницата за вход. Първо, трябва да получим понастоящем влезлия потребител:



This will give us only the **basic properties of our user**. That means only username (email in our case), roles and password. We can use it to **extract the current user** from the database:

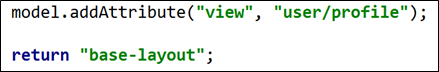
Това ще ни даде само основните свойства на нашия потребител. Това означава само потребителско име (имейл в нашия случай), роли и парола. Можем да го използваме, за да извлечем текущия потребител от базата данни:

Not that we've **extracted the user**, we can **add it to our model**:

Не че сме извадили потребителя, можем да го добавим към нашия модел:



Now we need to **return the view**: Сега трябва да върнем изгледа:



Overall the code should look like this: Като цяло кодът трябва да изглежда по следния начин:



Now we need to **create the view**. In your templates/user directory create a **new html** called "profile":

Сега трябва да създадем гледката. В шаблоните / потребителската директория създайте нов html, наречен "профил":

|  |
| --- |
| <**main**>  <**div class="container body-container"**>  <**div class="row"**>  <**div id="main" class="col-sm-9"**>  <**div**>  <**span th:text="${user.email}"**></**span**>  <**br**/>  <**span th:text="${user.fullName}"**></**span**>  </**div**>  </**div**>  </**div**>  </**div**>  </**main**> |

As you can see we are just using the user object that we've sent using our model. The result should be this:

Както можете да видите, ние просто използваме потребителския обект, който сме изпратили, използвайки нашия модел. Резултатът трябва да бъде следният:



That's all! We've created the **entities** that we need to create the **Many-To-Many** relation. Then we've **configured the Spring Security** module. We've created our **layout** **system** using **Thymeleaf** and finally we've used all this to **implement** **user** **register**, **login**, **logout** and **profile pages**. **Good work you've created the base skeleton**! J

Това е всичко! Създадохме елементите, от които се нуждаем, за да създадем връзката "Много към много". След това конфигурирахме модула Пролетна защита. Създадохме нашата система за оформление, използвайки Thymeleaf и накрая използвахме всичко това, за да внедрим потребителски регистър, вход, изход и страници на потребителския профил. Добра работа, която създадохте основния скелет! 

# VI. Създаване на статии В тази глава ще внедрим функционалността за създаване на статии. Стартирайте MySQL 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. Ще използваме MySQL, който ви е даден в скелета. За да започнете да използвате MySQL, просто щракнете двукратно върху mysql\_start.bat от главната директория (например c: \ project). Ще видите прозорец като този:I. Create Articles

In this chapter, we are going to implement the **article creation functionality**.

## 0. Start MySQL

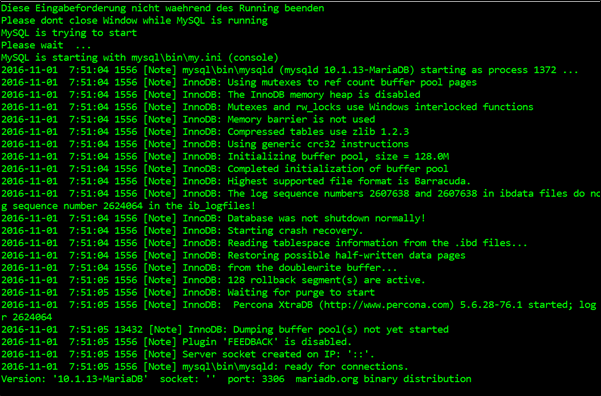
**Skip this step if you have gone through the above III chapters.**

If you are still reading:

Download the [project skeleton](http://softuni.bg/downloads/svn/soft-tech/May-2017/Software-Technologies-July-2017/10.%20Software-Technologies-Java-Blog-Basic-Functionality/10.%20Software-Technologies-Java-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). We will use [MySQL](https://www.mysql.com/), which you are given in the skeleton. To start using MySQL, just **double-click** **mysql\_start.bat** from the root directory (e.g. **c:\project**). You will see a window like this one:

VI. Създаване на статии  
В тази глава ще внедрим функционалността за създаване на статии.  
Стартирайте MySQL  
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. Ще използваме MySQL, който ви е даден в скелета. За да започнете да използвате MySQL, просто щракнете двукратно върху mysql\_start.bat от главната директория (например c: \ project). Ще видите прозорец като този:

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

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

If you are still reading:

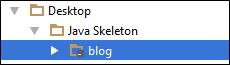
For this step, we will open the project with IntelliJ Idea Ultimate. Starting from the home screen, click on “**Import**”:

Това е, MySQL работи. Когато решите да спрете да работите в блога, просто затворете терминала и стартирайте mysql\_stop.bat файла.  
1. Отворете проекта  
Прескочете тази стъпка, ако сте преминали през горните глави III.  
Ако все още четете:  
За тази стъпка ще отворим проекта с IntelliJ Idea Ultimate. Започвайки от началния екран, кликнете върху "Импортиране":



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 User Role

Using **HeidiSQL** import new Role into the **database** with **name** "ROLE\_USER". If you don't know how to do it, refer to [chapter V part 14](https://euc-word-edit.officeapps.live.com/we/wordeditorframe.aspx?new=1&ui=bg%2DBG&rs=en%2DUS&WOPISrc=https%3A%2F%2Fsoftwareuniversity%2Dmy%2Esharepoint%2Ecom%2Fpersonal%2Farcobaleno%5Fstudents%5Fsoftuni%5Fbg%2F%5Fvti%5Fbin%2Fwopi%2Eashx%2Ffiles%2F8b5c64fe24474a1099b2736704df2638&wdPreviousSession=1ff595e7%2D6fa5%2D43fc%2Da68c%2D4735ea6c4316&wdNewAndOpenCt=1523426258332&wdo=7&wdOrigin=appStartPages&wdTpl=blank&wdLcid=1026&wdPreviousCorrelation=fa005244%2D8afc%2D446f%2Dade9%2D9f20b3d3e7c1&wdEnableRoaming=1&mscc=1&wdODB=1#_Finishing_the_User).

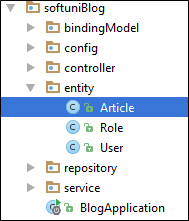
## 3. Start the project

You can find how to **start the project** in [chapter V part 12](https://euc-word-edit.officeapps.live.com/we/wordeditorframe.aspx?new=1&ui=bg%2DBG&rs=en%2DUS&WOPISrc=https%3A%2F%2Fsoftwareuniversity%2Dmy%2Esharepoint%2Ecom%2Fpersonal%2Farcobaleno%5Fstudents%5Fsoftuni%5Fbg%2F%5Fvti%5Fbin%2Fwopi%2Eashx%2Ffiles%2F8b5c64fe24474a1099b2736704df2638&wdPreviousSession=1ff595e7%2D6fa5%2D43fc%2Da68c%2D4735ea6c4316&wdNewAndOpenCt=1523426258332&wdo=7&wdOrigin=appStartPages&wdTpl=blank&wdLcid=1026&wdPreviousCorrelation=fa005244%2D8afc%2D446f%2Dade9%2D9f20b3d3e7c1&wdEnableRoaming=1&mscc=1&wdODB=1#_Starting_the_Project).

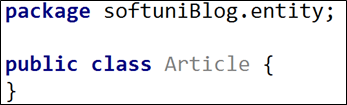
## 4. Article Entity

It's time to create our first entity. We are using [Hibernate](http://hibernate.org/orm/) for **ORM**. That means we are going to define our entities with [annotations](https://docs.jboss.org/hibernate/stable/annotations/reference/en/html/entity.html). In the src/main/java/softuniBlog **package** you can see few packages that **define our project**. A **package** is a **folder** **containing** **Java** files. The one we are interested in is the "entity" package. Inside, create a **new** **java class** called "Article":

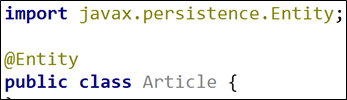
След като кликнете върху "OK", проектът трябва да започне да се зарежда и индексира. След няколко секунди / минути в зависимост от вашия компютър, вие ще можете да работите с проекта.  
2. Създайте потребителска роля  
Използването на HeidiSQL импортира нова Роля в базата данни с име "ROLE\_USER". Ако не знаете как да го направите, направете справка с глава V, част 14.  
3. Стартирайте проекта  
Можете да намерите как да започнете проекта в глава V, част 12.  
4. Предприятие по чл  
Време е да създадем първия си обект. Използваме Hibernate за ORM. Това означава, че ще определим нашите обекти с пояснения. В пакета src / main / java / softuniBlog можете да видите няколко пакета, които определят нашия проект. Пакетът е папка, съдържаща Java файлове. Онова, което ни интересува, е пакетът "субект". Вътре създайте нова класа в Java, наречена "Статия":



The file should look like this: Файлът трябва да изглежда така:



Now we need to tell **Hibernate** that this is an entity: Сега трябва да кажем на Hibernate, че това е субект:



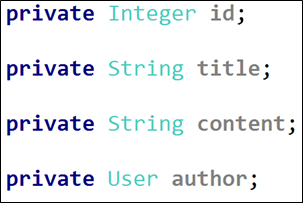
Now that our class is an entity we need to give our database **proper table name**:

Сега, когато нашият клас е субект, трябва да дадем на нашата база данни точното име на таблицата:



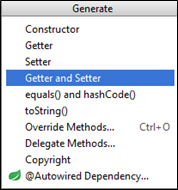
The next important thing is the **table columns**. We need columns for id, title, content and author. Create the following private fields:

Следващото важно нещо са колоните в таблицата. Нуждаем се от колони за идентификационен номер, заглавие, съдържание и автор. Създайте следните частни полета:



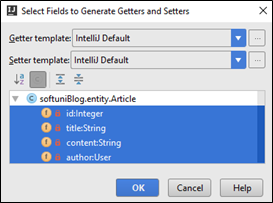
Before we explain each column, let's create [getters and setters](http://java.about.com/od/workingwithobjects/a/accessormutator.htm) for our fields. You should already be familiar with them. If you are curious why are we doing that, you can read more [here](https://www.tutorialspoint.com/java/java_encapsulation.htm). There is a **simple way to create them** in **IntelliJ Idea**. If you press "**Alt + Insert**", you should see that context menu:

Преди да обясним всяка колона, нека създадем getters и setters за нашите полета. Вече трябва да сте запознати с тях. Ако сте любопитни защо правим това, можете да прочетете повече тук. Има лесен начин да ги създадете в IntelliJ Idea. Ако натиснете "Alt + Insert", трябва да видите това контекстно меню:



Choosing the "**Getter and Setter**" option will **open new window**. You should select **all private fields** from there:

Избирането на опцията "Getter and Setter" ще отвори нов прозорец. Трябва да изберете всички частни полета оттук:

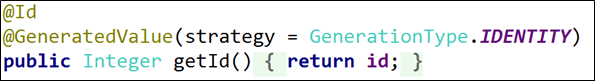


When you **click** "**OK**", you should **receive this code**: Когато кликнете върху "OK", трябва да получите този код:



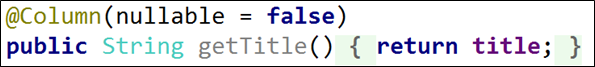
It might be **formatted in a different way**, but the **result should be the same**. Now we can **explain each column to the database**. We are going to **place our annotations on the getters**. The first one is the id **getter**:

Тя може да бъде форматирана по различен начин, но резултатът трябва да бъде същият. Сега можем да обясним всяка колона на базата данни. Ще поставим пояснителните си бележки върху получателите. Първият е получателят на игрите:



The **id** **column** will be the **primary key** in our database and as such we need to use the "@Id" annotation. The "@GeneratedValue" annotation tells **Hibernate** that the database should **generate the values automatically**. The next getter is the title:

Идентификационната колона ще бъде основният ключ в нашата база данни и затова трябва да използваме пояснението "@Id". Анотацията "@GeneratedValue" указва на хибернация, че базата данни трябва автоматично да генерира стойностите. Следващият получател е заглавието:

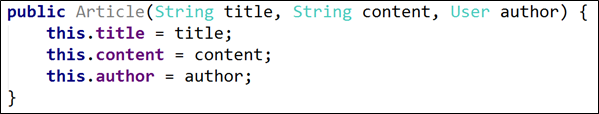


The "@Column" annotation gives us many useful features. For this case however, we only want to tell **Hibernate** that **this column can't be empty**. The content annotation is more interesting:

Анотацията "@Column" ни дава много полезни функции. За този случай обаче искаме само да кажем на Hibernate, че тази графа не може да бъде празна. Анотацията на съдържанието е по-интересно:

Here we are again making the field **required**. By default, fields of type "String" will use the **database type** "VARCHAR(255)". This type is **string** **limited** to **255** **symbols**. We can change the limit, but we can't be sure how long the content of an article will be. That's why we will **change the database type** to "text". The "text" type **doesn’t have limit** on its **length**. We won't touch the author field for now. It's the time to **create our constructor**:

Тук отново правим необходимото поле. По подразбиране полетата от тип "String" ще използват типа база данни "VARCHAR (255)". Този тип е низ ограничен до 255 символа. Можем да променим ограничението, но не можем да сме сигурни колко дълго ще бъде съдържанието на статия. Ето защо ще променим типа на базата данни на "текст". Типът "текст" няма ограничение за неговата дължина. Засега няма да се докосваме до автора. Време е да създадем нашия конструктор:



We will **use** **this** **constructor** to **create articles** easily. However, we need to create another **empty** **constructor** for **Hibernate**:

Ще използваме този конструктор за лесно създаване на статии. Трябва обаче да създадем друг празен конструктор за Hibernate:

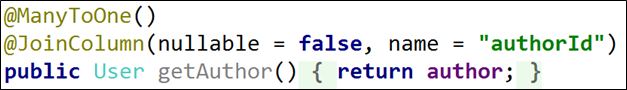


And this is pretty much everything. Our Article entity is almost ready. We need to **define** **the** **relationship** with the User entity now.

## 5. Article-User Relation

Remember that we've left the author field in the Article entity for later? Find the getter. Before we create the annotation, let's talk about the relation between our Article and the User entity. Our relation will be of type [OneToMany](https://en.wikipedia.org/wiki/One-to-many_(data_model)). In our case, we will use “one to many relationship” to tell the program that **one user** will have **many posts**. Let's see the annotations:

И това е почти всичко. Членовете на нашия член са почти готови. Трябва да дефинираме връзката с потребителската единица сега.  
5. Член-потребител връзка  
Не забравяйте, че сме оставили полето за автор в статията за член за по-късно? Намерете получателя. Преди да създадем анотацията, нека да поговорим за връзката между нашия член и потребителското звено. Нашата връзка ще бъде от тип OneToMany. В нашия случай ще използваме "една към много отношения", за да кажем на програмата, че един потребител ще има много публикации. Да видим поясненията:



The first one is the “ManyToOne” annotation. Many to one relationship represents [OneToMany](https://en.wikipedia.org/wiki/One-to-many_(data_model)) relationship from the side of the “many”. Because we are working with the Article entity, we are telling **Hibernate** that **many of our articles** will correspond **to one user**. The other annotation is “JoinColumn”, which tells **Hibernate** that it should **create a column** called "authorId" that will keep our relation and can't be null. This is everything from this side of the relation.

## 6. User-Article Relation

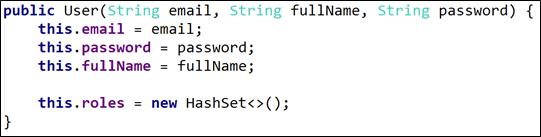
In the User entity, we need to **create a field**, which will keep **all articles** created by a given user:

Първата е анотацията "ManyToOne". Мнозина към една връзка представляват връзката OneToMany от страна на "много". Тъй като работим с члена на статията, казваме на Hibernate, че много от нашите статии ще отговарят на един потребител. Другата анотация е "JoinColumn", която казва на Hibernate, че трябва да създаде колона, наречена "authorId", която ще запази връзката ни и не може да бъде нищожна. Това е всичко от тази страна на връзката.  
6. Връзка между потребител и член  
В потребителското лице трябва да създадем поле, което да съхранява всички статии, създадени от даден потребител:



We creating collection of type "Set". This collection can **contain only unique values** unlike lists and arrays, so that’s why we are using it. **Find the constructor** that looks like this:

Създаваме колекция от тип "Set". Тази колекция може да съдържа само уникални стойности за разлика от списъци и масиви, затова го използваме. Намерете конструктора, който изглежда така:



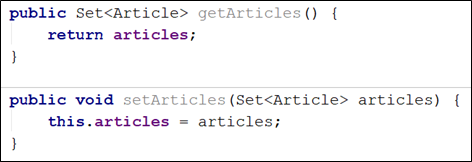
Add another line that will **initialize the articles** collection: Добавете друг ред, който ще инициализира колекцията от статии:



Here we are telling **Java** that our specific type of Set should be the [HashSet](https://docs.oracle.com/javase/7/docs/api/java/util/HashSet.html). The HashSet collection gives us **faster operations** over our collection, but it **doesn't keep the order** of elements. That means that we are going to **win performance**, but when we are **iterating** the collection the elements will be in "**random**" **order**.

Now, create getter and setter for our field:

Тук казваме на Java, че нашият конкретен тип комплект трябва да бъде HashSet. Колекцията HashSet ни дава по-бързи операции по нашата колекция, но тя не поддържа реда на елементите. Това означава, че ще спечелим спектакъл, но когато итерим колекцията, елементите ще бъдат в "случаен" ред.  
Сега създайте getter и setter за нашето поле:



Let's add the annotation for our relation: Да добавим анотацията за нашата връзка:



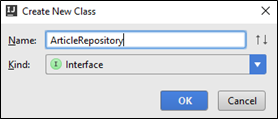
It is pretty simple. It means that **Hibernate** should go to our Article entity and find the "author" field that we've created earlier. Then it will get the **properties** of the relation from there and use them as a base when creating the [foreign key](https://dev.mysql.com/doc/refman/5.6/en/create-table-foreign-keys.html) constraints in the database.

Our relation is ready now.

## 7. Create Article Repository

If you want to read more about **repositories**, you can do it in [chapter V part 6](https://euc-word-edit.officeapps.live.com/we/wordeditorframe.aspx?new=1&ui=bg%2DBG&rs=en%2DUS&WOPISrc=https%3A%2F%2Fsoftwareuniversity%2Dmy%2Esharepoint%2Ecom%2Fpersonal%2Farcobaleno%5Fstudents%5Fsoftuni%5Fbg%2F%5Fvti%5Fbin%2Fwopi%2Eashx%2Ffiles%2F8b5c64fe24474a1099b2736704df2638&wdPreviousSession=1ff595e7%2D6fa5%2D43fc%2Da68c%2D4735ea6c4316&wdNewAndOpenCt=1523426258332&wdo=7&wdOrigin=appStartPages&wdTpl=blank&wdLcid=1026&wdPreviousCorrelation=fa005244%2D8afc%2D446f%2Dade9%2D9f20b3d3e7c1&wdEnableRoaming=1&mscc=1&wdODB=1#_Creating_User_Repository). Here we won't focus on the details. Right now, we **can't create new articles** because we **don't have access to our database**. **Spring** gives us really easy way of communicating with the database. It's called **repository**. Each repository gives us **basic functions** for workingwithgiven **entity** in the **database**. In the "repository" package create a **new java class** called "ArticleRepository" of type **interface**:

Това е доста просто. Това означава, че Hibernate трябва да отиде в нашия член и да намери полето "автор", което сме създали по-рано. Тогава тя ще получи свойствата на връзката от там и ще ги използва като база при създаването на ограниченията на чуждия ключ в базата данни.  
Нашата връзка вече е готова.  
7. Създайте хранилище на статии  
Ако искате да прочетете повече за хранилища, можете да го направите в глава V, част 6. Тук няма да се съсредоточим върху подробностите. В момента не можем да създаваме нови статии, защото нямаме достъп до нашата база данни. Пролетта ни дава наистина лесен начин за комуникация с базата данни. Нарича се хранилище. Всяко хранилище ни дава основни функции за работа с дадено лице в базата данни. В пакета "хранилище" създайте нов java клас, наречен "ArticleRepository" от тип интерфейс:



We should have this: Трябва да имаме това:



The only thing we want to do is tell **Spring** that our **repository** will be of **type** JpaRepository<Entity, Primary Key>:

Единственото нещо, което искаме да направим, е да кажем на Пролетта, че нашето хранилище ще бъде от тип JpaRepository <Entity, Primary Key>:

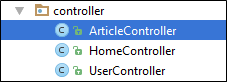


Here we've said that our entity is Article and its **primary key** in the database is of type Integer. **Spring** will do everything else for us.

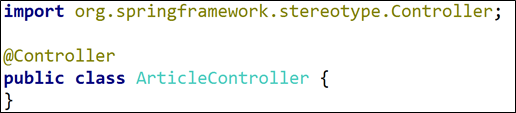
## 8. Create Article Controller

We have finally reached the point in which we can create our **controller**. In the "controller" package create a new class called "ArticleController":

Тук сме казали, че нашата организация е член и нейният първичен ключ в базата данни е от тип Integer. Пролетта ще направи всичко останало за нас.  
8. Създайте член контролер  
Най-накрая стигнахме до момента, в който можем да създадем нашия контролер. В пакета "контролер" създайте нов клас, наречен "ArticleController":

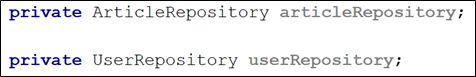


Add the following annotation: Добавете следната анотация:



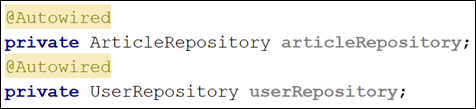
This class will **create**, **edit**, **delete** articles. That means that it will use **routes**. In order to let **Spring,** know that this class will be controller, we need to use the "@Controller" annotation. This annotation also gives us **access** to **requests** and gives us the ability to respond to them. Now, we need to create private fields that will **give** **us** **access** to the **users** and **articles** in the **database**. These fields will be our **repositories**:

Този клас ще създава, редактира, изтрива статии. Това означава, че той ще използва маршрути. За да оставим Spring, знаем, че този клас ще бъде контролер, трябва да използваме пояснението "@Controller". Тази анотация ни дава достъп до заявките и ни дава възможност да отговорим на тях. Сега трябва да създадем частни полета, които ще ни дадат достъп до потребителите и статиите в базата данни. Тези полета ще бъдат нашите хранилища:



Add the following annotation to both of them:

Добавете следното пояснение към двете:

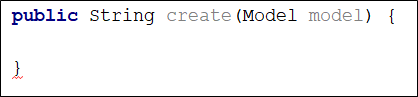


In short, **Spring** creates **object** of **each** **type** that we have in our application each time we **start** our **application**. It keeps them in something called [Spring IoC Container](https://www.tutorialspoint.com/spring/spring_ioc_containers.htm). Using the "@Autowired" annotation, we tell **Spring** that **it should initialize** and **configure** our **repositories** **automatically**. We are ready to start creating articles.

## 9. Creating Articles Part I

We will split the process in **two parts**. The **first part** will be **showing the form** and the **second one** is **creating the article**. Starting with the first part, create the following method in our ArticleController:

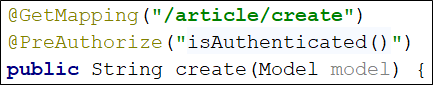
Накратко, "Пролет" създава обект от всеки тип, който имаме в нашето приложение, всеки път, когато стартираме приложението си. Тя ги държи в нещо наречено Spring IoC Container. Като използваме пояснението "@Autowired", казваме, че пролетта трябва да инициализира и конфигурира нашите хранилища автоматично. Ние сме готови да започнем да създаваме статии.  
9. Създаване на членове Част I  
Ще разделим процеса на две части. Първата част ще показва формата, а втората ще създаде статията. Започвайки с първата част, създайте следния метод в нашия членController:



Our

method will use "Model" that **Spring** will **send to the view automatically**. The "Model" is a **special dictionary** that we can use to send **any data** that we want to our **view**. The first thing we want to do is create the annotations:

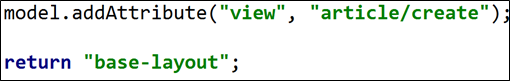
Нашият метод ще използва "Модел", който пролетта ще изпрати автоматично до изгледа. "Моделът" е специален речник, който можем да използваме, за да изпратим всички данни, които искаме да видим. Първото нещо, което искаме да направим, е да създадем поясненията:



The "@GetMapping" annotation tells **Spring** that this method **cannot be called** if the user wants to **submit data**. It should be **only used** for **viewing data**, in our case **showing the form**. The "@PreAuthorize" annotation uses **Spring Security**. That annotation receives a **parameter**, which tell the **authentication** **module** **who can access our method**. We want to **limit the article creation** to **logged in users** only and that’s why we are using the "isAuthenticated()" parameter.

If you want to **know more about** how our **templating system** is working, you can find the information in [chapter V part 10](https://euc-word-edit.officeapps.live.com/we/wordeditorframe.aspx?new=1&ui=bg%2DBG&rs=en%2DUS&WOPISrc=https%3A%2F%2Fsoftwareuniversity%2Dmy%2Esharepoint%2Ecom%2Fpersonal%2Farcobaleno%5Fstudents%5Fsoftuni%5Fbg%2F%5Fvti%5Fbin%2Fwopi%2Eashx%2Ffiles%2F8b5c64fe24474a1099b2736704df2638&wdPreviousSession=1ff595e7%2D6fa5%2D43fc%2Da68c%2D4735ea6c4316&wdNewAndOpenCt=1523426258332&wdo=7&wdOrigin=appStartPages&wdTpl=blank&wdLcid=1026&wdPreviousCorrelation=fa005244%2D8afc%2D446f%2Dade9%2D9f20b3d3e7c1&wdEnableRoaming=1&mscc=1&wdODB=1#_Creating_Base_Layout). In our method write the following code:

Анотацията "@GetMapping" казва на Spring, че този метод не може да бъде извикан, ако потребителят иска да изпрати данни. Тя трябва да се използва само за преглед на данни, в нашия случай показва формата. Анотацията "@PreAuthorize" използва пролетната сигурност. Тази анотация получава параметър, който казва на модула за удостоверяване кой има достъп до нашия метод. Искаме да ограничим създаването на статии само до регистрирани потребители и затова използваме параметъра "isAuthenticated ()".  
Ако искате да научите повече за това как функционира нашата система за създаване на шаблони, можете да намерите информацията в глава V, част 10. В нашия метод напишете следния код:

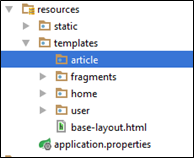


This code will **add** to our **model** a **key-value pair**. The **key** will be the **view** we want to render and the **value** is the **path to our view**. We want to load the "create" file from the "article" folder. Then we simply tell **Spring** to use our **base layout**.

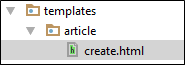
## 10. Creating the View

In order to use **loops** and **logical statements** in our **HTML** we will once again use **templating engine**. You should be **familiar** with **Twig** and **Handlebars** by now. Today you are going to use **Thymeleaf**. [Thymeleaf](http://www.thymeleaf.org/) is a really **easy to use** once you get the hang of it, but it **can be confusing at first**. The idea behind it, is to **replace** the default **HTML attributes** with its **custom attributes** that give us **more functionality**. To start it all, in the "resources/templates" folder **create a new folder** called "article":

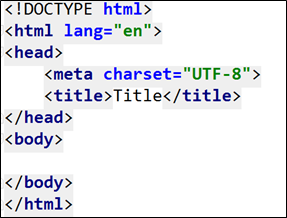
Този код ще добави към нашия модел двойка ключ-стойност. Ключът ще бъде гледката, която искаме да покажем, а стойността е пътят към нашето виждане. Искаме да заредим файла "create" от папката "article". Тогава просто казваме на пролетта да използва нашето оформление на базата.  
10. Създаване на изглед  
За да използваме цикли и логически изявления в нашия HTML, отново ще използваме шаблониращ двигател. Вече трябва да познавате Twig и Handlebars. Днес ще използвате Thymeleaf. Thymeleaf е наистина лесен за използване, след като сте се докоснали до него, но в началото може да бъде объркващо. Идеята зад нея е да замени стандартните атрибути на HTML с нейните потребителски атрибути, които ни дават повече функционалност. За да започнете всичко, в папката "ресурси / шаблони" създайте нова папка, наречена "статия":



Inside that folder create a **new HTML file** called "create": В тази папка създайте нов HTML файл, наречен "create":



By default, our file will look like this: По подразбиране файлът ви ще изглежда така:



We **don’t need that**, so **delete everything**. Use the following code:

Нямаме нужда от това, така че изтрийте всичко. Използвайте следния код:

|  |
| --- |
| <**main**>  <**div class="container body-content span=8 offset=2"**>  <**div class="well"**>  <**form class="form-horizontal" th:action="@{/article/create}" method="POST"**>  <**fieldset**>  <**legend**>New Post</**legend**>    <**div class="form-group"**>  <**label class="col-sm-4 control-label" for="article\_title"**>Article Title</**label**>  <**div class="col-sm-4 "**>  <**input type="text" class="form-control" id="article\_title" placeholder="Article Title"**  **name="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="content"**></**textarea**>  </**div**>  </**div**>    <**div class="form-group"**>  <**div class="col-sm-4 col-sm-offset-4"**>  <**a class="btn btn-default" th:href="@{/}"**>Cancel</**a**>  <**input type="submit" class="btn btn-primary" value="Submit"**/>  </**div**>  </**div**>  </**fieldset**>  </**form**>  </**div**>  </**div**>  </**main**> |

Some of the **Thymeleaf** **attributes** may seem like they are not working, but everything is okay, don't worry. So, **how to identify** Thymeleaf attribute? By the "th:" in front of the attribute name. Let's examine the ones we are using.

At the beginning of the code we can see "th:action="@{/article/create}"". This means that when the **form is submitted** the **request** should go to the "/article/create" **route**. Then it will be processed by some method. Overall the **usage** of "@{}" means that we want to be **redirected** **to** the **route** in the **curly brackets**.

The next and last Thymeleaf attribute is "th:href". It will also **redirect** us **to** a **given** **route**. We can now **test the form** and see how it looks. **Start the application** and go to <http://localhost:8080/article/create>:

Някои от атрибутите на Thymeleaf може да изглеждат така, сякаш не работят, но всичко е наред, не се безпокойте. И така, как да идентифицирате атрибута на Thymeleaf? Чрез "th:" пред името на атрибута. Нека разгледаме тези, които използваме.  
В началото на кода може да се види "th: action =" @ / / article / create} "". Това означава, че когато се подаде формулярът, заявката трябва да отиде в маршрута "/ article / create". След това тя ще бъде обработена по някакъв начин. Като цяло използването на "@ {}" означава, че искаме да бъдем пренасочени към маршрута в скосените скоби.  
Следващият и последният атрибут на Thymeleaf е "th: href". Това също ще ни пренасочи към даден маршрут. Вече можем да тестваме формата и да видим как изглежда. Стартирайте приложението и отидете на http: // localhost: 8080 / article / create:

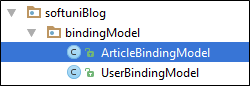


Looks good, but it **doesn't work**. We need to fix that.

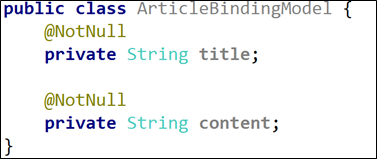
## 11. Creating Article Binding Model

In the previous part, we've **created our html**. That gave us the **design of the form**. We still need to **validate** the **user input**. This is done by creating **binding models**. The idea behind them is to **fill the user input inside** and **validate it**. **If it validates**, **we can** **use it** in our application. In the "bindingModel" package create a new class called "ArticleBindingModel":

Изглежда добре, но това не работи. Трябва да поправим това.  
11. Създаване на модел за обвързване на членове  
В предишната част създадохме нашия html. Това ни даде дизайна на формата. Все още трябва да потвърдим входа на потребителя. Това се прави чрез създаване на задължителни модели. Идеята зад тях е да запълнят потребителския вход и да го потвърдят. Ако го валидираме, можем да го използваме в нашата молба. В пакета "bindingModel" създайте нов клас, наречен "ArticleBindingModel":

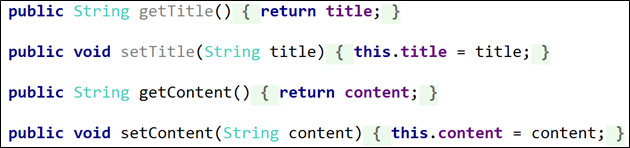


In that class create the following **private fields**: В този клас създайте следните частни полета:



The "@NotNull" annotation is the only **validation** we are going to use. If the user tries to **submit** our **form** **without data**, it **will not validate**. If you check, you will see that these fields have **exactly the same name**, as the **input fields names** in our create form. This is **really important**. If they have **different names** **Spring** **won't be able** to **autofill** the **binding model**. The last thing that we need here is to create getters and setters:

Анотацията "@NotNull" е единственото потвърждение, което ще използваме. Ако потребителят се опита да ни изпрати формуляр без данни, той няма да бъде валидиран. Ако проверите, ще видите, че тези полета имат точно същото име като имената на полетата за въвеждане в нашия формуляр за създаване. Това е наистина важно. Ако имат различни имена, Spring няма да могат автоматично да попълнят модела на свързване. Последното нещо, от което се нуждаем тук, е да създаваме купувачи и създатели:

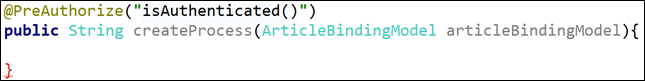


Our **form validation is ready**. Let's create the articles in the database.

## 12. Creating Articles Part II

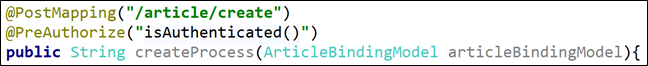
Here comes the second part that we've talked earlier about. In our ArticleController create a new method called "createProcess":

Утвърждаването на формата ни е готово. Нека създадем статиите в базата данни.  
12. Създаване на членове Част II  
Тук идва втората част, за която говорихме по-рано. В нашия членController създайте нов метод, наречен "createProcess":



You are familiar with the annotation from the previous method. When we use the **binding model** in our method, **Spring** will **autofill** it, if we've created it correctly. We need one more annotation before we create the functionality:

Вие сте запознати с анотацията от предишния метод. Когато използваме свързващия модел в нашия метод, Spring ще го попълни автоматично, ако сме го създали правилно. Преди да създадем функционалността, имаме нужда от още едно пояснение:



Before we talk about the "@PostMapping" annotation, take a look at the **route**. It’s the **exactly same route** as the one we've used in our other method. So, what have we done? With this annotation, we told **Spring** that **this method expects data** that it needs to **autofill in our binding model**. The annotation handles "POST" **request** that are usually what the **HTML forms** are using as a "method" of the **request**. In summary, the **other method** **will be called** when the user wants to **create new article** (**render the form**) and **this method** will be called **when he submits the data**.

So, what do we need? First, we need to get the **currently** **logged in user**:

Преди да поговорим за пояснението "@PostMapping", разгледайте маршрута. Това е точно същия маршрут като този, който сме използвали при другия ни метод. И така, какво направихме? С това пояснение казахме на пролетта, че този метод очаква данните, които е необходимо да се попълнят автоматично в нашия обвързващ модел. Анотацията обработва заявката "POST", която обикновено се използва от HTML формулярите като "метод" на заявката. В обобщение, другият метод ще бъде извикан, когато потребителят иска да създаде нова статия (да направи формуляра) и този метод ще бъде извикан, когато подаде данните.  
И така, от какво се нуждаем? Първо, трябва да получим понастоящем влезлия потребител:

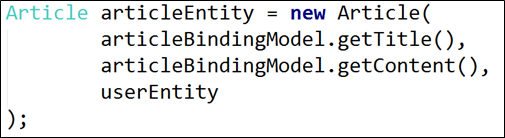
This will give us only the **basic properties of our user**. That means only username (email in our case), roles and password. We can use it to **extract the current entity user** from the database:

Това ще ни даде само основните свойства на нашия потребител. Това означава само потребителско име (имейл в нашия случай), роли и парола. Можем да го използваме, за да извлечем текущия потребителски обект от базата данни:



We are using the user repository to **find a user** by his **email**. **Spring Security** saves **username**, but in our case this is our **email**. Now that we have the **user**, we can **create new article**:

Използваме потребителското хранилище, за да намерим потребител по имейл. Spring Security запазва потребителското име, но в нашия случай това е нашият имейл. Сега, когато имаме потребител, можем да създадем нова статия:



Then, we can **upload it to our database**, using our **article repository**:

След това можем да го качим в нашата база данни, използвайки хранилището на статията ни:



Finally, we want to **redirect our user** to the **home page** of our blog. We will use the "redirect:" syntax:

И накрая, искаме да пренасочим потребителя към началната страница на нашия блог. Ще използваме синтаксиса "пренасочване:":



In summary, we've got the **user** that **Spring Security** is using, then got the **real entity user** using his **email**. Then we've **created a new article** and **saved it to the database**. Finally, we've **redirected the user** to the **home page**. The code should look like this:

В обобщение, имаме потребител, който използва Пролетната сигурност, а след това получи истинския потребител, използвайки имейла си. След това създадохме нова статия и я запазихме в базата данни. И накрая, пренасочихме потребителя към началната страница. Кодът трябва да изглежда така:

Let's add the "**create**" **button** to our **navigation bar** and test our method.

## 13. Add Button to the Base Layout

Open the "templates/fragments/header" file. Find the following section:

Да добавим бутона "Създаване" в навигационната лента и да тестваме нашия метод.  
13. Добавете бутон към основното оформление  
Отворете файла "шаблони / фрагменти / заглавка". Намерете следната секция:

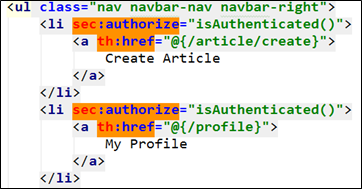


Here we are displaying the buttons in the navigation bar. We are using the "sec:authorize" **attribute** from **Thymeleaf Security**, which gives is the ability to **check** if there is **logged in user**. Add the following code at the **beginning of the list**:

Тук показваме бутоните в навигационната лента. Използваме атрибута "sec: authorize" от Thymeleaf Security, който дава възможност да се провери дали има потребител. Добавете следния код в началото на списъка:

|  |
| --- |
| <**li sec:authorize="isAuthenticated()"**>  <**a th:href="@{/article/create}"**>  Create Article  </**a**>  </**li**> |

It should look like this now: Трябва да изглежда така:

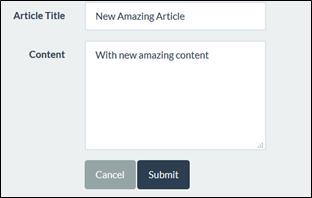


Let’s test our application: Нека да тестваме приложението си:



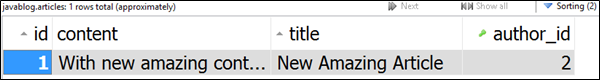
We have the button in the navigation bar. Can we register a new post?

Имаме бутона в навигационната лента. Можем ли да регистрираме нова публикация?



After we submit the form, we get **redirected to the home page**. That should tell us that **our method is working**. Let's see the database:

След като изпратим формуляра, ще се пренасочим към началната страница. Това трябва да ни каже, че методът ни работи. Да видим базата данни:

**It is working**! In the next chapter, we will **display our articles** on the **home page**.

# VII. List Articles

## 1. Listing All Articles

We can create articles now, so the next logical thing is to display them. First, open the Article entity and create a **new method** that will return **half of our content**:

Работи! В следващата глава ще покажем статиите си на началната страница.  
VII. Списък на членовете  
1. Вписване на всички статии  
Вече можем да създаваме статии, така че следващото логично нещо е да ги покажете. Първо, отворете членството и създайте нов метод, който ще върне половината от съдържанието ни:



We want to tell **Hibernate** that **this method shouldn't be saved in our database**. We will do that by using the following annotation:

Искаме да кажем на Hibernate, че този метод не трябва да се съхранява в нашата база данни. Ще направим това, като използваме следното пояснение:



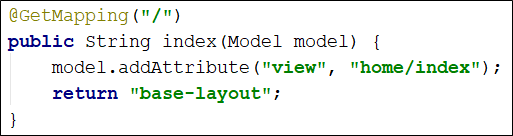
There are [other annotations](http://www.objectdb.com/java/jpa/entity/fields) that can manipulate what goes into the database. Now we just need return half of our content:

Съществуват и други анотации, които могат да манипулират какво се случва в базата данни. Сега просто трябва да върнем половината от съдържанието ни:

Now we have our **summary**. We can start **listing the posts**.

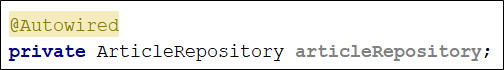
Open the HomeController and find the index() method:

Сега имаме нашето резюме. Можем да започнем да записваме публикациите.  
Отворете HomeController и намерете метода index ():



This will be the method we are going to use to **display our articles**. Before that, create a **new private article repository**:

Това ще бъде методът, който ще използваме, за да показваме нашите статии. Преди това създайте ново хранилище за частни статии:



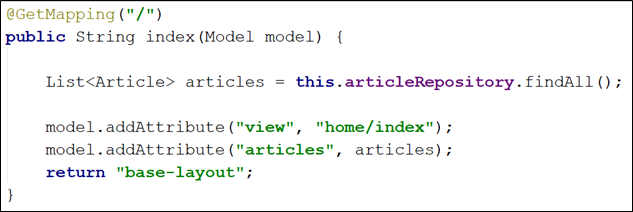
Now, in our method we can use it to **get all articles**:

Сега, в нашия метод, можем да го използваме, за да получим всички статии:



Finally, we should **add them** **to** the **model** in order to send them to the view:

И накрая, трябва да ги добавим към модела, за да ги изпратим на изгледа:



That's all for our controller. Now you need to **open** the "templates/home/index" view and use the following code:

Това е всичко за нашия контрольор. Сега трябва да отворите изгледа "templates / home / index" и да използвате следния код:

|  |
| --- |
| <**main**>  <**div class="container body-content"**>  <**div class="row"**>  <**th:block th:each="article : ${articles}"**>  <**div class="col-md-6"**>  <**article**>  <**header**>  <**h2 th:text="${article.title}"**></**h2**>  </**header**>    <**p th:text="${article.summary}"**></**p**>    <**small class="author" th:text="${article.author.fullName}"**></**small**>    <**footer**>  <**div class="pull-right"**>  <**a class="btn btn-default btn-xs"** [**th:href="@{/article/{id}(id=${article.id})}"**>Read](mailto:th:href="@{/article/{id}(id=${article.id})}">Read) more **&raquo;**</**a**>  </**div**>  </**footer**>  </**article**>  </**div**>  </**th:block**>  </**div**>  </**div**>  </**main**> |

Let's review our Thymeleaf attributes. The first one that we can see is "th:block". **It is not exactly** an **attribute**. It is an **empty block** that can be used as a **container for other Thymeleaf attributes**.

Inside, there is other attribute - "th:each="article : ${articles}"". This is a foreach loop. Thy syntax is really similar to the foreach loop in Java. We are getting each article from the articles collection that we've sent to our view. One thing that has to be mentioned is that the "${}" syntax tells **Thymeleaf** that we are going to **use variable** in our **view**.

The next attribute is "th:text". It fills the tag with the given text.

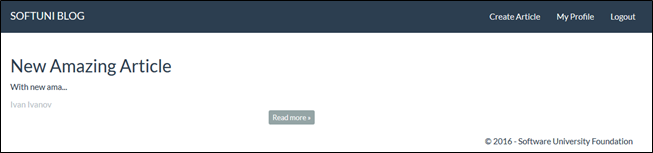
One last thing. Our "th:href" attribute is different from the ones we've seen. You can **ignore** that the **route** it **redirects to is invalid**, because **we will fix that** in the next part. The **focus** should be on that part:

Нека разгледаме нашите атрибути Thymeleaf. Първият, който можем да видим, е "th: block". Това не е точно атрибут. Това е празен блок, който може да се използва като контейнер за други атрибути на Thymeleaf.  
Вътре има и друг атрибут - "th: each =" article: $ {articles} "". Това е foreach цикъл. Синтаксисът ви е наистина подобен на foreach loop в Java. Получаваме всяка статия от колекцията от статии, която сме изпратили на нашето мнение. Едно нещо, което трябва да се спомене, е, че синтаксисът "$ {}" казва на Thymeleaf, че ще използваме променлива според нас.  
Следващият атрибут е "th: text". Попълва маркера с дадения текст.  
Последно нещо. Нашият атрибут "th: href" е различен от тези, които сме виждали. Можете да пренебрегнете, че маршрутът, към който пренасочва, е невалиден, защото ще го поправим в следващата част. Акцентът трябва да бъде върху тази част:



We are **sending parameters** in our **URL** using that syntax. In **curly** **brackets**, we **define the parameter name** and **in the end** of the **URL** we use **normal brackets to fill each parameter**. We can test if the listing works:

Изпращаме параметри в нашия URL адрес, използвайки този синтаксис. В къси скоби определяме името на параметъра и в края на URL адреса използваме нормални скоби за попълване на всеки параметър. Можем да проверим дали регистрацията работи:

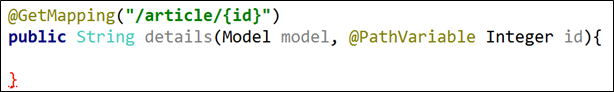
It works like a charm. Now we need to create **article details** page.

## 2. Single Article Details

Now we will create the article details page, that will show the full content of our articles.

Create new method in our ArticleController called "details":

Тя работи като чар. Сега трябва да създадем страница с подробности за статия.  
2. Подробни подробности за един член  
Сега ще създадем страницата с подробности за статия, която ще покаже пълното съдържание на статиите ни.  
Създайте нов метод в нашия членController, наречен "подробности":

Something new! In our route, we declare parameter using curly brackets. Then in our method we use the "@PathVariable" annotation to tell Spring that this parameter should be taken from the URL. We are now free to use it in our method. The first thing we want to do is check if there is article with the given id in our database. If such article doesn't exist, we will redirect the user to the home page:

Нещо ново! В нашия маршрут декларираме параметър с помощта на скосени скоби. След това в нашия метод използваме пояснението "@PathVariable", за да кажем на Spring, че този параметър трябва да бъде взет от URL адреса. Вече можем да го използваме в нашия метод. Първото нещо, което искаме да направим, е да проверим дали има статия с дадения id в нашата база данни. Ако такава статия не съществува, ще пренасочим потребителя към началната страница:

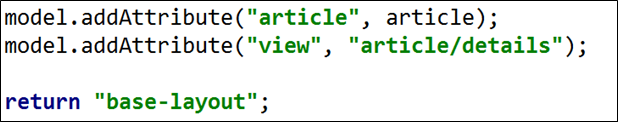


The next thing is to get the article from the database using our repository:

Следващото нещо е да получите статията от базата данни, използвайки нашето хранилище:

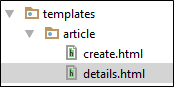
Using this method, we are searching in the database by primary key. Now we want to send the article and the view to our layout:

Използвайки този метод, ние търсим в базата данни чрез първичен ключ. Сега искаме да изпратим статията и изгледа към нашето оформление:



Now we need to create our view. In the article folder, create new HTML called "details":

Сега трябва да създадем нашия възглед. В папката на статиите създайте нов HTML, наречен "подробности":



Delete the existing code and use the following: Изтрийте съществуващия код и използвайте следното:

|  |
| --- |
| <**main**>  <**div class="container body-content"**>  <**div class="row"**>  <**div class="col-md-12"**>  <**article**>  <**header**>  <**h2 th:text="${article.title}"**></**h2**>  </**header**>    <**p th:text="${article.content}"**> </**p**>    <**small class="author" th:text="${article.author.fullName}"**></**small**>    <**footer**>    <**div class="pull-right"**>    <**a class="btn btn-success btn-xs" th:href="@{/article/edit/{id}(id = ${article.id})}"**>Edit</**a**>  <**a class="btn btn-danger btn-xs" th:href="@{/article/delete/{id}(id = ${article.id})}"**>Delete</**a**>    <**a class="btn btn-default btn-xs" th:href="@{/}"**>back **&raquo;**</**a**>  </**div**>  </**footer**>  </**article**>  </**div**>  </**div**>  </**div**>  </**main**> |

Have in mind that the "Edit" and "Delete" **buttons** **won't work** at this point, because the **routes** are invalid. Try to **open article** in our blog:

Имайте предвид, че бутоните "Редактиране" и "Изтриване" няма да работят в този момент, тъй като маршрутите са невалидни. Опитайте се да отворите статия в нашия блог:

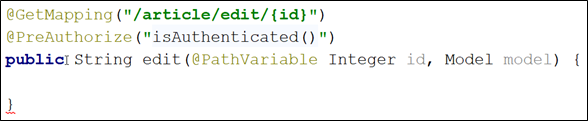
It is working. J

# VIII. Editing Articles

## 1. Creating the Get Method

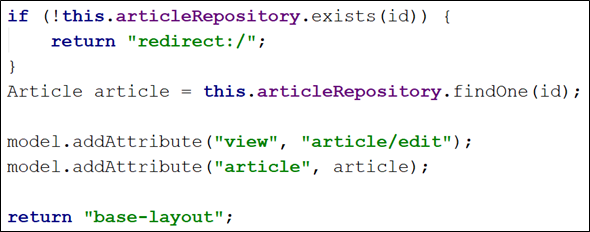
We know the drill now. Create new method in our ArticleController:

Работи.   
VIII. Редактиране на статии  
1. Създаване на метод за получаване  
Знаем тренировката сега. Създайте нов метод в нашия членController:



This method will be **similar** to our **details** method:

Този метод ще бъде подобен на нашия метод за подробности:

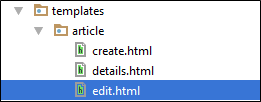


The **only difference** is that we are using **another view**. View that doesn't exist yet.

## 2. Creating the View

In the article folder create a new view called "edit":

Единствената разлика е, че използваме друг изглед. Преглед, който все още не съществува.  
2. Създаване на изгледа  
В папката на статиите създайте нов изглед, наречен "редактиране":

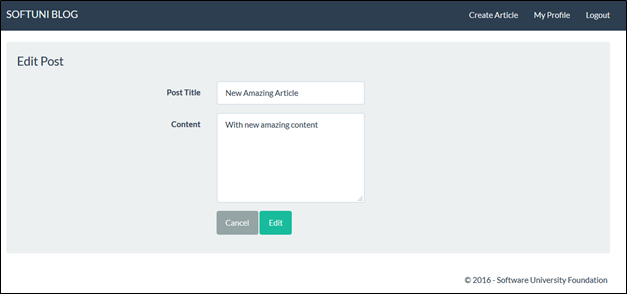


Use the following code: Използвайте следния код:

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

Here, you can see we are using "th:value" and "th:field". The value attribute is used to give **input fields** value. The <textarea> is a **special input field** that **doesn’t have value attribute**. Because of that, we are using "th:field", which will **replace** some of our **original attributes** in order to **fill the content** in our textarea. Everything else is similar to what we've done before. Let's see how it looks:

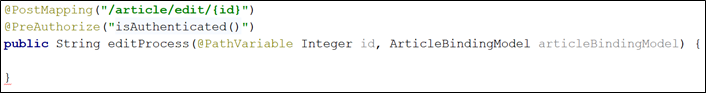
Тук можете да видите, че използваме "th: value" и "th: field". Стойността на атрибута се използва, за да се даде стойност на полетата за въвеждане. <Textarea> е специално поле за въвеждане, което няма атрибут на стойност. Поради това използваме "th: field", което ще замести някои от нашите оригинални атрибути, за да запълни съдържанието в текста ни. Всичко останало е подобно на това, което сме правили преди. Да видим как изглежда:

Not that bad. Let's make it work.

## 3. Creating the Post Method

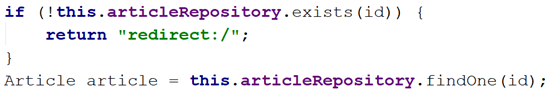
New task – create the post method. In our ArticleController create new method with the following annotations:

Не е толкова зле. Нека да го направим.  
3. Създаване на метод за публикуване  
Нова задача - създайте следния метод. В нашия членController създайте нов метод със следните пояснения:



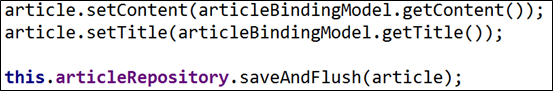
As you can see we will use our **binding model** to **validate** the **user input**. The next thing we need to do is **check if the article exists**, and get it if it does:

Както виждате, ще използваме нашия обвързващ модел, за да потвърдим входа на потребителя. Следващото нещо, което трябва да направим, е да проверим дали статията съществува и да я получи, ако го направи:



Once we have our article, we just need to **set the new title and content** and **save our article** in the database:

Щом имаме статията си, ние просто трябва да зададем новото заглавие и съдържание и да запазим статията си в базата данни:



Finally, **redirect** the user **to the article details**:

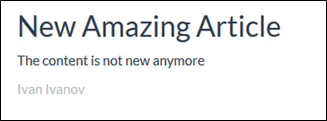
Накрая, пренасочвайте потребителя към подробностите за статия:



Test it and see if it works: Пробвайте и вижте дали работи:



It should be working just fine: Тя трябва да работи добре:

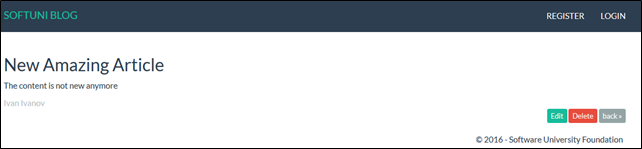


However, there is a **slight problem**…

## 4. Hiding Buttons

Don't know if you've noticed, but when you are **not logged in**, you still see the edit and delete buttons:

Има обаче лек проблем ...  
4. Скриване на бутоните  
Не знаете дали сте забелязали, но когато не сте влезли в профила си, пак виждате бутоните за редактиране и изтриване:



We don't want that. Open your details **view** and find the buttons:

Ние не искаме това. Отворете изгледа с подробности и намерете бутоните:

You can see that our buttons **aren't secured**. Write the following code around them:

Можете да видите, че нашите бутони не са защитени. Напишете следния код около тях:



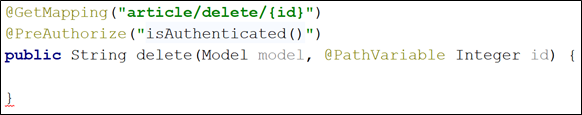
Using **Thymeleaf Security** we are making sure that **only logged in users** **will** **see the buttons**.

# IX. Deleting Articles

## 1. Creating the Get Method

Here we go again. In our ArticleController we will create another method:

С помощта на Thymeleaf Security се уверяваме, че само влезлите в системата потребители ще видят бутоните.  
IX. Изтриване на статии  
1. Създаване на метод за получаване  
Пак започваме. В нашия член Контролър ще създадем друг метод:



You should write the following code, that may look familiar:

Трябва да напишете следния код, който може да изглежда познат:

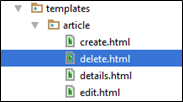


The only difference between this code and the edit() method is the **view** we are using.

## 2. Creating the View

You need to **create a new view** called "delete" in the "article" folder:

Единствената разлика между този код и метода edit () е изгледа, който използваме.  
2. Създаване на изгледа  
Трябва да създадете нов изглед, наречен "изтриване" в папката "статия":

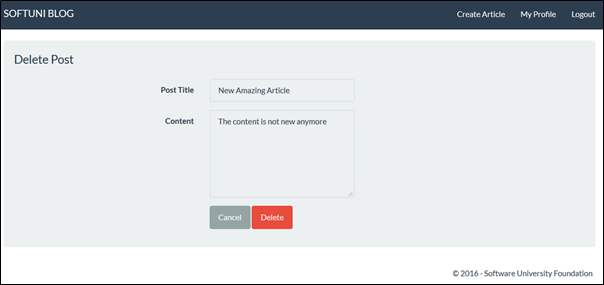


For that view we will use the same code that we've used for the edit view, except for the fact that **the** action **will be different** and **the fields will be** disabled:

За този изглед ще използваме същия код, който сме използвали за изгледа за редактиране, с изключение на факта, че действието ще бъде различно и полетата ще бъдат деактивирани:

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

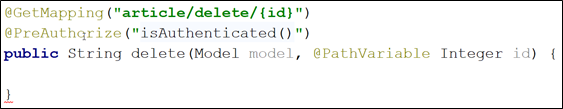
That is how it should look: Ето как трябва да изглежда:

Nothing new here, let's move on.

## 3. Creating the Post Method

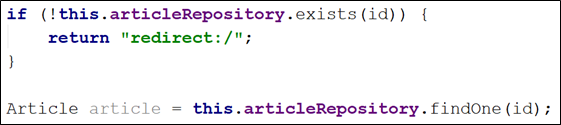
This **new** method in the ArticleController will actually be interesting:

Нищо ново тук, нека да продължим.  
3. Създаване на метод за публикуване  
Този нов метод в ArticleController действително ще бъде интересно:



We **don't need a binding model** here, because **we are not submitting the form**. We are just **using it to verify** that **the user wants to delete the article**. The first part is going to be standard:

Тук нямаме нужда от задължителен модел, защото не представяме формуляра. Просто го използваме, за да потвърдим, че потребителят иска да изтрие статията. Първата част ще бъде стандартна:



We are checking if such **article exists** and then **taking it from the database**. Now we simply need to tell our **repository** to **remove it** from the database:

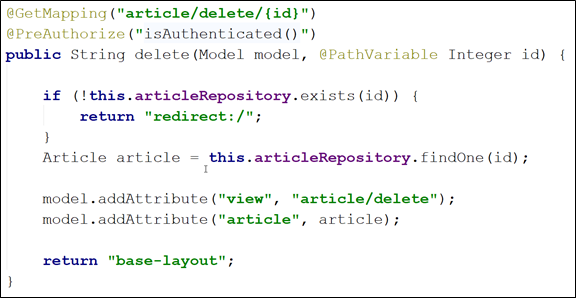
Проверяваме дали съществува такава статия и след това я вземете от базата данни. Сега просто трябва да кажем на нашия хранилище да го премахне от базата данни:



Finally, we need to **redirect the user** to the home page: И накрая, трябва да пренасочим потребителя към началната страница:



Overall, the code should look like this: Като цяло, кодът трябва да изглежда така:



We are **verifying that the article exists**, then **deleting it from the database**. It should work now, so you can test it.

With that we finished our Java Blog. Feel free to build on your project even further. J

Ние проверяваме дали статията съществува, след това я изтриваме от базата данни. Трябва да работи сега, така че да можете да го тествате.  
С това завършихме нашия Java блог. Чувствайте се свободни да изградите още повече своя проект. 