# 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.

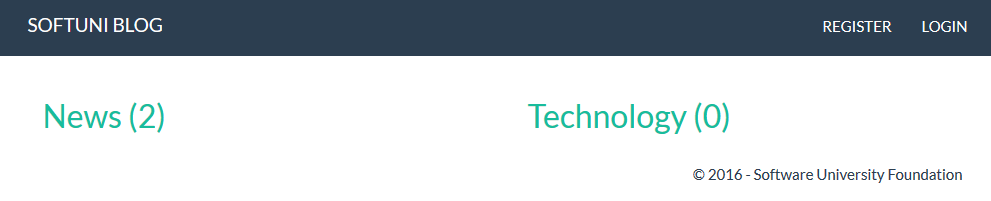
**Chapters from III to V are for advanced users. There’s a** [**skeleton**](https://softuni.bg/downloads/svn/soft-tech/Sep-2016/Software-Technologies-Oct-2016/09.%20Java-Blog-Basic-Functionality/Java%20Skeleton.zip) **which you can use and start from chapter VI, after you set up the JDK in chapter II.**

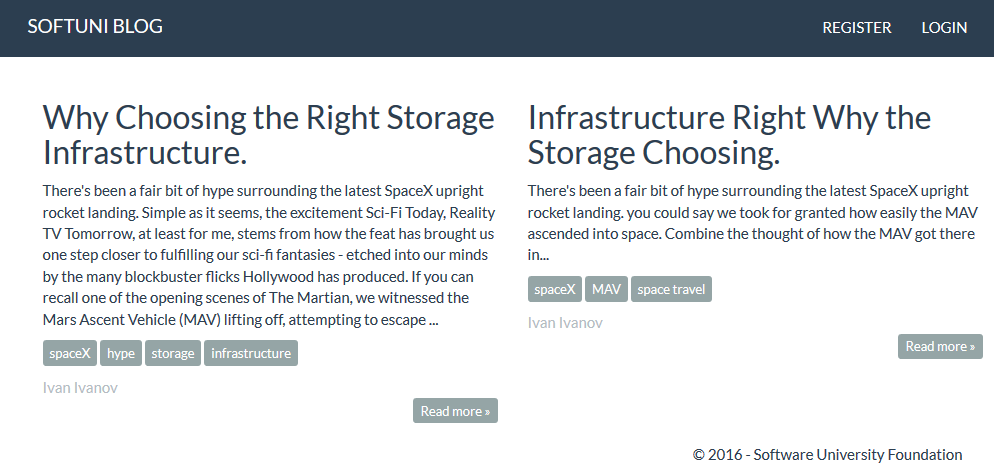
# What Will You Create

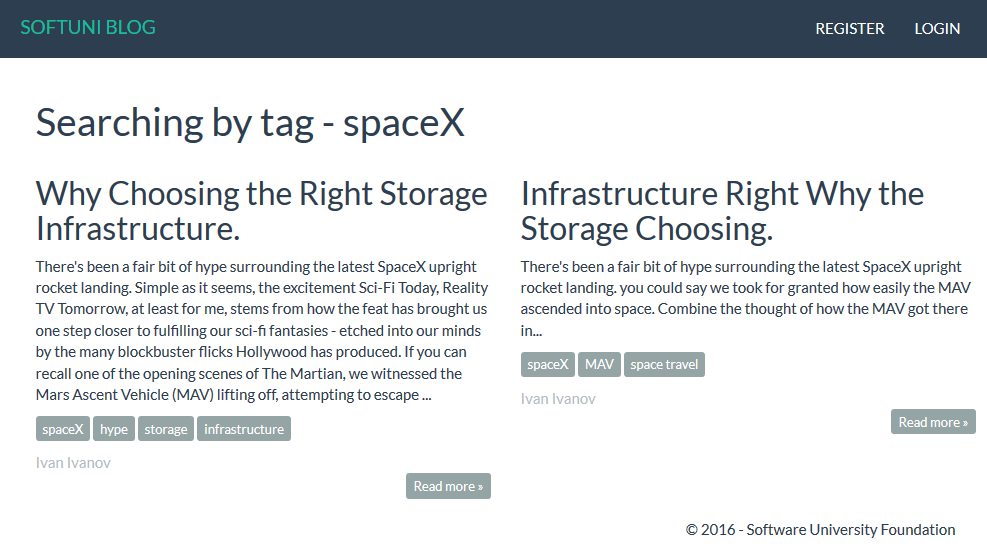
In the end of the whole tutorial, you should have blog which supports the following functionality:

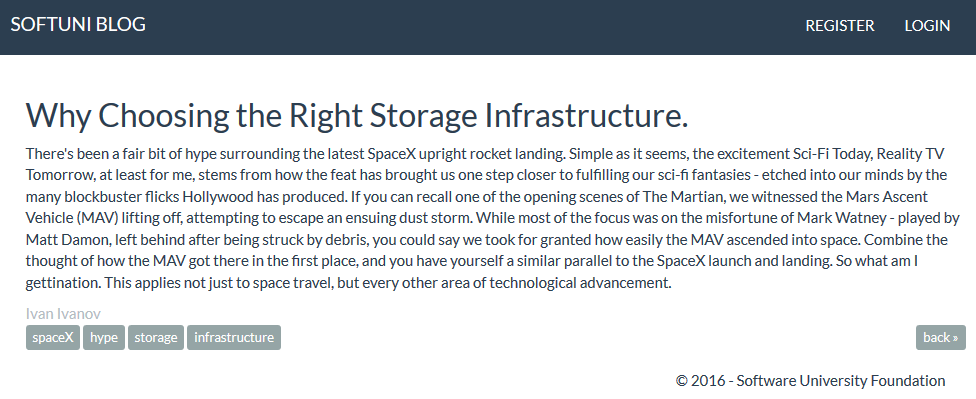
* **Register** and **Login Users**
* **Create**, **Update**, **Read** and **Delete** **Articles**
* **Admin** functionality – **Edit** and **Delete** **Users**
* **Create**, **Update** and **Delete** **Categories**
* **Create**, **Update** and **Delete** **Tags**
* **Filter** the **articles** by **Tags** and **Categories**
* **Responsive Design** using **Bootstrap**
* **More…**

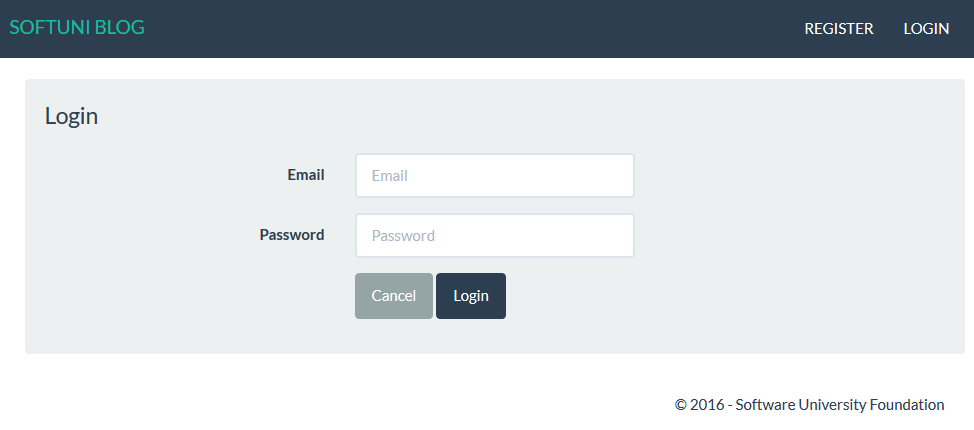
Pictures:

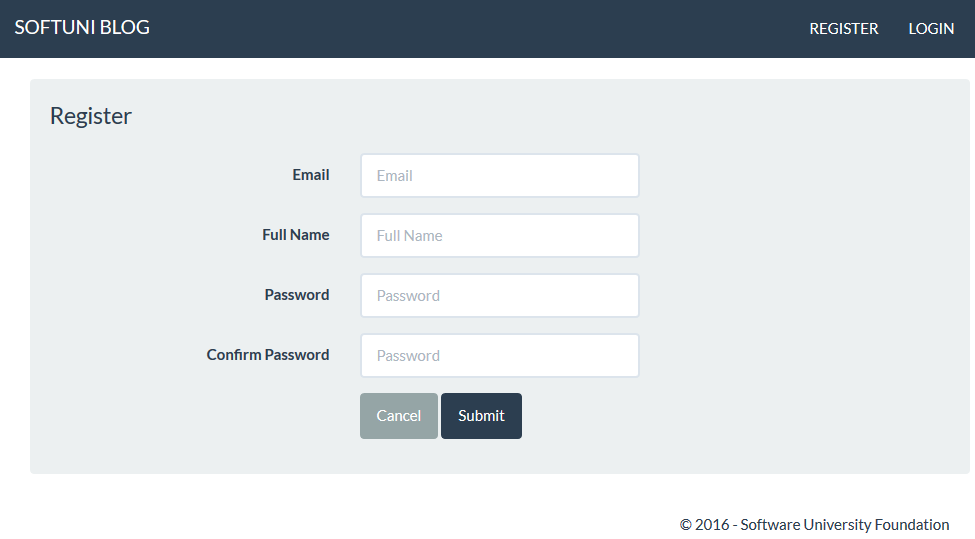




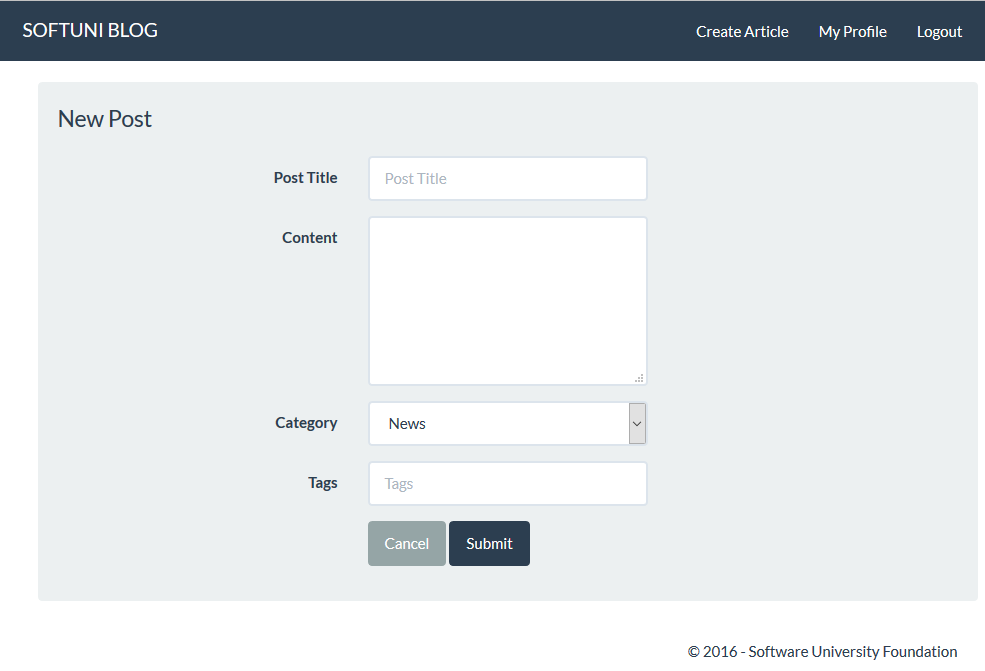


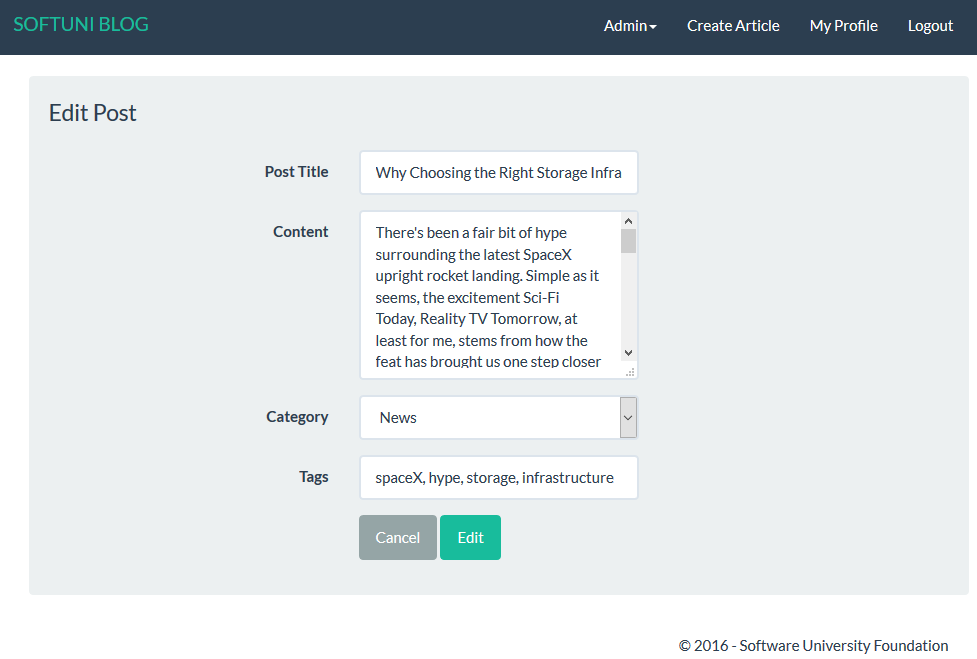


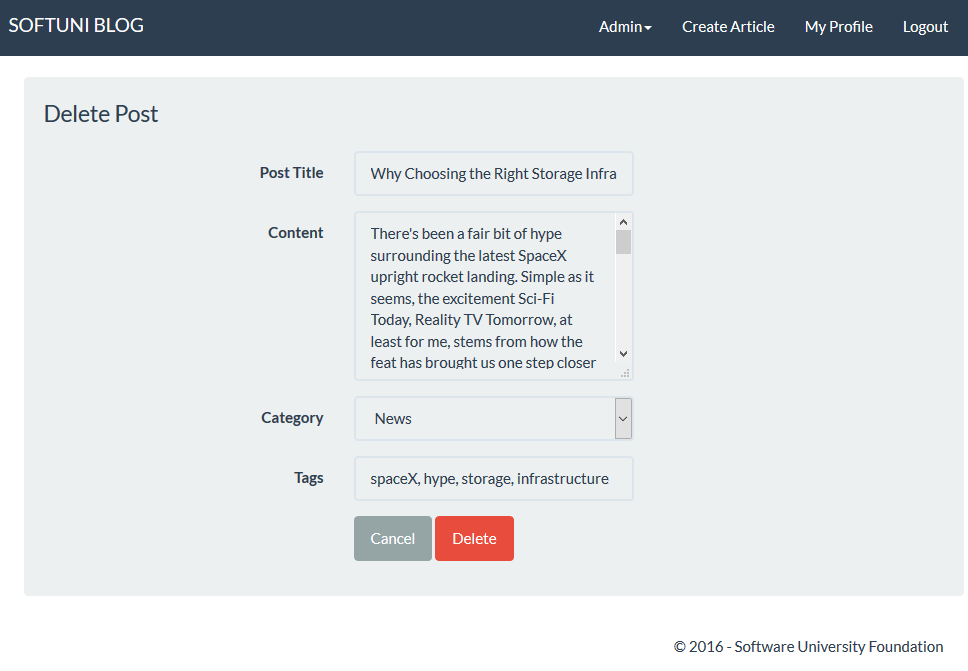


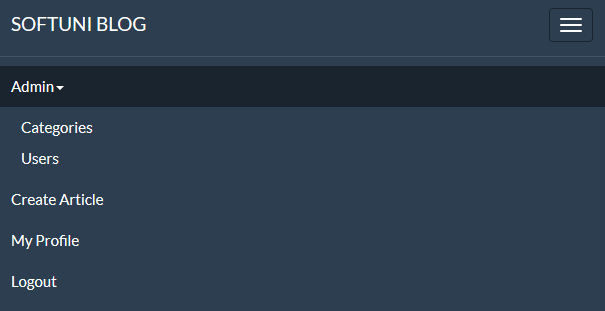


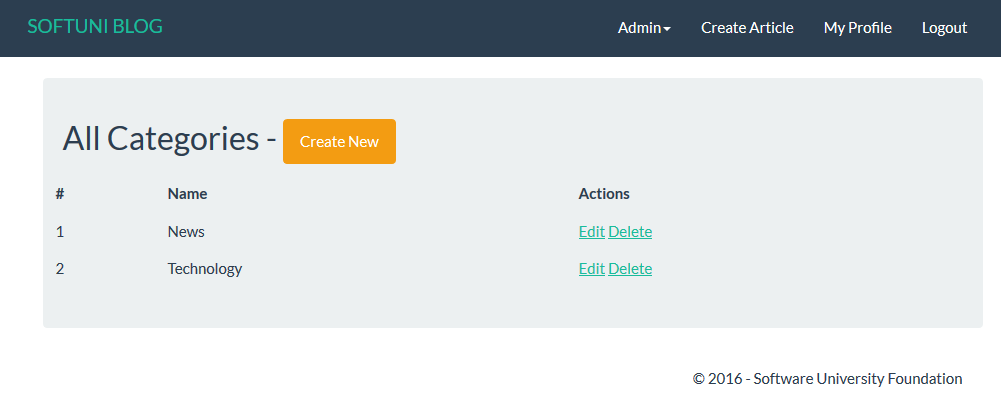


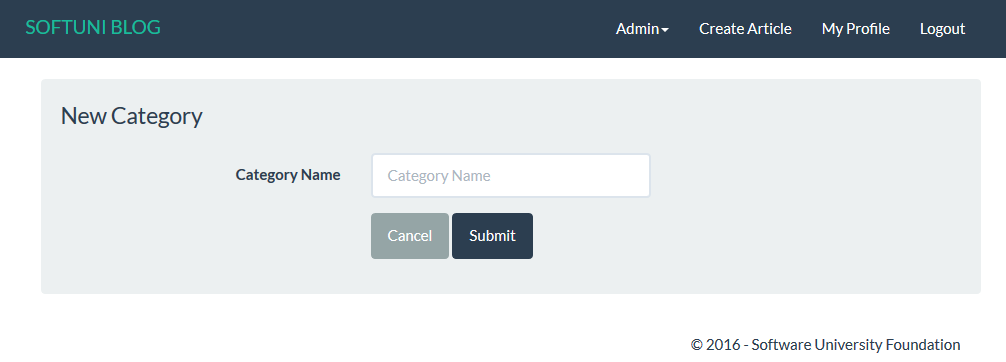


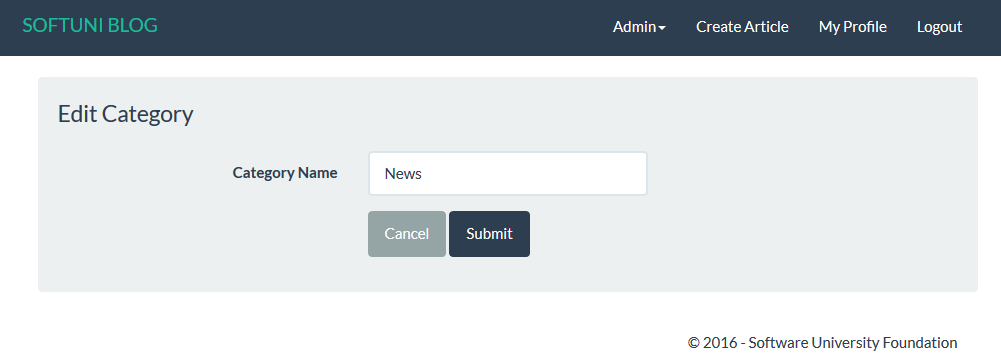


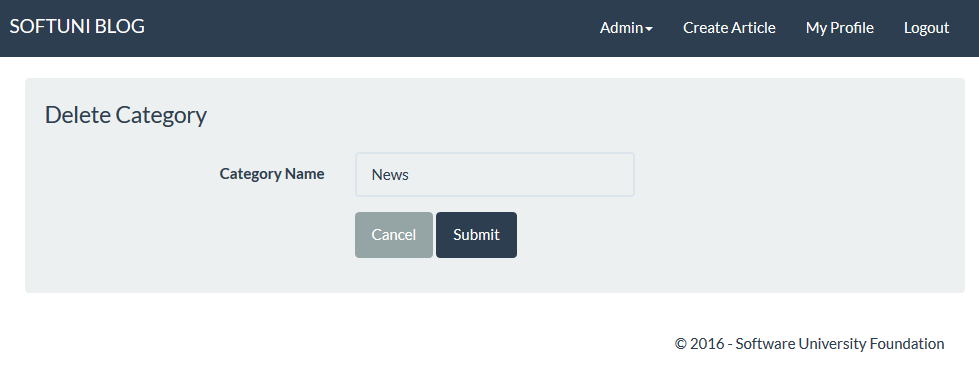


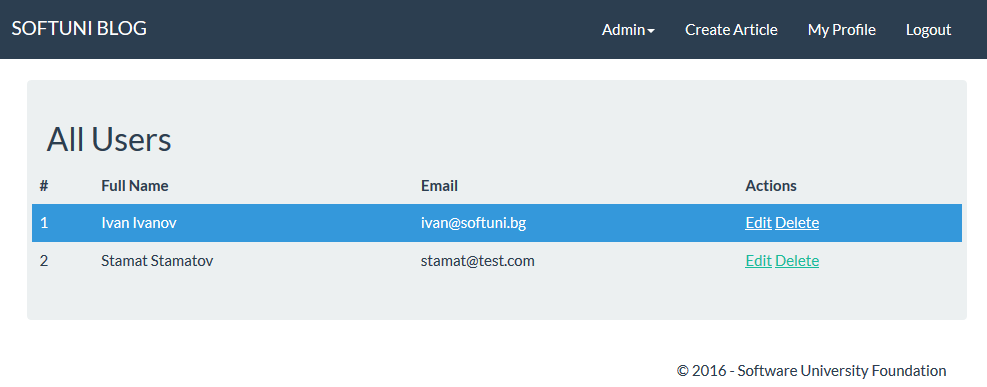


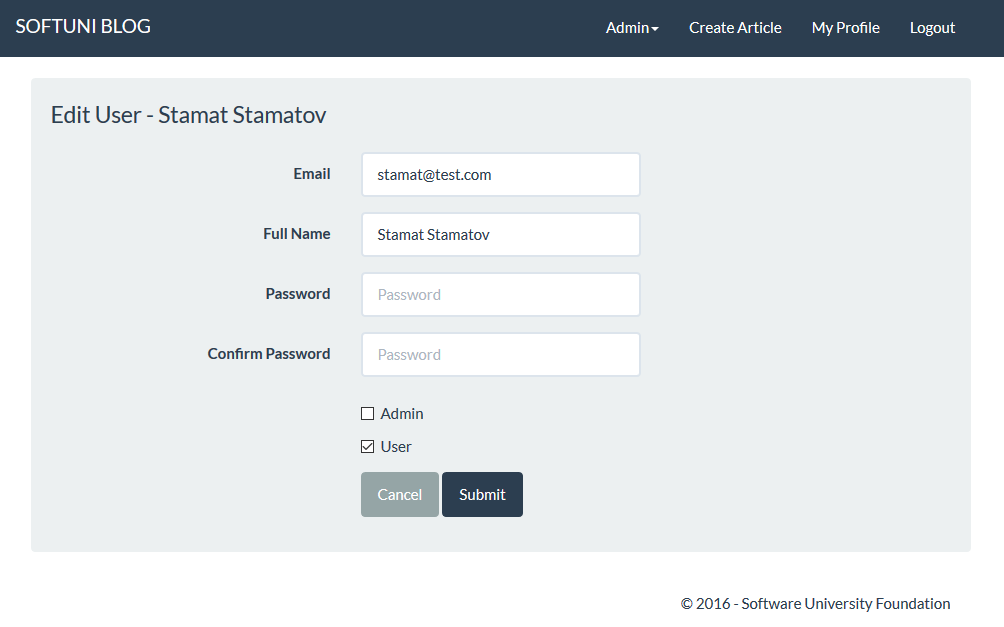


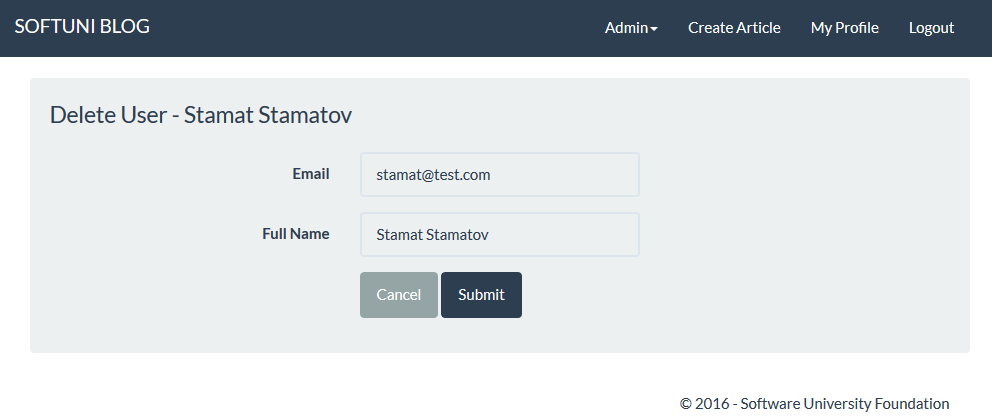


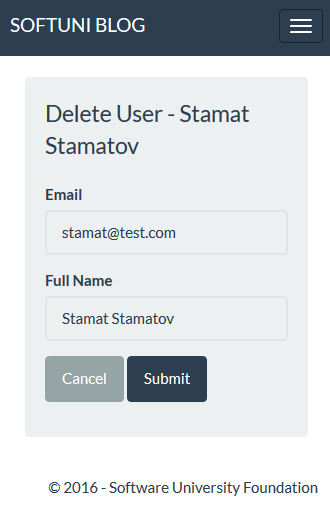












# Set Up JDK and IntelliJ Idea Configuration

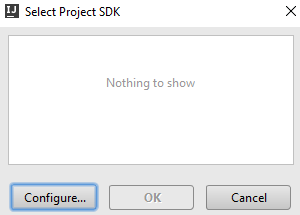
Before we start you need to download the [Java Development Kit](http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html), also known as **JDK**. Download the "**Java SE Development Kit 8u112**". After downloading it, install it **without changing the installation directory**. That will install it in the "**Program Files**" folder if you are on **Windows**.

## Using the Skeleton

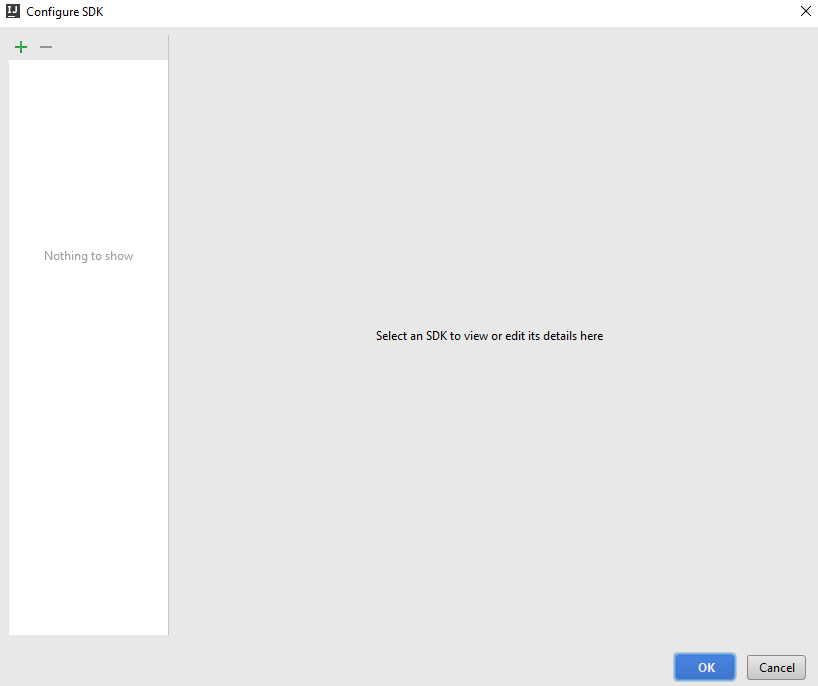
If you are using the skeleton and see something like this:

http://puu.sh/sjdFW/9e33c51c05.pnghttp://puu.sh/sjdFW/9e33c51c05.png

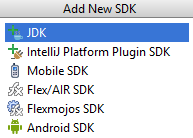
You should set-up the SDK. Click on "**Setup SDK**". You should see this screen:



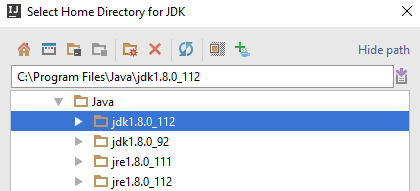
Click on "**Configure**" and see if you receive this screen:



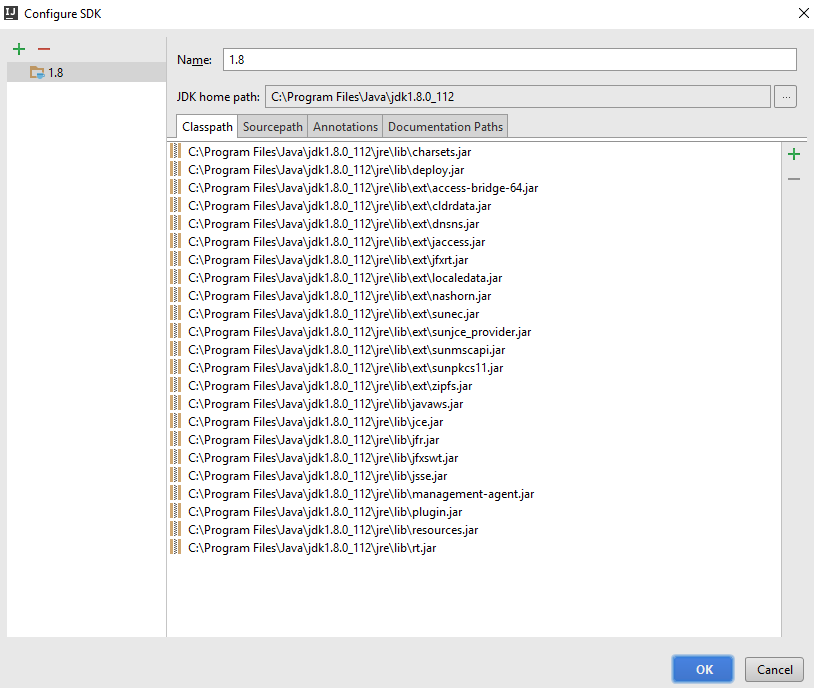
Click on the **green plus sign** in the top left corner of the window and choose **JDK**:



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



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



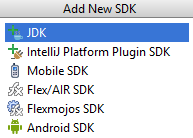
That is everything, your **JDK is now configured**.

## Creating New Project

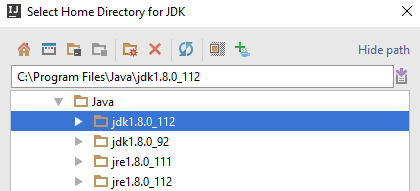
In **IntelliJ** **Idea** **Ultimate**, you should see this when you try to create a new project:

http://puu.sh/sjdth/612b88a470.png

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



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

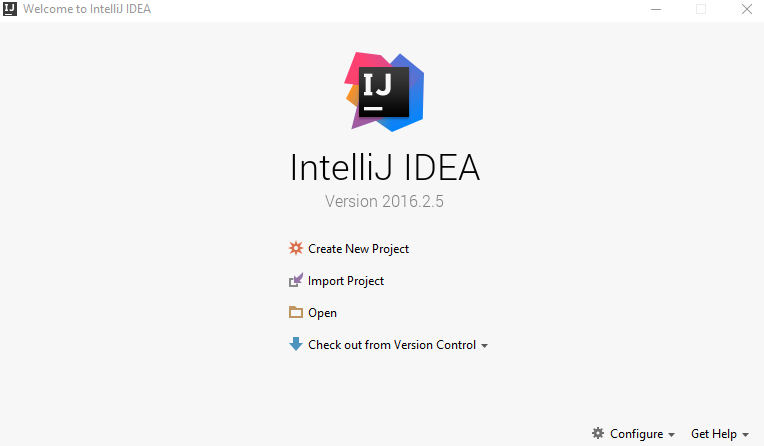


Click "**OK**" and you are **ready to create your project**.

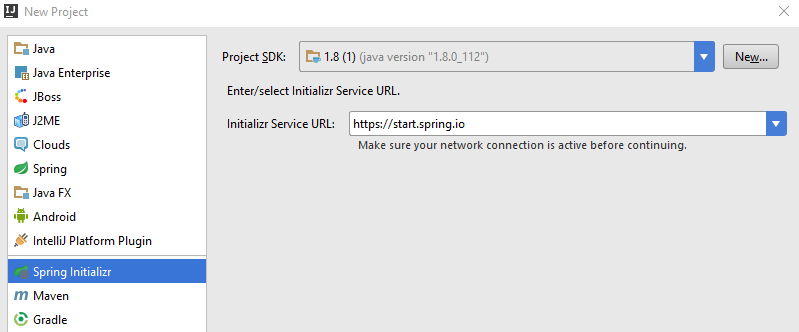
# Create Spring Project

## 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**":

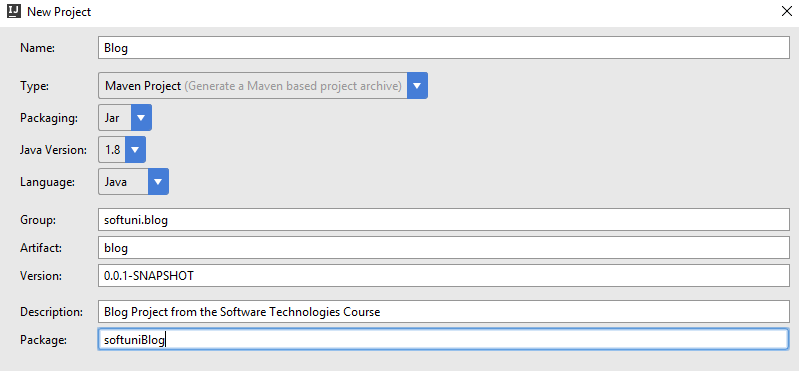


In the newly opened window, on the **left side**, you should see "**Spring Initializr**" as a **project type**:



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

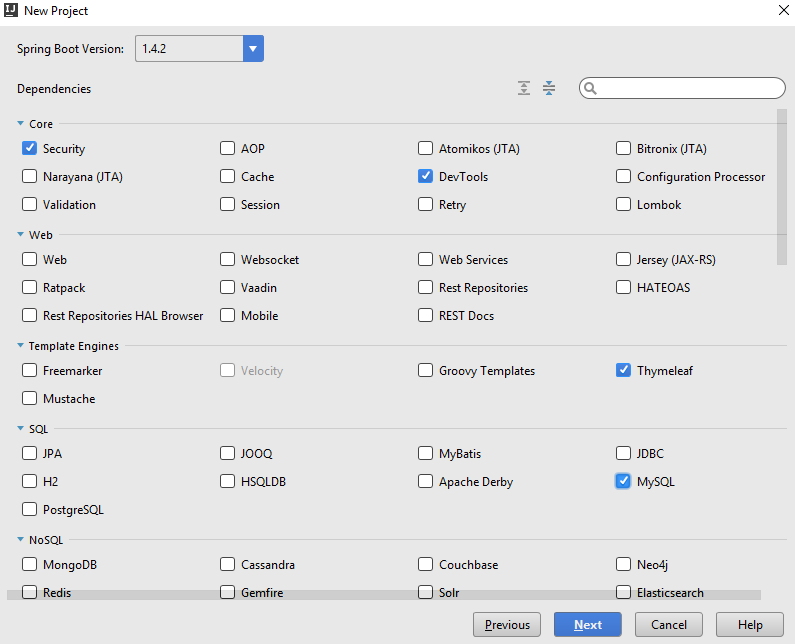
Click on "**Next**":



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:

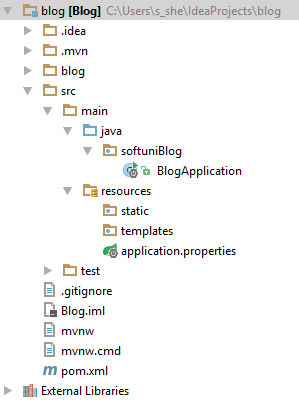
* **Security**
* **DevTools**
* **Thymeleaf**
* **MySQL**

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:



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

## Import Additional Dependecies

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:



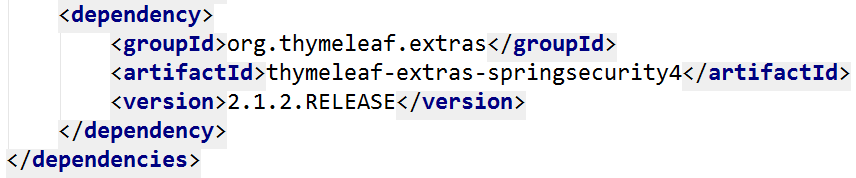
We want to **include** **additional dependency**, that will help us later. Before we continue, if you see the **following window**:



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**:

|  |
| --- |
| <**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**> |

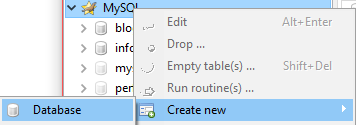
**Insert** this at the **bottom of the dependencies section**, and you have this:



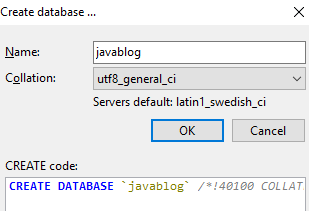
This will give us some **additional commands** that we are going to use in the **following chapters**.

## Create the Database Connection

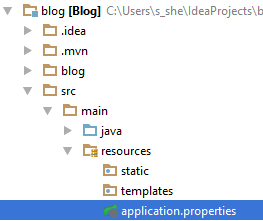
The last thing we are going to do in this chapter is create the DB connection. For database, we are going to use **MySQL**, the **same** **DB** we've used in the **PHP Blog**. That means that you will need to have **XAMPP** [installed](https://softuni.bg/downloads/svn/soft-tech/May-2016/Software-Technologies-June-2016/06.%20Software-Technologies-XAMPP/06.%20Software-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:



Use the following values:



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



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

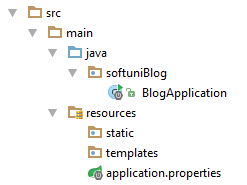
|  |
| --- |
| *# Database connection with the given database name* **spring.datasource.url** = **jdbc:mysql://localhost:3306/javablog** *# Username and password* **spring.datasource.username** = **root spring.datasource.password** =  *# Show or not log for each sql query* **spring.jpa.show-sql** = **true** *# 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** = **create** *# Naming strategy* **spring.jpa.hibernate.naming.strategy** = **org.hibernate.cfg.ImprovedNamingStrategy** *# Allows Hibernate to generate SQL optimized for a particular DBMS* **spring.jpa.properties.hibernate.dialect** = **org.hibernate.dialect.MySQL5Dialect** *#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.

# 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:



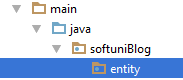
It contains main folder, which is then **separated into 2 different** folders. The first one is the "**java**" folder. This folder contains our **blog package**. **Inside** that **package,** we are going to **create** our **entities**, **controllers**, **configurations**, etc. The other folder is called "**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.

# Spring Security

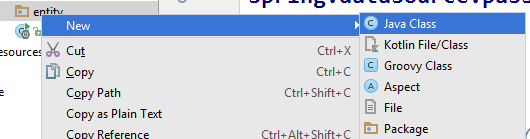
At the moment, you **cannot use your project**. Why? Because we've **imported the dependency for Spring**, that gives us the **authentication module**, but we haven’t configured it, yet. To do that we will create User **entity** using Hibernate. Then we are going to tell **Spring Security** what to use 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.

## Creating the User Entity

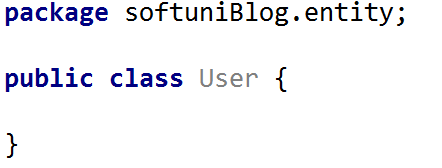
In our "java/softuniBlog" package create a new package called "entity":



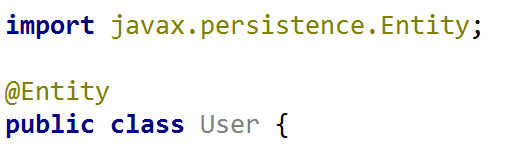
This package will contain all of our entities – **users**, **articles**, **roles**, **tags**, **categories**, etc. We will start by **creating new Java** **class** called "User":



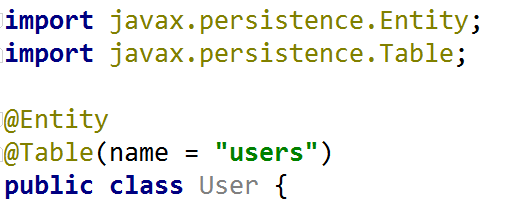
By default, it should look something similar to this:



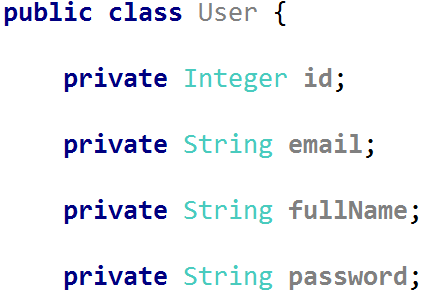
Let's start with the first annotation:



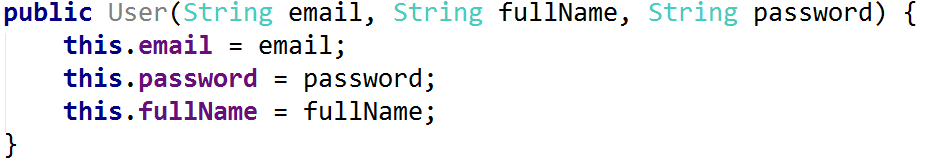
This [annotation](https://en.wikipedia.org/wiki/Java_annotation) means that the User class will become [entity](http://stackoverflow.com/questions/2550197/whats-the-difference-between-entity-and-class) that will get **saved into our database**. The next annotation is going to **define the table name**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



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



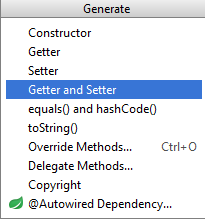
That is the information that we will keep in the database for our user. **ID**, which will be the **unique key**, **email**, **name** and **password**. The next thing that we are going to **create** is **constructor**, which should **help** us **with** the **user creation** later on:



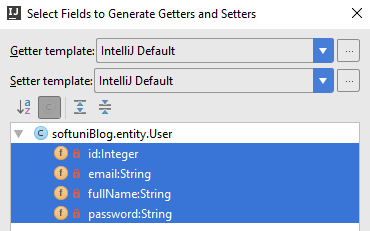
**Spring Security** will need **second constructor** in order to provide us with useful features. **It** **should be empty**:



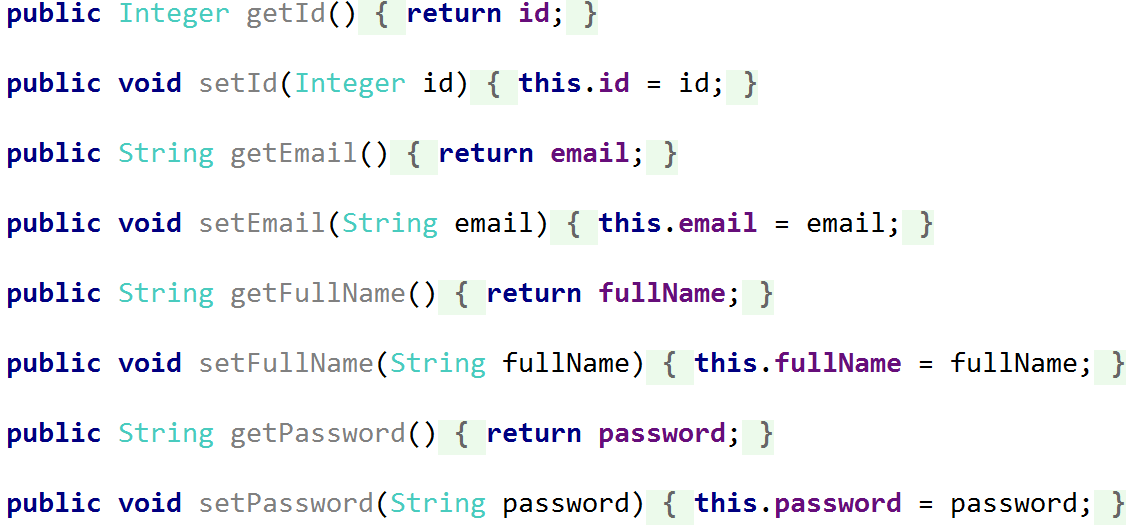
Now we need [getters and setters](http://java.about.com/od/workingwithobjects/a/accessormutator.htm). You should already be familiar with them. If you are curious why are we doing that, you can read more [here](https://www.tutorialspoint.com/java/java_encapsulation.htm). There is a **simple way to create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



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

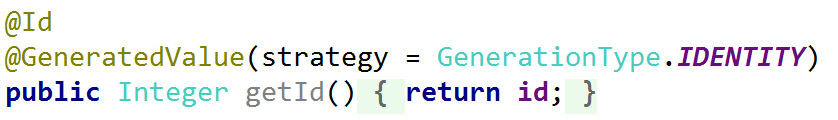


When you **click** "**OK**", you should **receive this code**:



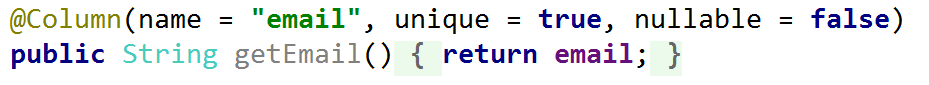
It might be **formatted in a different way**, but the **result should be the same**.

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:

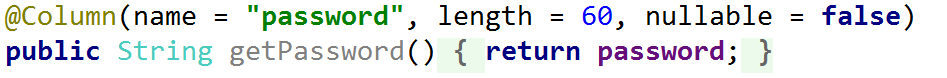


The "@Id" annotation tells [Hibernate](http://hibernate.org/orm/) that **this field will be the primary key** for our **database**. The second annotation makes the **field generated automatically**, without us doing anything. The next annotations are really similar.

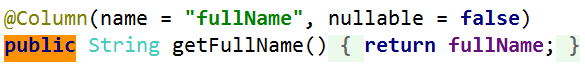
**Email**:



**Password**:



**Name**:

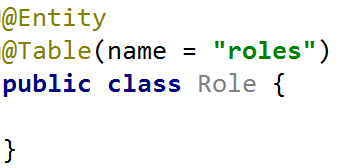


In all three of them, we are **defining** the **column name** and we are **making them non-nullable**. That means they can't contain null value. For the **password** field, we are limiting the **max length to 60 symbols**. Finally, we are telling **Hibernate** that the **Email** **should** **be** **unique**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

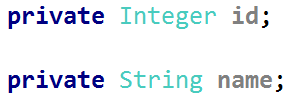
Our user is **almost modelled**. But we need to **give him a role**. In order to do that, we need to **create new entity**.

## Creating the Role Entity

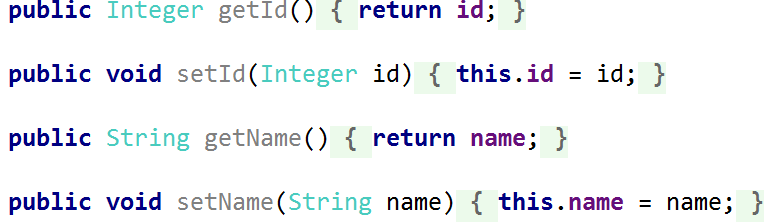
Create new class in the "entity" package that will be called "Role" and should have the **following annotations**:



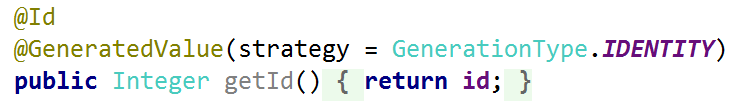
The next thing is the private fields:



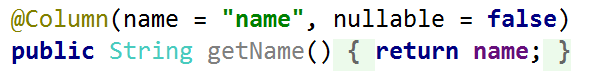
The name will be in the following format "ROLE\_\*". Then, we have to create the getters and setters:



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



And the **name shouldn't** be null:



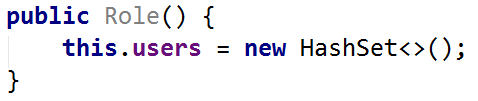
**This** is the Role **entity**. Now we need to create the relationship between the User and the Role.

## 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:

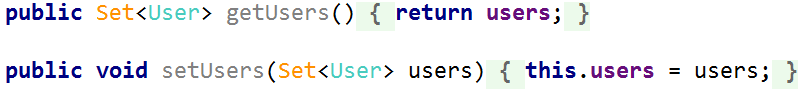


To use it, **similar to every other collection**, we need to **initialize** it using **constructor**:

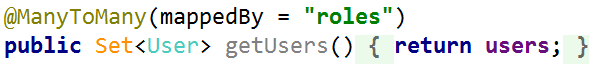


You can read more about the HashSet [here](https://docs.oracle.com/javase/7/docs/api/java/util/HashSet.html).

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



And the annotation will be this:



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

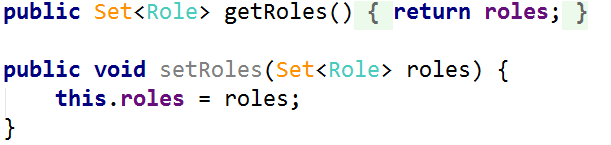
We should jump to the User entity now.

## Creating the User-Role Relation

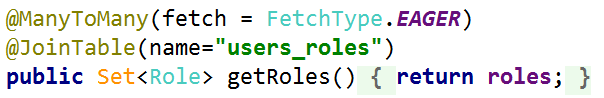
As we've said, we should create private field in the User entity:



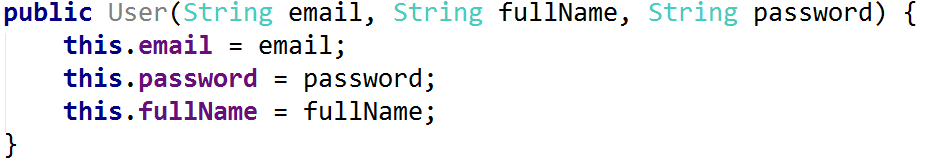
This will keep the **unique roles each user** **has**. Create the **getter** and **setter** now:



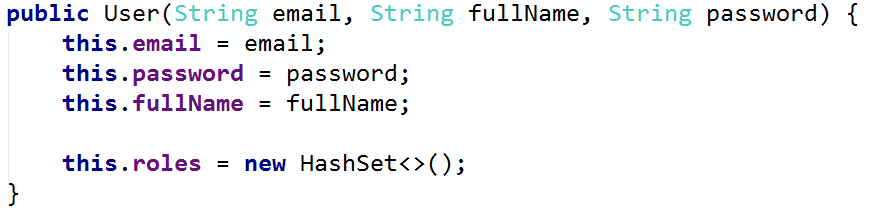
Now you need to **add** the following **annotations**:



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:



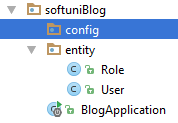
We are not **assigning default role** when we **create new user**. That's why we need to change the constructor like that:



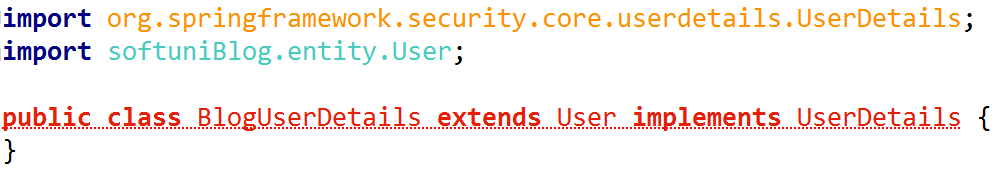
Here we are **initializing our collection**, saving us problems later on.

## Creating UserDetails Implementation

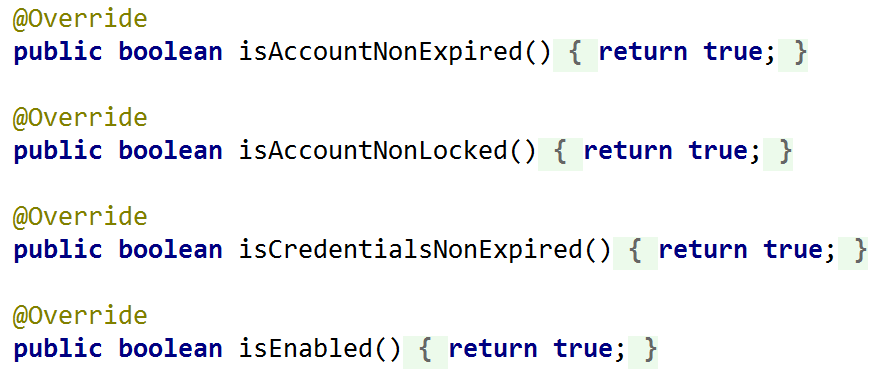
In order to use the **built-in functionality** from **Spring Security** we need to create a **new class**, that will make sure that we are **creating the users** using **the right way**. In the "softuniBlog" package, create a new package called "config":



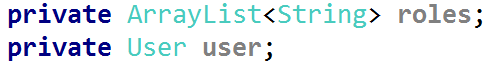
Inside of it create a **new class** called "BlogUserDetails". Then change it like that:



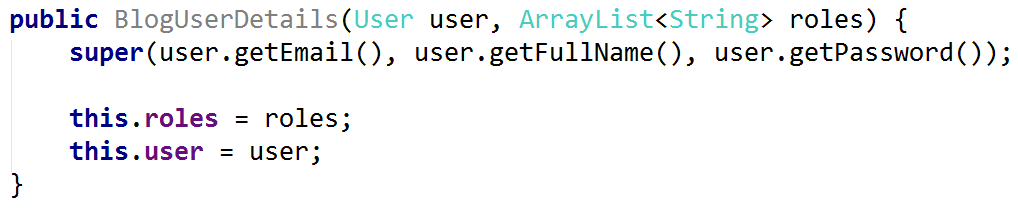
**Don't worry** that everything goes **red**, we will take care of that. First, add the following code inside the class:



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**:

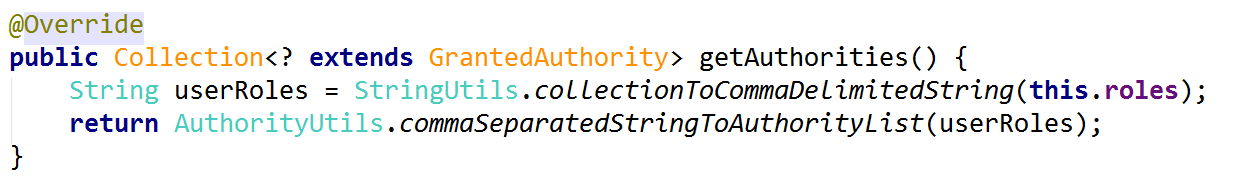


The User is our **entity type** User and the **roles** is a simple **list** collection. And now we need to **create a constructor** for this class. It should look like that:

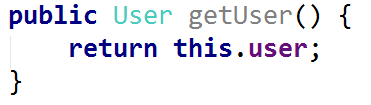


As you can see we're setting our roles and user fields using the parameters we are taking in the constructor. However, we are doing something else as well. We are using some sort of method called "super()". This is way **more complicated** to explain than it looks so we'll leave it for your future courses (**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:



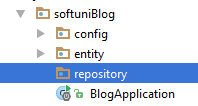
This will get our **roles** (that we currently keep as strings) and **join** **them** into one string. Then it will return collection of [authorities](http://docs.spring.io/spring-security/site/docs/3.0.x/reference/authz-arch.html). The **authorities** in **Spring** are the things we call "**roles**"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



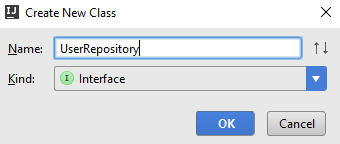
That is all, now we need to **find a way** to get our **users** and **roles** from the **database**.

## 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":



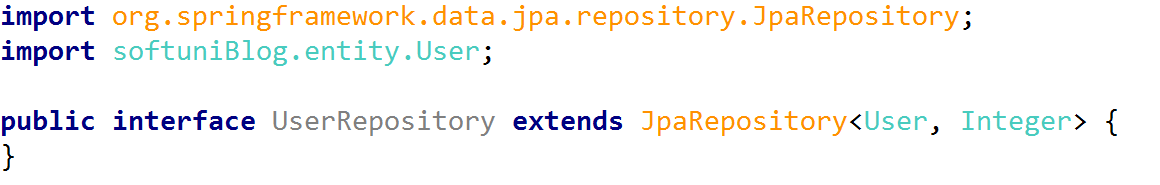
Now we will create UserRepository:



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:



We should quickly change it to:



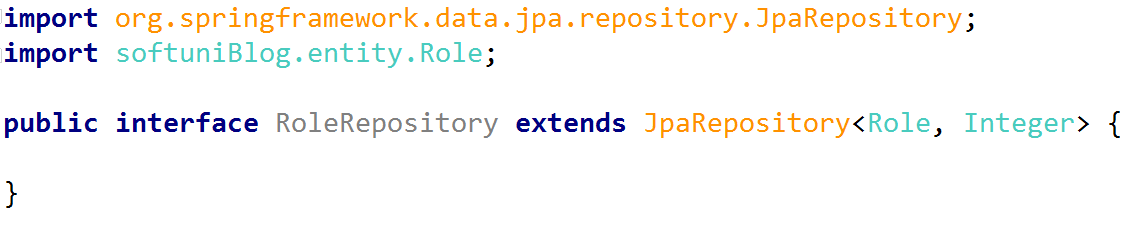
This will give us **some methods** that we are going to use later on in our blog, but for now we want to **create the following method** in our **repository**:



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.

## Creating Role Repository

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



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

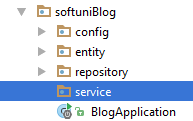


This will get us the **role** with **given name**. It is almost the same as the method in the UserRepository, but the **criteria** and **return types** are **different**.

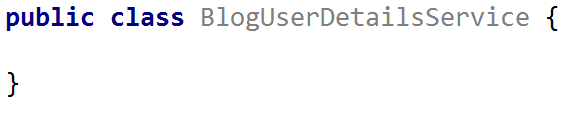
We are ready with our repositories for now.

## Creating User Service

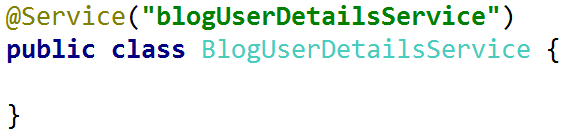
The next thing we need to implements is the so called "userService". It is used to get **user from the database** and transform it to **Spring Security** User. Create a new package called "service":



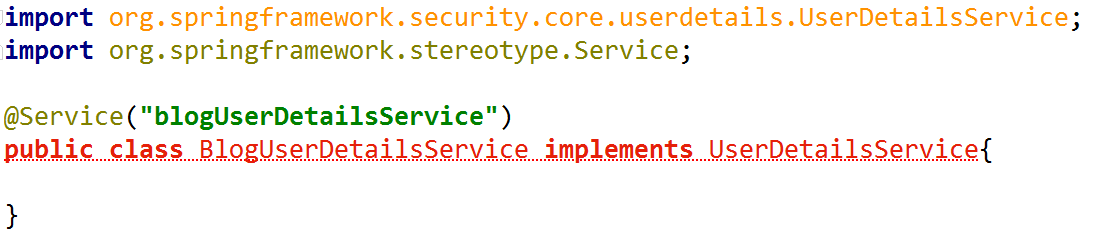
Create a new class called BlogUserDetailsService:



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



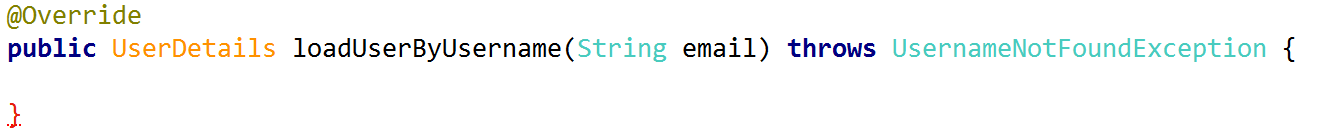
This will **give** our **service** a **name**. Now, we need to change the class like that:



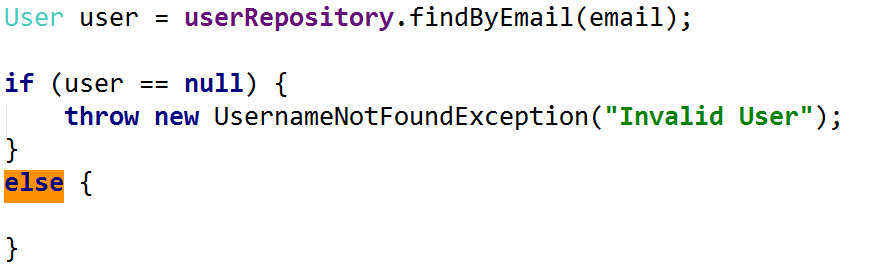
Again, everything **becomes red**, but that's nothing to worry about. We will start by creating a **private** **field** and **constructor** to initialize it:



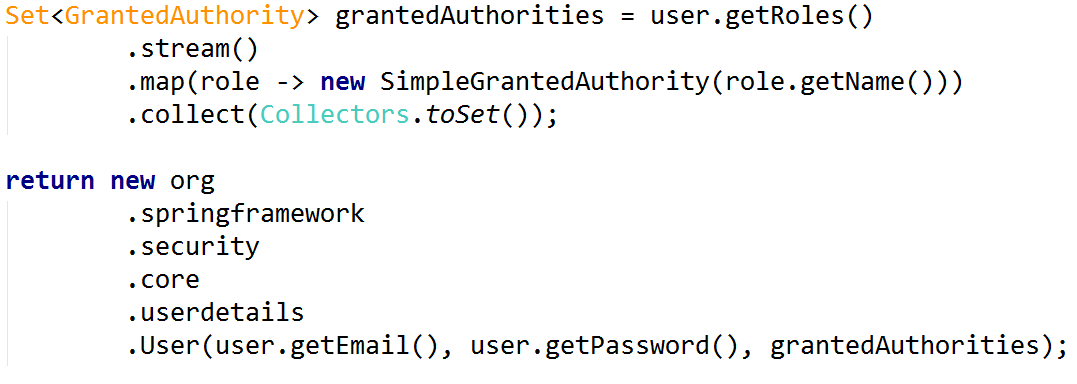
This private field has the "final" keyword, which means that we will **not be able to change it** after **initialization**. Now we need to **override** one of the **base class methods**:



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**:



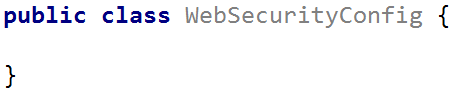
The case, where the user exist is more interesting:



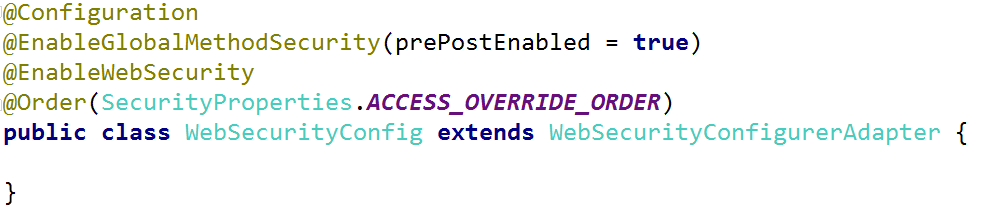
Here we get all of the **user roles** and **create a collection of authorities**. Then we create a new **Spring Security** User with the given **email**, **password** and **authorities**. This is **everything** for **our** **service**, but we are **not** **done**, yet.

## 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:



Now it will get really messy, really quick:



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)

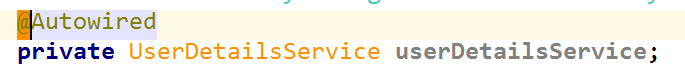
[@Order](http://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/core/annotation/Order.html)

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**:



We will need to create the following annotation for it:

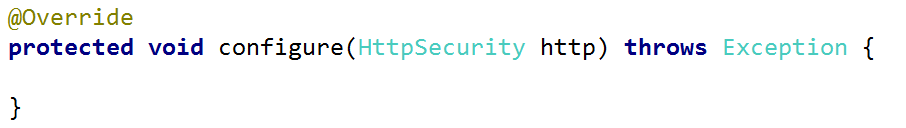


Using that **annotation**, we are telling our class to **initialize the field automatically**. The next thing that we want **Spring** to do **automatically** is to **change the default password encoder** to BCrypt:

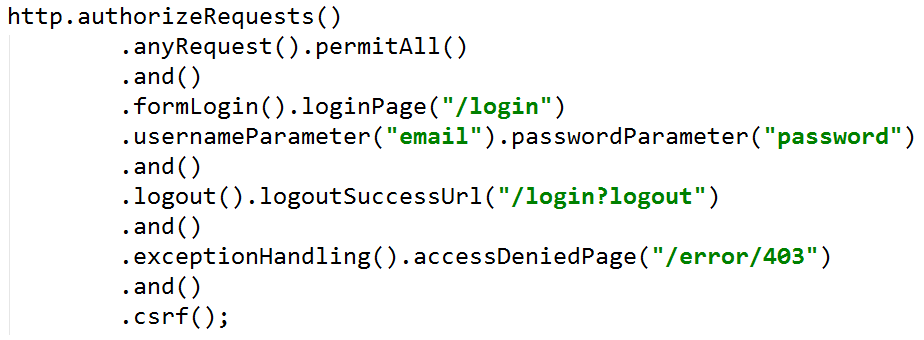


Here we are setting the default userDetailsService to use our **field** and we are setting the passwordEncoder to a **more secure** one.

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



This method is going to define **what permissions are needed** to **access** our **blog**. Write the following code:



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**.

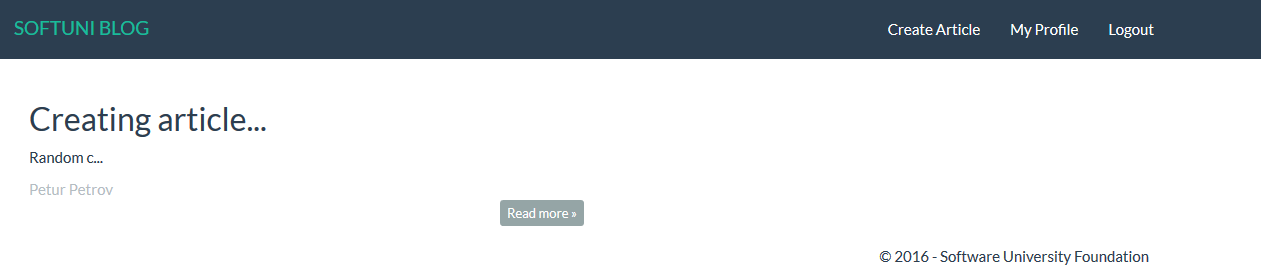
It’s the beginning of the end guys…

## 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:



**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**:



Or



The other two parts as you can see for **logged users** and **admins**. We **won't create** the **admin functionality now**, so we'll leave that. 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):

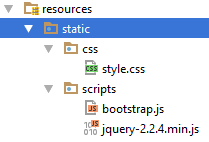


For every link in our navigation menu we are using "sec:authorize". This is coming from **Thymeleaf Security** and gives us the ability to **check** **if** **someone** is **logged in**, **has a specific role** or is just a **guest**.

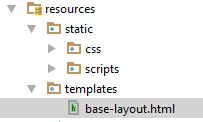
Another section is the "**footer**". It represents this:



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 js and css folders we gave you:



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:



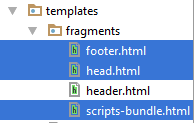
Now, create a **new directory** in the templates package called "fragments". It will contain the fragments(sections), we've talked earlier about:



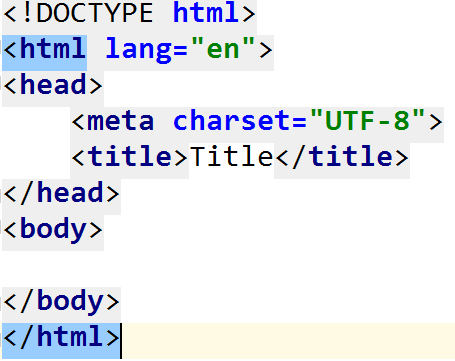
Inside of it, create a **new HTML file** called "header":



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



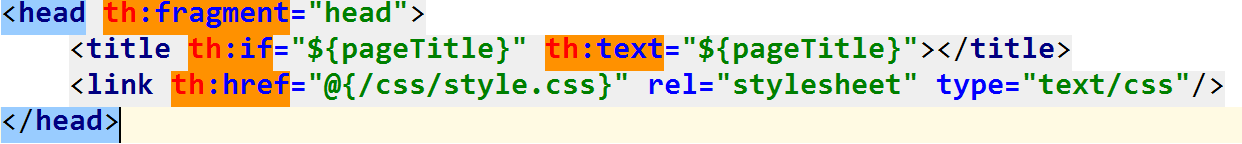
Each of the HTMLs we've created should look like this at the moment:



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

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



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":

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

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

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

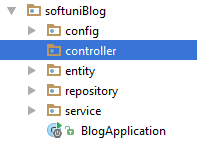
Its job is to import the 2 **JavaScript** files we are going to use. As you can see, all of our fragments have a specific role in our design, but let's combine them together in our "base-layout.html":

|  |
| --- |
| <!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**> |

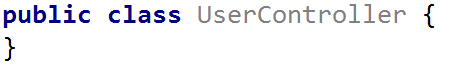
This will be the only complete and valid HTML file. However, it is not just a normal HTML file. It imports Thymeleaf and Thymeleaf Security, which will be of great use. As you can see our **head** **tag** uses something called "th:include". This will replace our current <head> tag with the html file 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**.

## Creating User Controller

Create a new package called "controller":



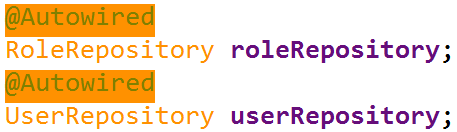
Now create new class called UserController:



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:

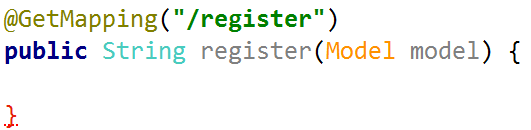


That way we are telling **Spring** that this class can **define routes** and that will **take care of actions** related with **our entities**. First let's create private fields for our repositories:

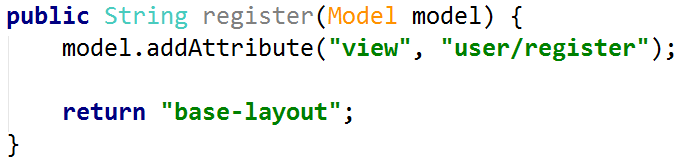


We are using the "@Autowired" annotation again, to tell **Spring** to **initialize those fields**.

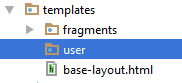
First, we need to be able to create users. Create the following function:



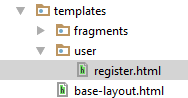
We are using the "@GetMapping" annotation. This annotation defines that the type of [request](http://www.w3schools.com/TAGS/ref_httpmethods.asp) we are going to **process in our method** is "**GET**". That means that if **someone sends data** (i.e. user data), this method won't be called. **This** **method** 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**:



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":



Now create **new HTML file** called register:



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

|  |
| --- |
| <**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**> |

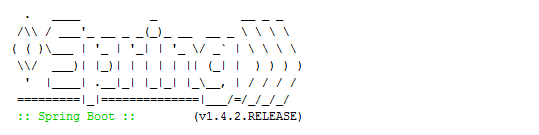
You should be familiar with this code, so **let's see if it works**.

## Starting the Project for the First Time

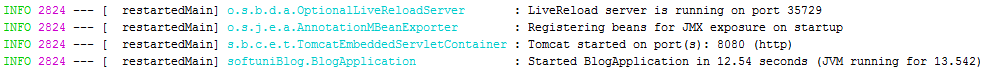
In the **top-right side** of **IntelliJ Idea** you should see this:



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



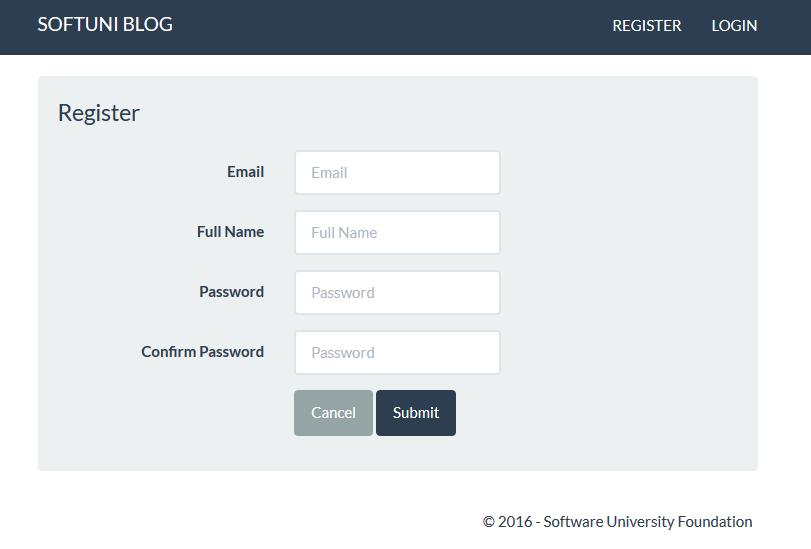
Wait until you see this:



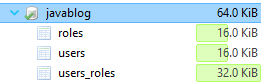
The last row means that the **server is running**. Open <http://127.0.0.1:8080/> and see what you get. You should **receive this error message**:



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



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:



Stop the blog using the  icon.

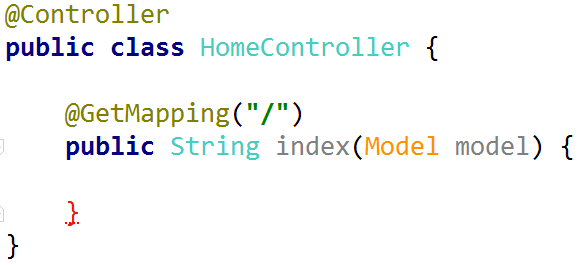
**Before we continue** with the **user register**, we should create a **home view** to fix the error we've received earlier.

## Creating Home Contoller

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



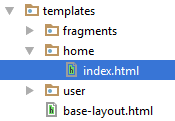
This controller will **list all of our articles** later on, but for now it will just **return an empty page**. Create a new function called "index":



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



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



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

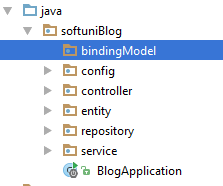
If we **start the application** and **visit the home page**, we should see this:



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.

## Finishing the User Register

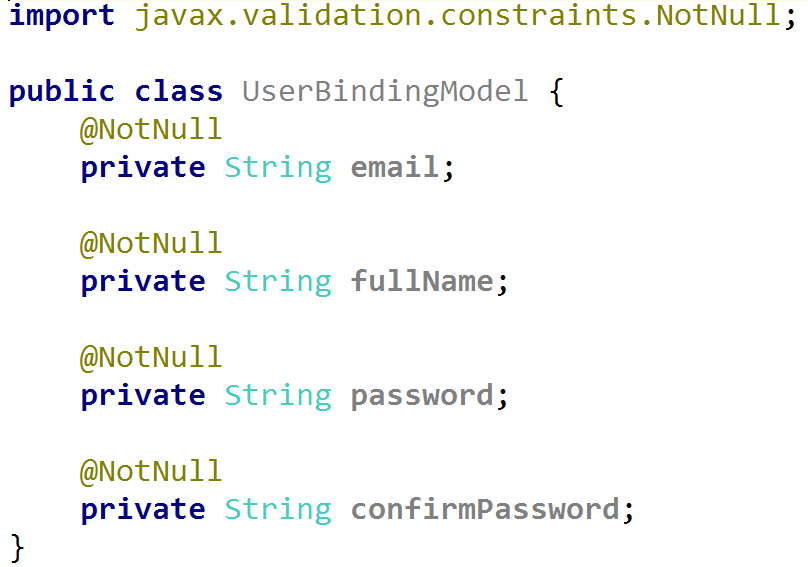
Before we go back to the UserController we need to do something else. In the softuniBlog package create a new package called "bindingModel":



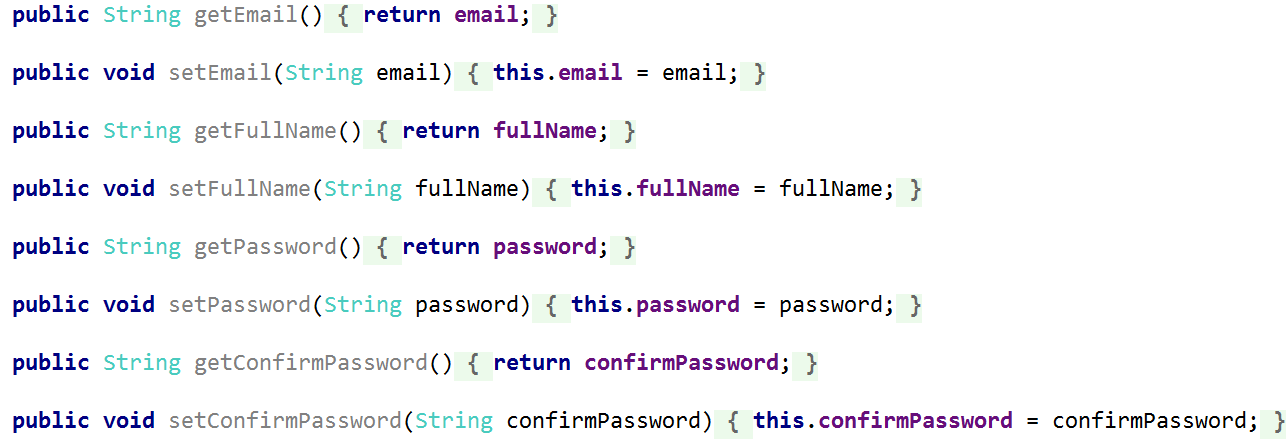
Create a new class called UserBindingModel:



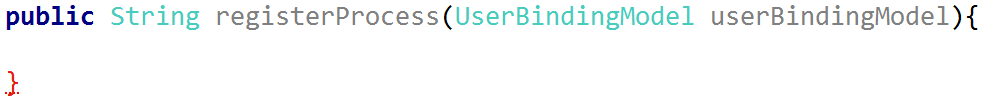
This class will **take the data** **from** our **register** **form** and we will **use** **it** to **create a new user**. In order to do that the binding model should contain **the exact fields** that **our form has**. Here is how we should start:



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:



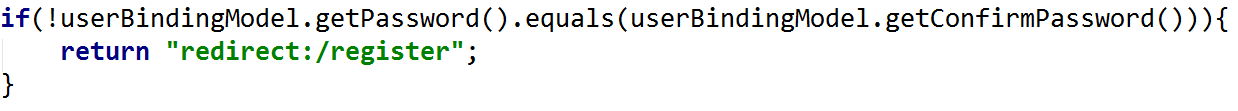
Let's put this model to good use. Go back to the UserController. Create a new method called "registerProcess":



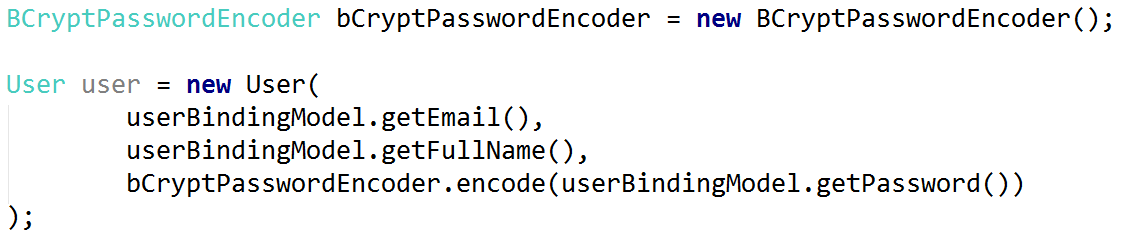
This method will have the hard job to **create a new user**. **Spring** will automatically map the form data to our binding model. The only thing that we need to do is **define routing**:



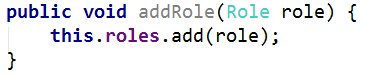
The PostMapping annotation corresponds to the form method. It means that we will **receive data** from somewhere. The first we want to do in our method is to **check if the passwords match**. If they don't we will **ignore the form submission**:



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:



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:



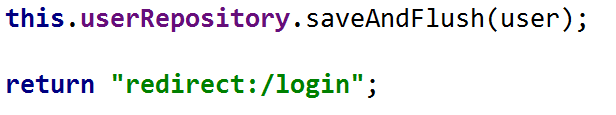
Back in our UserController we can't use that method straight away. First, we need to **get the role from our database**:



Now we can **add it to our user**:



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



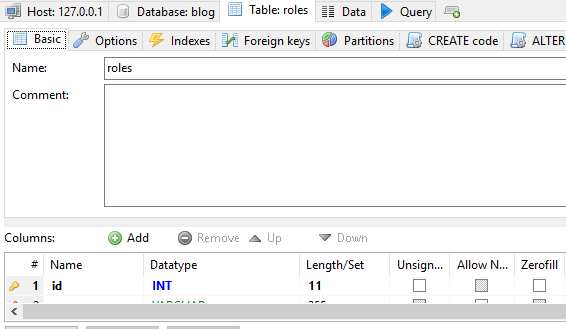
Before we test if it works, we want to change something. Maybe you have noticed that **every time** you start the application it **drops the old database** and **creates new one**. **We don't want** **that**. Find your application.properties file. Edit the following line:



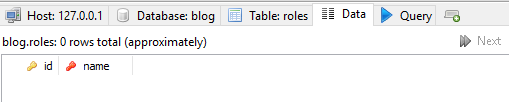
To



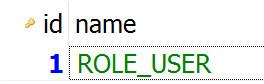
Start the application now. Before you open the application, let's create a new role into the database. Open **HeidiSQL**. **Double-click** on the roles table. The main screen should change to this:



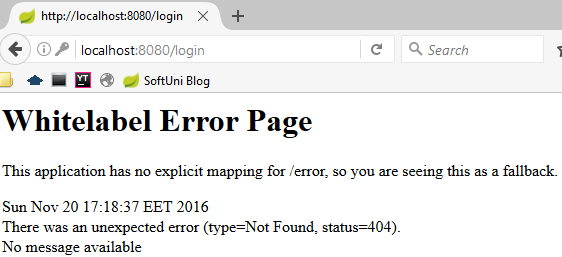
In the navigation, you will see "**Data**" tab. Open it and you should see this:



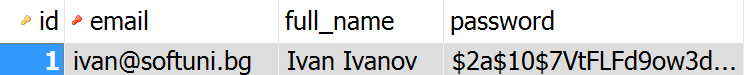
Our database is empty. Click on the "**green plus**" in the **main toolbar**  and you will **be able to enter data** in a **new row**. **Create one role** called "**ROLE\_USER**". It should look like that:



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



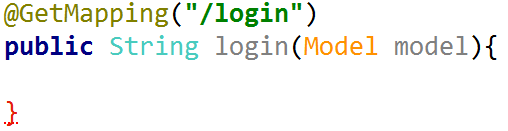
**This is an error yes**, but take a **look at the URL**. We are trying to access "/login" that **doesn't exist**. That means that **our user should be created**. Take a look at the users table in the **database**:



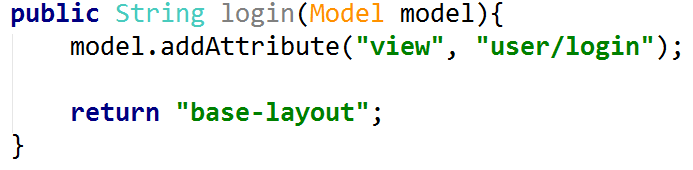
That is **successful registration**! Now we can **create the login**.

## Implement User Login

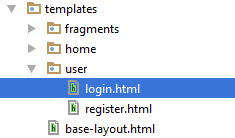
To create a login, we need **only 2 things**. A **method** and a **view**. **Spring Security** **will take care of the rest**. Let's start with the method in our UserController:



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



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



You should delete the existing code and use the following:

|  |
| --- |
| <**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**> |

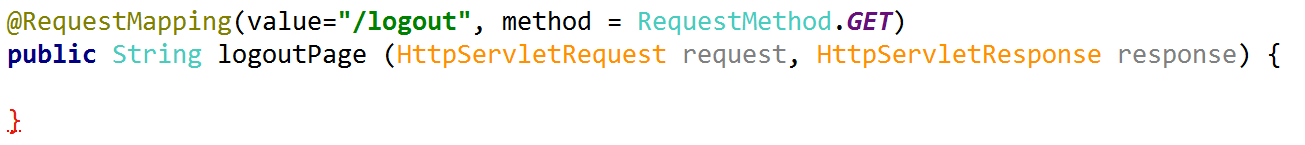
This should be everything. **Start the blog** and **try to login** with the user you've created previously. You should have this if everything is working:



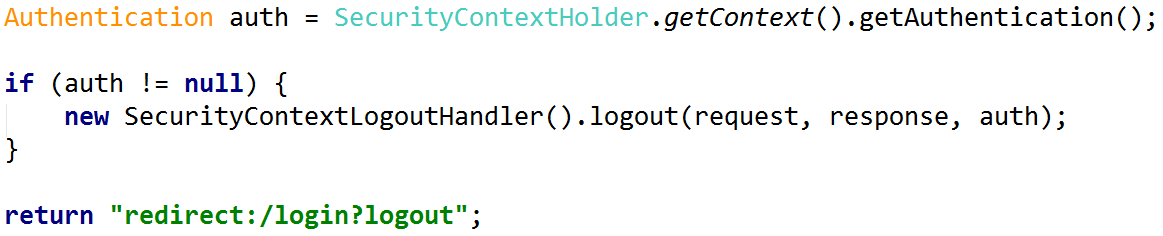
**The login works**. If you try to **open the user profile** or try **logout** you should **receive errors**. Those are our next targets.

## Implement User Logout

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



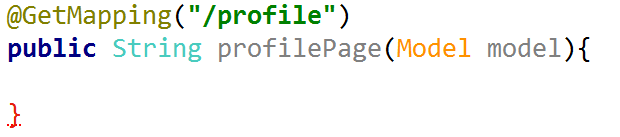
First of all, we are using the "@RequestMapping" annotation. This annotation combines "GET" and "POST" requests (**not** **only**) and we need to specify that we are interested in the "GET" requests only. This method should have the following code:



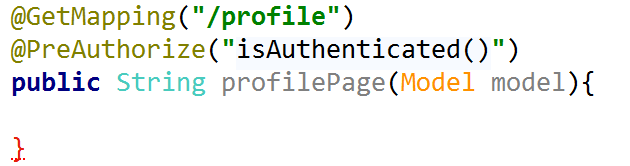
It **checks if there is logged in user** and if there is, it simply **tells the authentication module to logout the user**. Then it **redirects to the login** page again. **The logout is ready**.

## Create User Profile Page

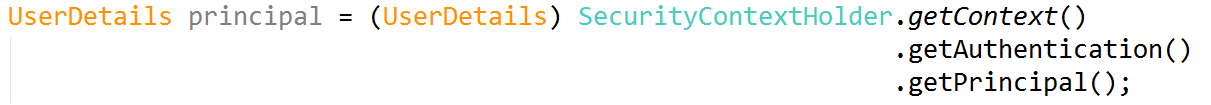
One final thing to create is the **user profile page**. It should give **basic info about the user**. Start by creating a new method in the UserController called "profilePage":



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**:



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:



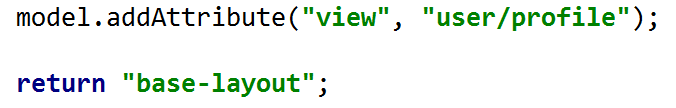
This will give us only the **basic properties of our user**. That means only username (email in our case), roles and password. We can use it to **extract the current user** from the database:



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



Now we need to **return the view**:



Overall the code should look like this:



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

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



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**! ☺

# Create Articles

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

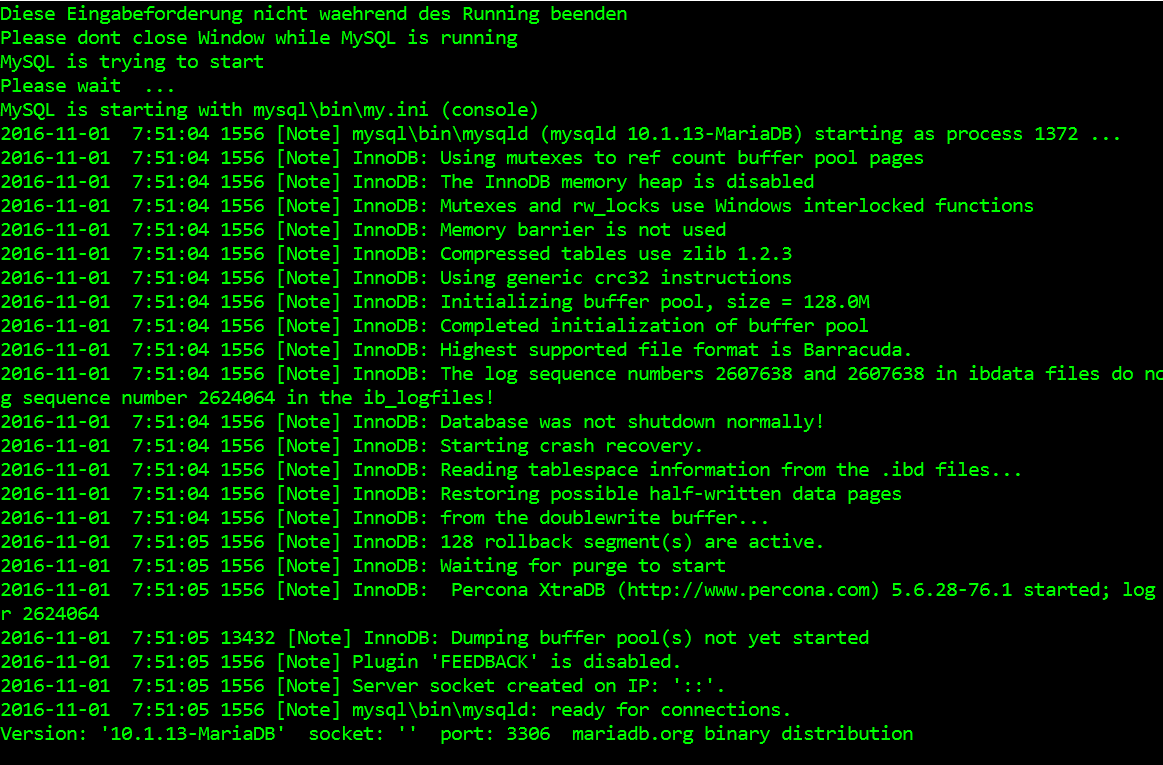
## Start MySQL

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

If you are still reading:

Download the [project skeleton](https://softuni.bg/downloads/svn/soft-tech/Sep-2016/Software-Technologies-Oct-2016/09.%20Java-Blog-Basic-Functionality/Java%20Skeleton.zip), extract it in a shortest path you can make, e.g. in **c:\project**.

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



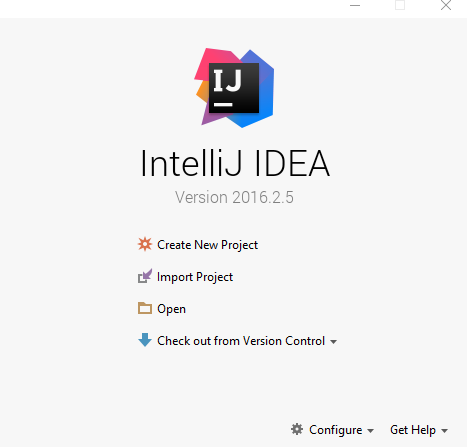
That’s it, MySQL is running. When you decide to stop working on the blog, just close the terminal and run the **mysql\_stop.bat** file.

## Open the Project

**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 “**Open**”:



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



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

## Create the User Role

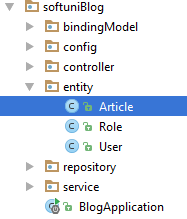
Using **HeidiSQL** import new Role into the **database** with **name** "ROLE\_USER". If you don't know how to do it, refer to [chapter V part 14](#_Finishing_the_User).

## Start the project

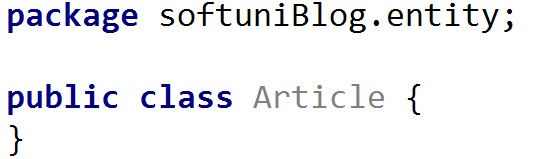
You can find how to **start the project** in [chapter V part 12](#_Starting_the_Project).

## Article Entity

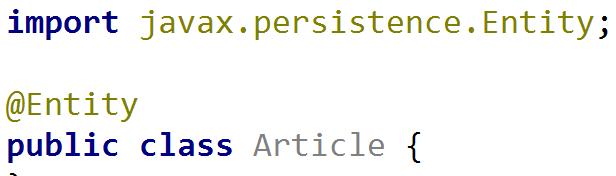
It's time to create our first entity. We are using [Hibernate](http://hibernate.org/orm/) for **ORM**. That means we are going to define our entities with [annotations](https://docs.jboss.org/hibernate/stable/annotations/reference/en/html/entity.html). In the src/main/java/softuniBlog **package** you 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":



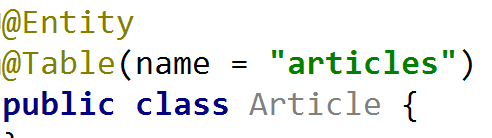
The file should look like this:



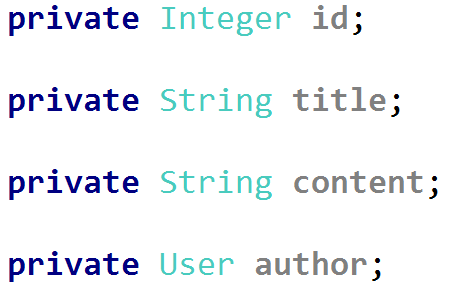
Now we need to tell **Hibernate** that this is an entity:



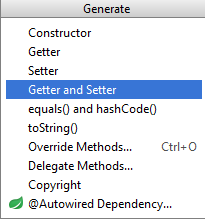
Now that our class is an entity we need to give our database **proper table name**:



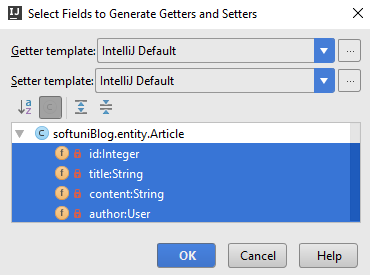
The next important thing is the **table columns**. We need columns for id, title, content and author. Create the following private fields:



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:



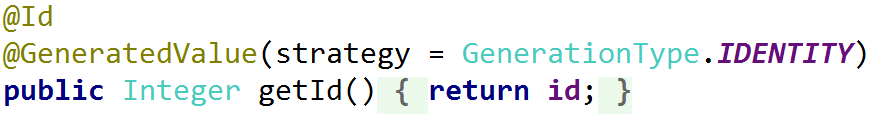
Choosing the "**Getter and Setter**" option will **open new window**. You should select **all private fields** from there:



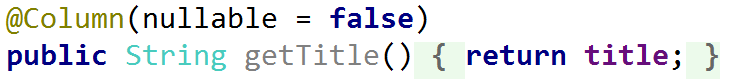
When you **click** "**OK**", you should **receive this code**:



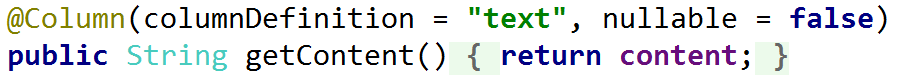
It might be **formatted in a different way**, but the **result should be the same**. Now we can **explain each column to the database**. We are going to **place our annotations on the getters**. The first one is the id **getter**:



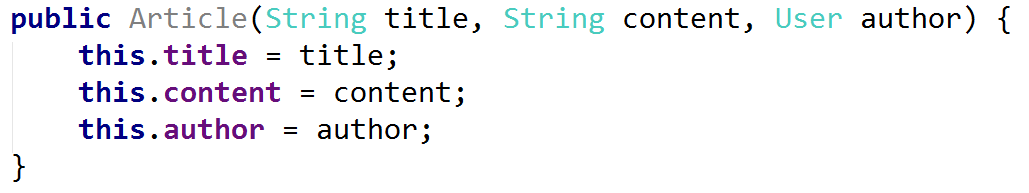
The **id** **column** will be the **primary key** in our database and as such we need to use the "@Id" annotation. The "@GeneratedValue" annotation tells **Hibernate** that the database should **generate the values automatically**. The next getter is the title:



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:



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**:



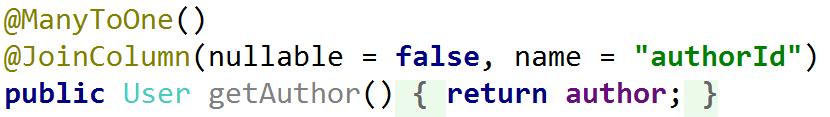
We will **use** **this** **constructor** to **create articles** easily. However, we need to create another **empty** **constructor** for **Hibernate**:



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

## 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:



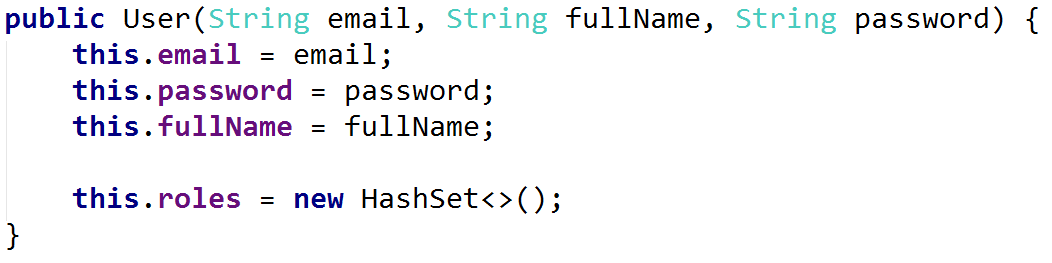
The first one is the “ManyToOne” annotation. Many to one relationship represents [OneToMany](https://en.wikipedia.org/wiki/One-to-many_(data_model)) relationship from the side of the “many”. Because we are working with the Article entity, we are telling **Hibernate** that **many of our articles** will 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.

## User-Article Relation

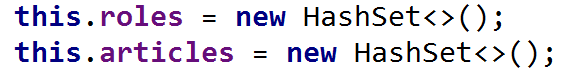
In the User entity, we need to **create a field**, which will keep **all articles** created by a given user:



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:

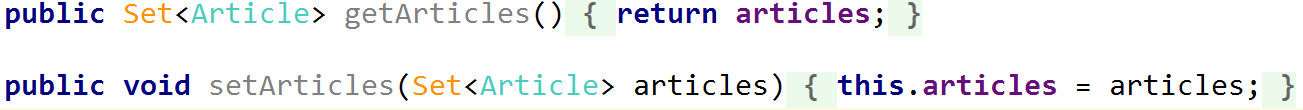


Add another line that will **initialize the articles** collection:

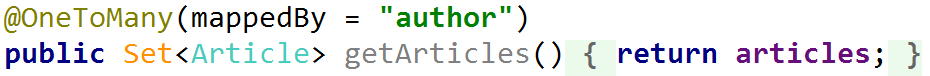


Here we are telling **Java** that our specific type of Set should be the [HashSet](https://docs.oracle.com/javase/7/docs/api/java/util/HashSet.html). The HashSet collection gives us **faster operations** over our collection, but it **doesn't keep the order** of elements. That means that we are going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



Let's add the annotation for our relation:

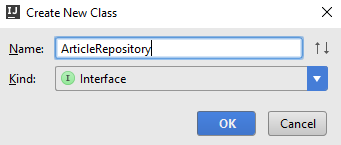


It is pretty simple. It means that **Hibernate** should go to our Article entity and find the "author" field that we've created earlier. Then it will get the **properties** of the relation from there and use them as a base when creating the [foreign key](https://dev.mysql.com/doc/refman/5.6/en/create-table-foreign-keys.html)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

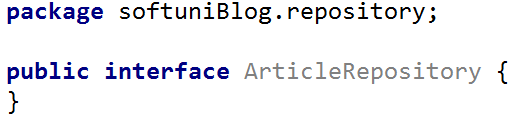
Our relation is ready now.

## Create Article Repository

If you want to read more about **repositories**, you can do it in [chapter V part 6](#_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 working with given entity** in the database. In the "repository" package create a **new java class** called "ArticleRepository" of type **interface**:



We should have this:



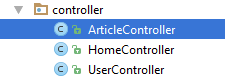
The only thing we want to do is tell **Spring** that our **repository** will be of **type** JpaRepository<Entity, Primary Key>:



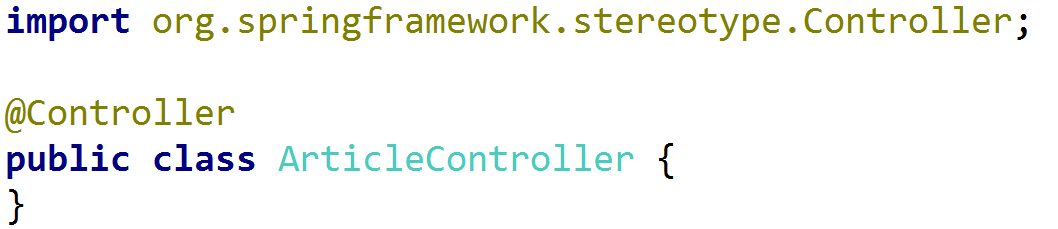
Here we've said that our entity is Article and its **primary key** in the database is of type Integer. **Spring** will do everything else for us.

## Create Article Controller

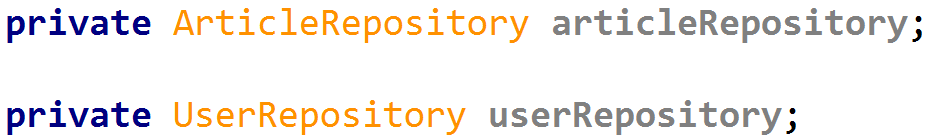
We have finally reached the point in which we can create our **controller**. In the "controller" package create a new class called "ArticleController":



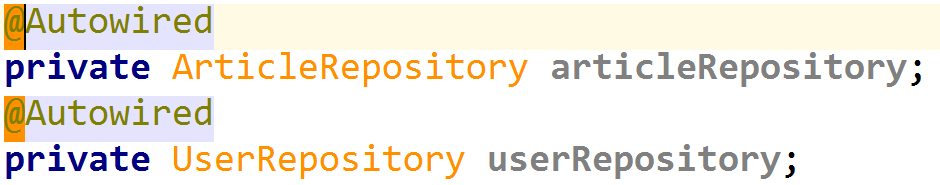
Add the following annotation:



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**:



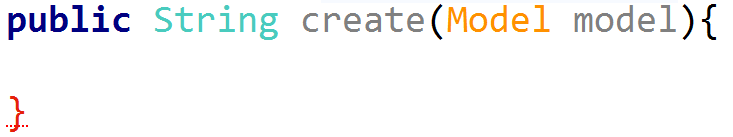
Add the following annotation to both of them:



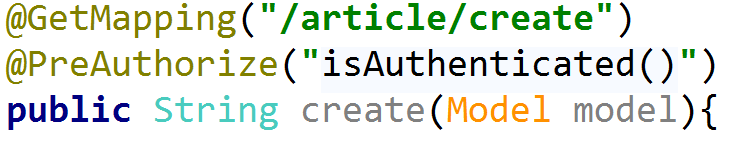
In short, **Spring** creates **object** of **each** **type** that we have in our application each time we **start** our **application**. It keeps them in something called [Spring IoC Container](https://www.tutorialspoint.com/spring/spring_ioc_containers.htm). Using the "@Autowired" annotation, we tell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

## 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:

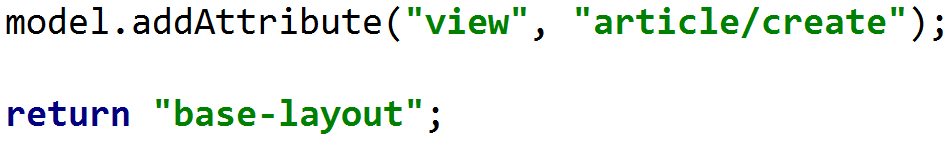


Our method will use "Model" that **Spring** will **send to the view automatically**. The "Model" is a **special dictionary** that we can use to send **any data** that we want to our **view**. The first thing we want to do is create the annotations:



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.

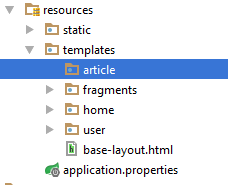
If you want to **know more about** how our **templating system** is working, you can find the information in [chapter V part 10](#_Creating_Base_Layout). In our method write the following code:



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**.

## Creating the View

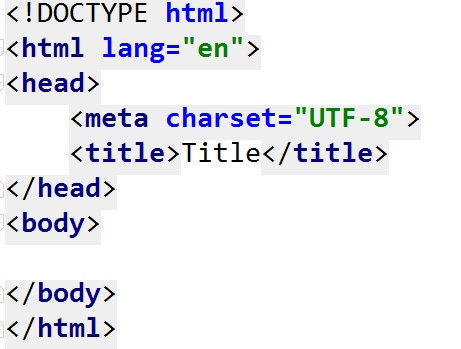
In order to use **loops** and **logical statements** in our **HTML** we will once again use **templating engine**. You should be **familiar** with **Twig** and **Handlebars** by now. Today you are going to use **Thymeleaf**. [Thymeleaf](http://www.thymeleaf.org/) is a really **easy to use** once you 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":



Inside that folder create a **new HTML file** called "create":



By default, our file will look like this:



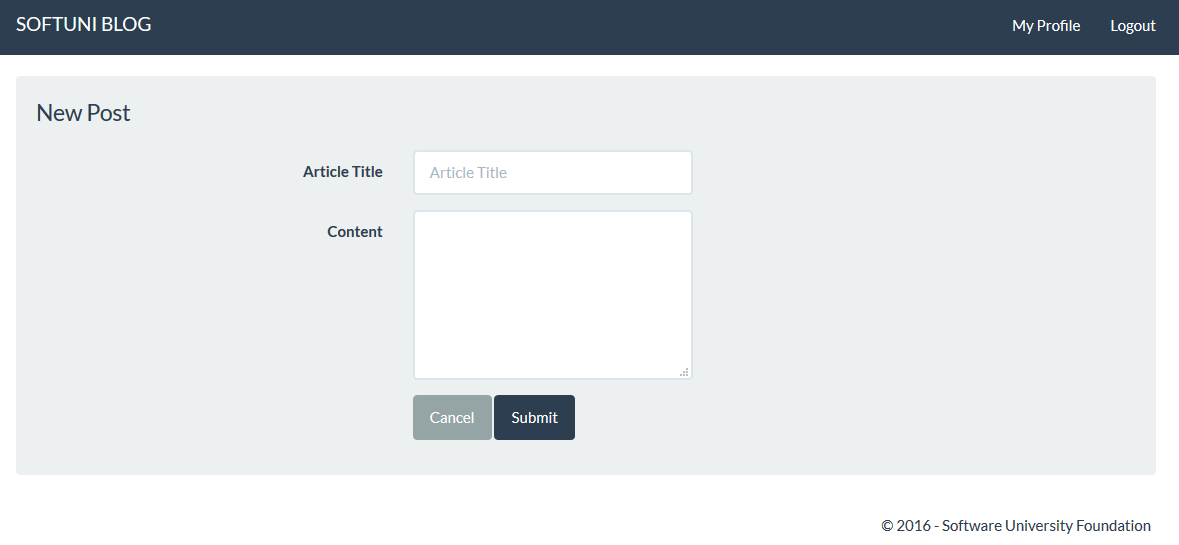
We **don’t need that**, so **delete everything**. Use the following code:

|  |
| --- |
| <**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**.

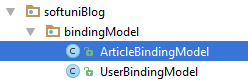
The next and last Thymeleaf attribute is "th:href". It will also **redirect** us **to** a **given** **route**. We can now **test the form** and see how it looks. **Start the application** and go to <http://localhost:8080/article/create>:



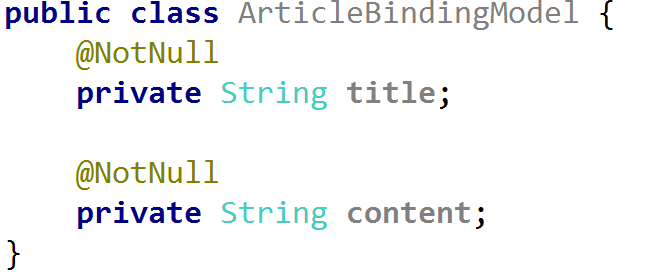
Looks good, but it **doesn't work**. We need to fix that.

## 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":



In that class create the following **private fields**:



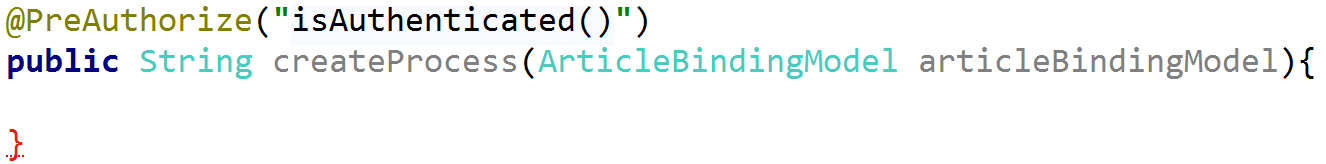
The "@NotNull" annotation is the only **validation** we are going to use. If the user tries to **submit** our **form** **without data**, it **will not validate**. If you check, you will see that these fields have **exactly the same name**, as the **input fields names** in our create form. This is **really 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:



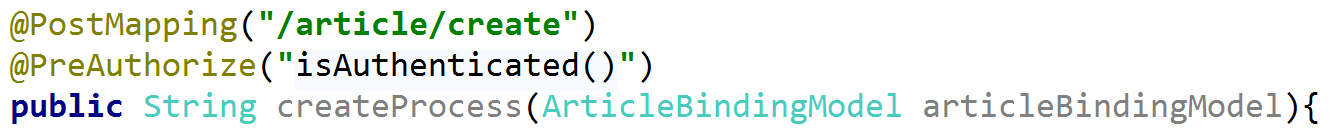
Our **form validation is ready**. Let's create the articles in the database.

## Creating Articles Part II

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



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**.

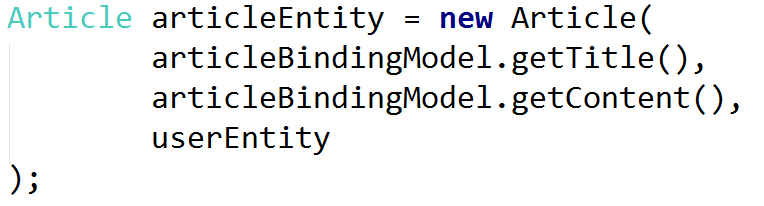
So, what do we need? First, we need to get the **currently** **logged in user**:



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:



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**:



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



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



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:



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

## Add Button to the Base Layout

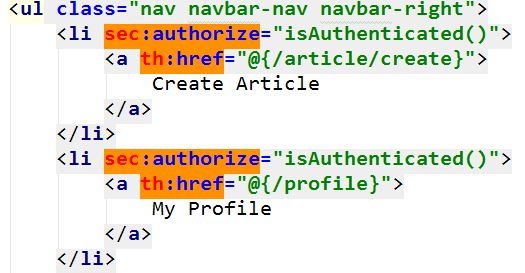
Open the "templates/fragments/header" file. Find the following section:



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**:

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

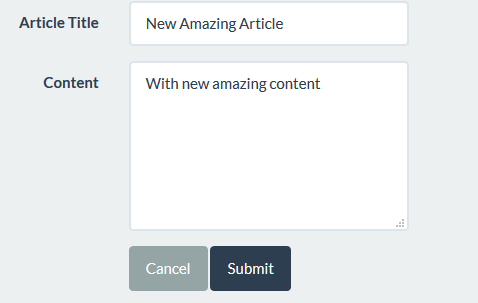
It should look like this now:



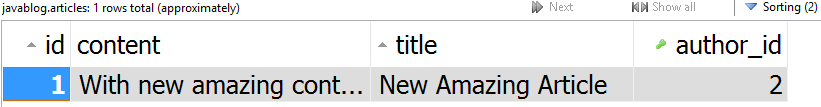
Let’s test our application:



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



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:

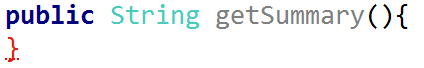


**It is working**! In the next chapter, we will **display our articles** on the **home page**.

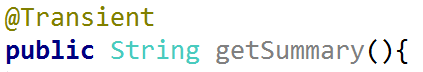
# List Articles

## 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**:



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

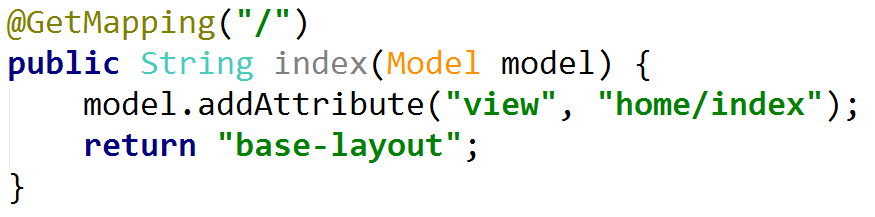


There are [other annotations](http://www.objectdb.com/java/jpa/entity/fields) that can manipulate what goes into the database. Now we just need return half of our content:

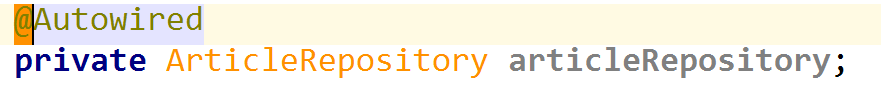


Now we have our **summary**. We can start **listing the posts**.

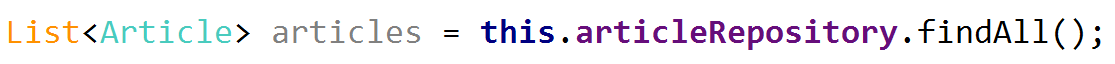
Open the HomeController and find the index() method:



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



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



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



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

|  |
| --- |
| <**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 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**.

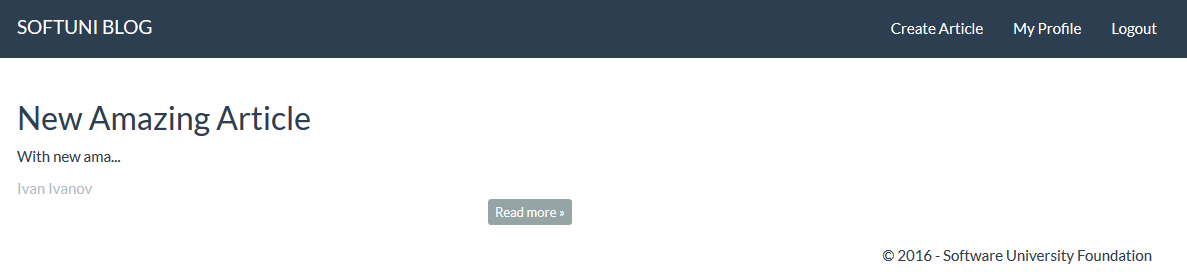
Inside, there is other attribute - "th:each="article : ${articles}"". This is a foreach loop. Thy syntax is really similar to the foreach loop in Java. We are getting each article from the articles collection that we've sent to our view. One thing that has to be mentioned is that the "${}" syntax tells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



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:

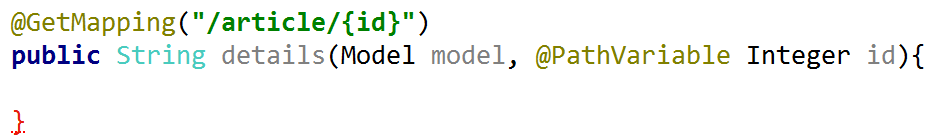


It works like a charm. Now we need to create **article details** page.

## 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":



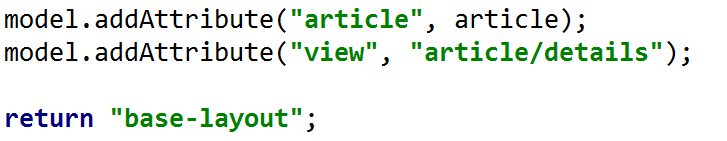
Something new! In our route, we **declare parameter** using **curly brackets**. Then in our method we use the "@PathVariable" annotation to tell **Spring** that **this parameter** should be **taken from the URL**. We are now free to use it in our method. The first thing we want to do is **check** **if** 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**:



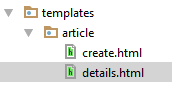
The next thing is to **get the article** from the database using our repository:



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:



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



**Delete the existing code** and use the following:

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

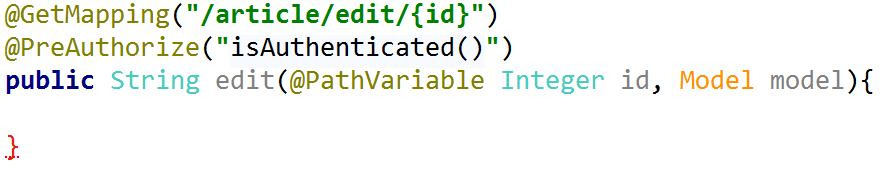


It is working. We need to **hide** the Edit and Delete **buttons** for everyone but **admins** and the **author**. We will do it in the following chapters.

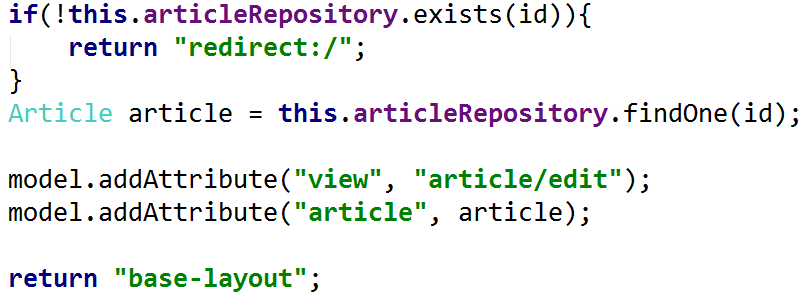
# Editing Articles

## Creating the Get Method

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



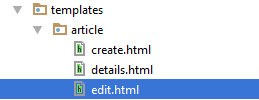
This method will be **similar** to our **details** method:



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

## Creating the View

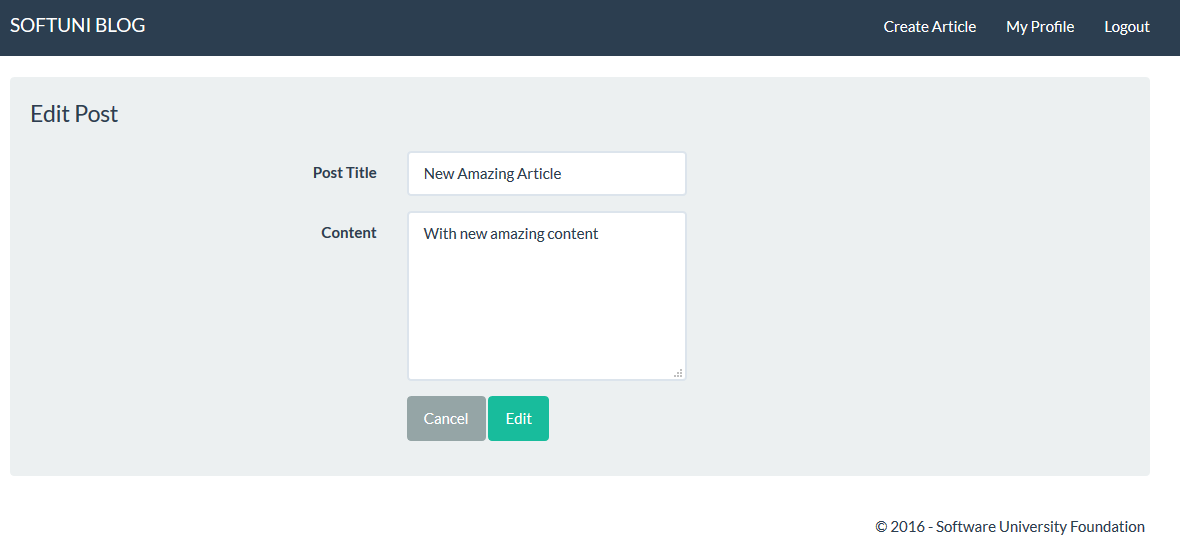
In the article folder create a new view called "edit":



Use the following code:

|  |
| --- |
| <**main**>  <**div class="container body-content span=8 offset=2"**>  <**div class="well"**>  <**form class="form-horizontal" th:action="@{/article/edit/{id}(id=${article.id})}" method="POST"**>  <**fieldset**>  <**legend**>Edit Post</**legend**>   <**div class="form-group"**>  <**label class="col-sm-4 control-label" for="article\_title"**>Post Title</**label**>  <**div class="col-sm-4 "**>  <**input type="text" class="form-control" id="article\_title" placeholder="Post Title" name="title" th:value="${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**> |

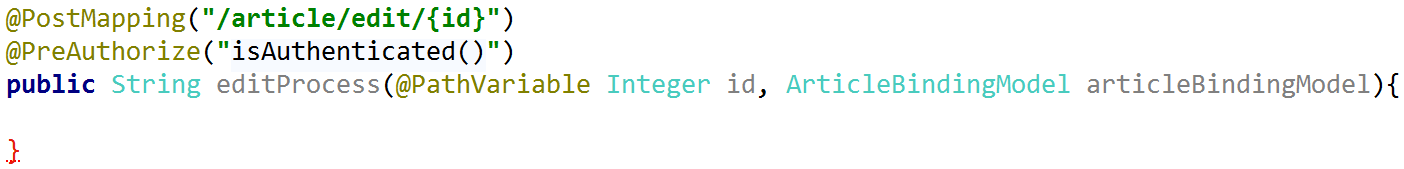
Here, you can see we are using "th:value" and "th:field". The value attribute is used to give **input fields** value. The <textarea> is a **special input field** that **doesn’t have value attribute**. Because of that, we are using "th:field", which will **replace** some of our **original attributes** 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:



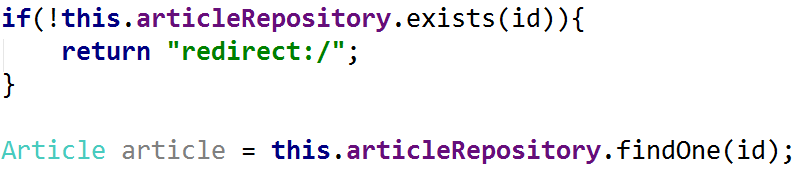
Not that bad. Let's make it work.

## Creating the Post Method

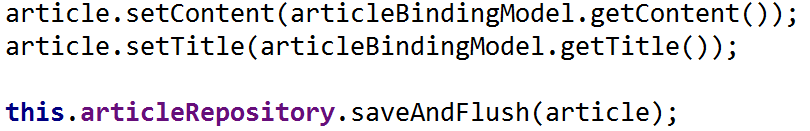
New task – create the post method. In our ArticleController create new method with the following annotations:



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:



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



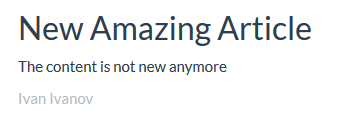
Finally, **redirect** the user **to the article details**:



Test it and see if it works:



It should be working just fine:



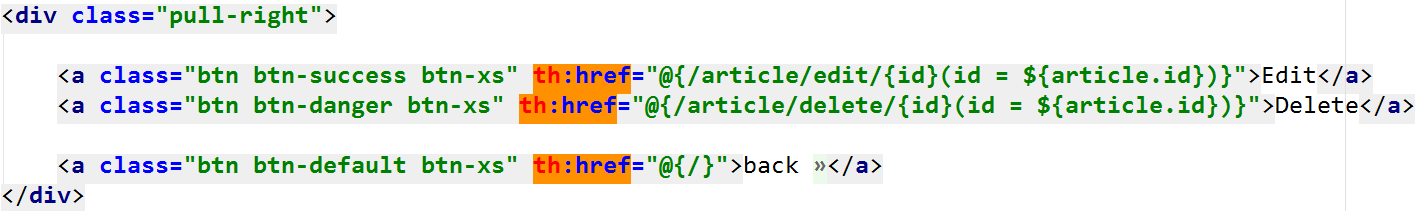
However, there is a **slight problem**…

## Hiding Buttons

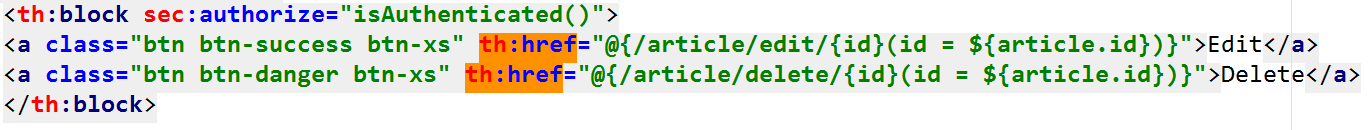
Don't know if you've noticed, but when you are **not logged in**, you still see the edit and delete buttons:



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



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

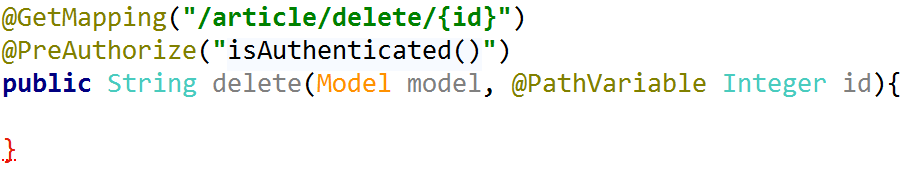


Using **Thymeleaf Security** we are making sure that **only logged in users** **will** **see the buttons**.

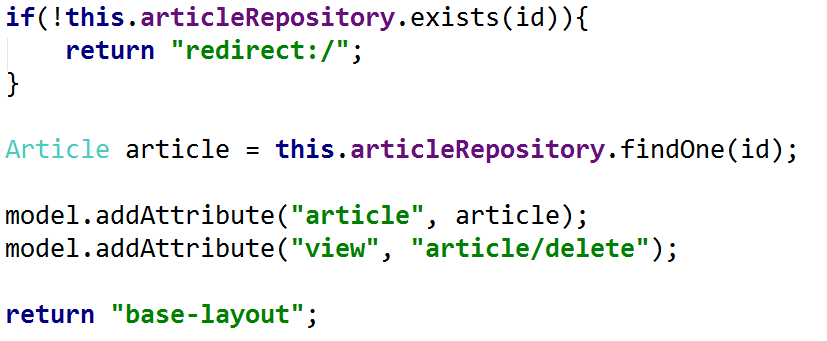
# Deleting Articles

## Creating the Get Method

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



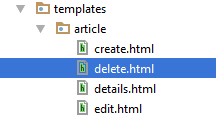
You should write the following code, that may look familiar:



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

## Creating the View

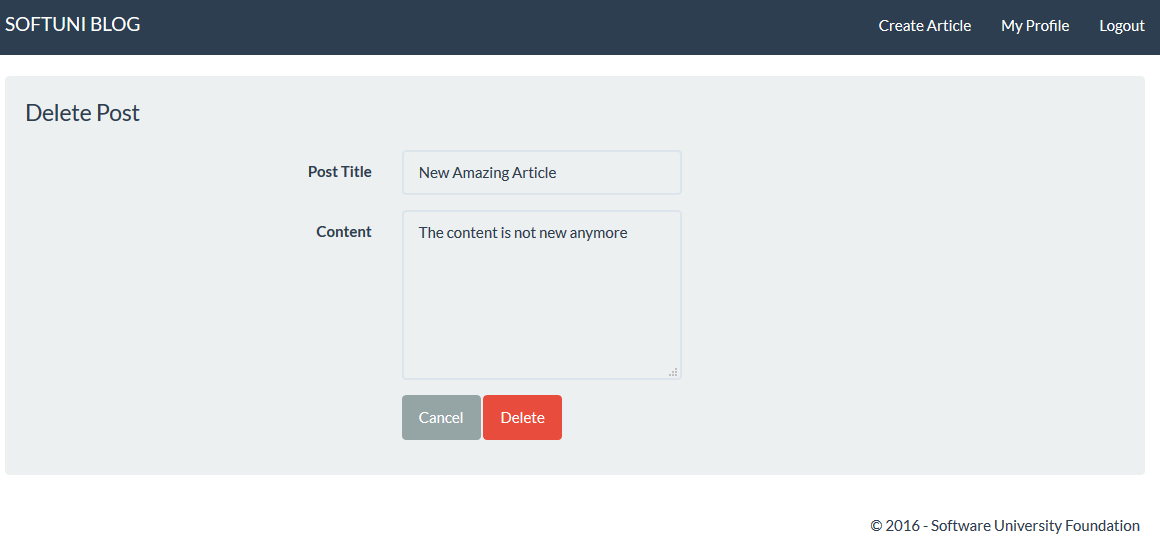
You need to **create a new view** called "delete" in the "article" folder:



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:

|  |
| --- |
| <**main**>  <**div class="container body-content span=8 offset=2"**>  <**div class="well"**>  <**form class="form-horizontal" th:action="@{/article/delete/{id}(id=${article.id})}" method="POST"**>  <**fieldset**>  <**legend**>Delete Post</**legend**>   <**div class="form-group"**>  <**label class="col-sm-4 control-label" for="article\_title"**>Post Title</**label**>  <**div class="col-sm-4 "**>  <**input type="text" class="form-control" id="article\_title" placeholder="Post Title" name="title" th:value="${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**> |

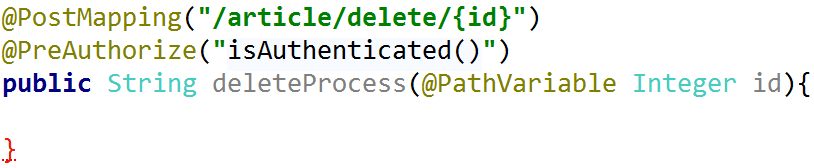
That is how it should look:



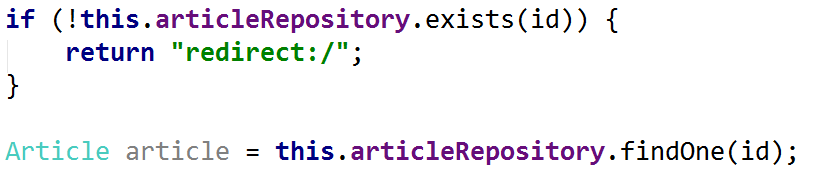
Nothing new here, let's move on.

## Creating the Post Method

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



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:



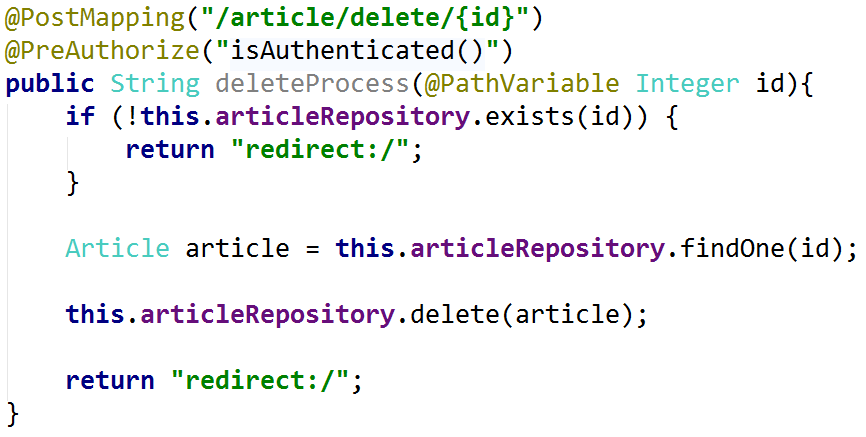
We are checking if such **article exists** and then **taking it from the database**. Now we simply need to tell our **repository** to **remove it** from the database:



Finally, we need to **redirect the user** to the home page:



Overall, the code should look like this:



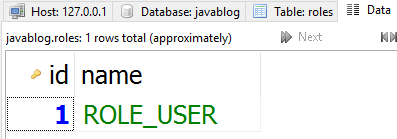
We are **verifying that the article exists**, then **deleting it from the database**. It should work now, so you can test it.

In the next chapters, we are going to add the admin functionality. ☺

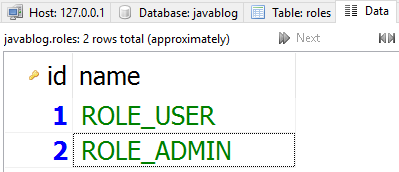
# Roles

## Creating Role

If you open **HeidiSQL**, you will see that we have **1 row** in our roles table:



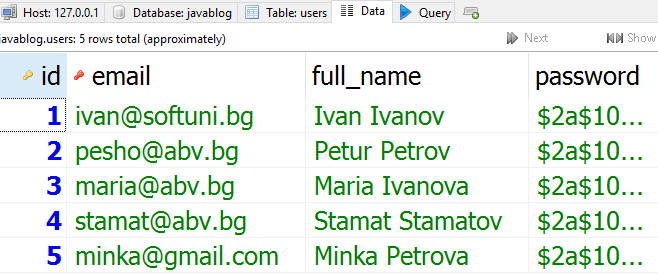
Using the method from [chapter V part 14](#_Finishing_the_User) you must create another role called "**ROLE\_ADMIN**":



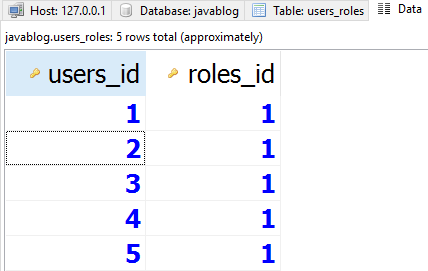
Now we can add admin to our blog.

## Adding Admin

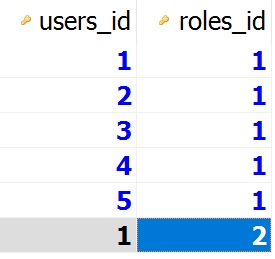
Register few users:



Open the **data** **view** of our "users\_roles" table:



At the moment, all of our users **don't have admin privileges**. Create a **new row** in the **database** and **manually assign** one of your **users** with the **second** **role**:

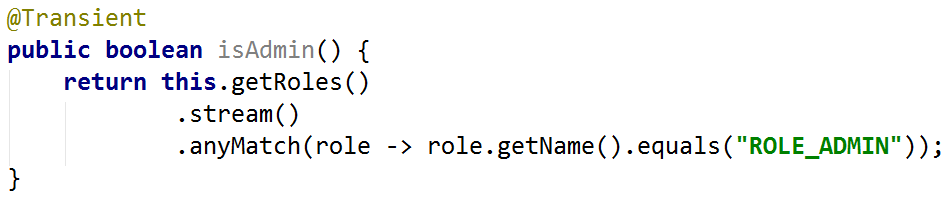


Now that the **user** with **id 1** has **role** with **id 2**, we can create our functionality.

# Validations

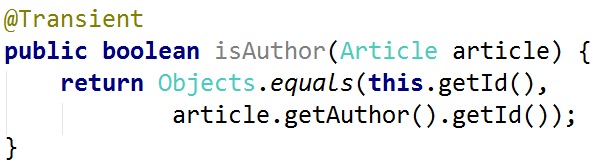
## Creating User Helper Methods

Before we are trying to **verify** that our **user** is either **admin** **or** **author** of the **article**, we should make it easier to create the checks. In our User entity create a new method:



This method uses [Stream API](https://www.tutorialspoint.com/java8/java8_streams.htm) in java (similar to **LINQ** in **C#**) to find if the **admin role exists** in the **current user roles**. It returns true or false based on the result.

The second method will check if the **current user is author** of a given **article**:

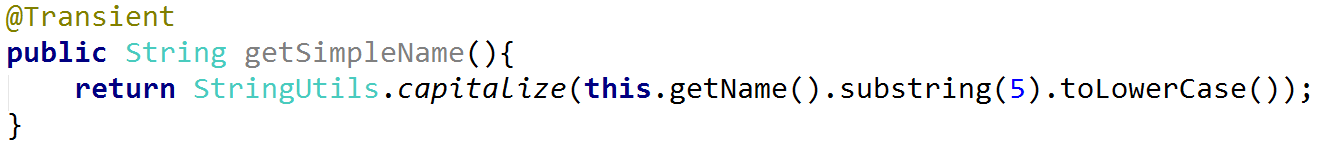


We are comparing the author id with our user id and again return true or false based on the result.

We are using the "@Transient" annotation on both methods to tell **Hibernate** to **not save them** in our database.

## Creating Role Helper Method

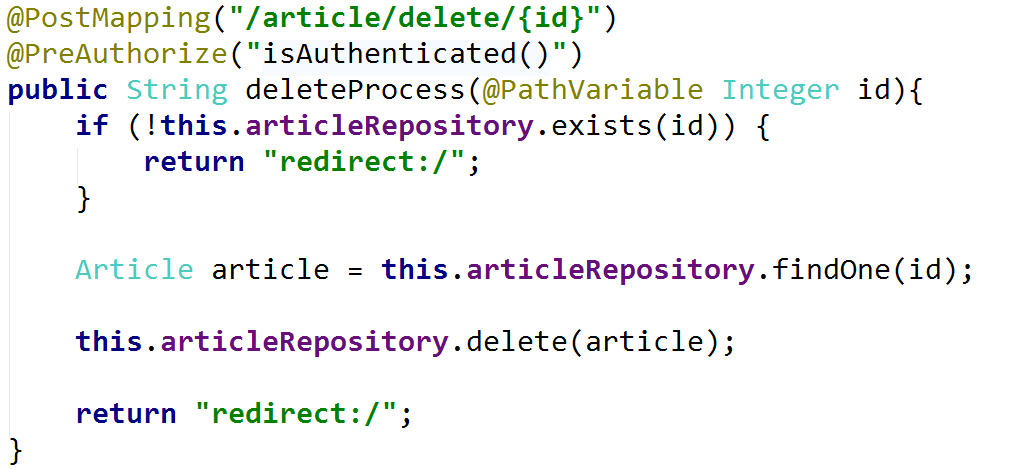
We will create another helper method, this time in the Role entity that will give us **better looking name** for any given role:



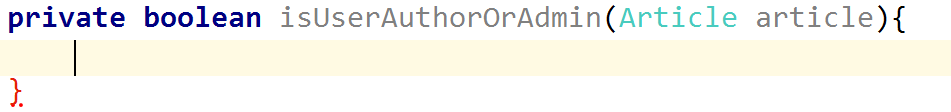
Here, we are taking everything **after** the **underscore** ('**\_**') and making it **lower case**. Then, we are making the **first letter capital**, to make the role name **aesthetically pleasing**. Again, this method should not be saved in our database.

## Validating the Article Requests

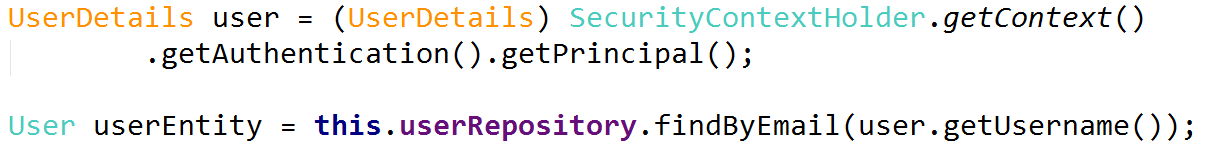
In our ArticleController find the deleteProcess() method:



The only thing that we are validating here is that the **article exists**. We can create **helper** **method** that will check if the **user** is **admin** or **author**:



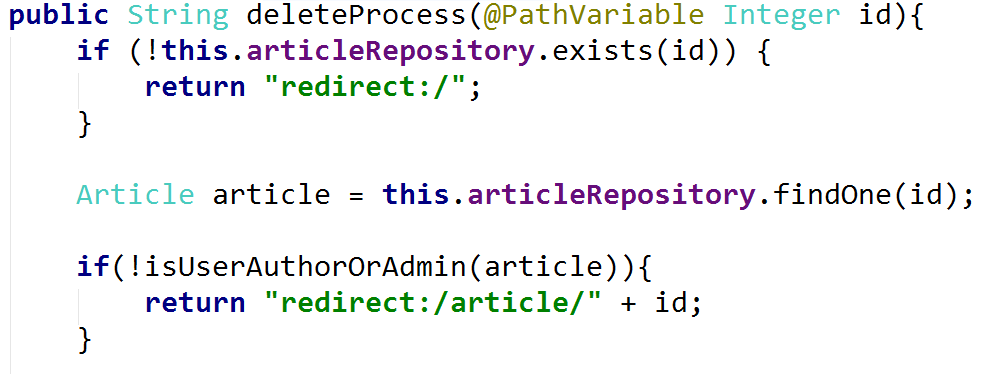
In this method, we need to find our **current user** in order to make our check:



Then, we can **return the result** of the helper methods we've created earlier:

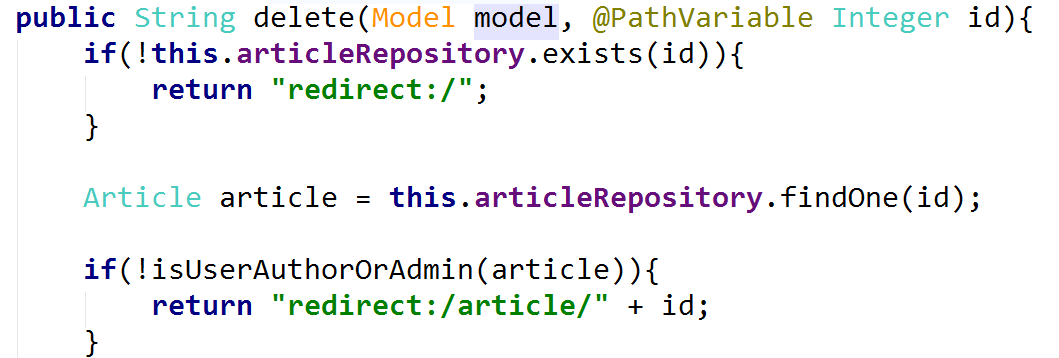


The last thing to do here is use that method in our deleteProcess() method:

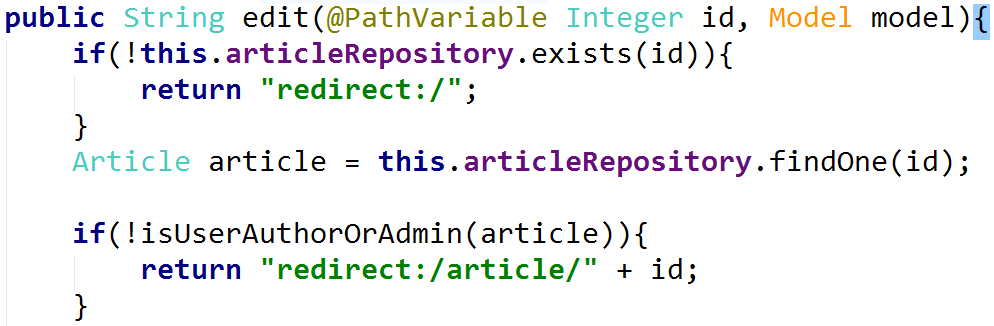


We need to perform that check for every other of the following methods:

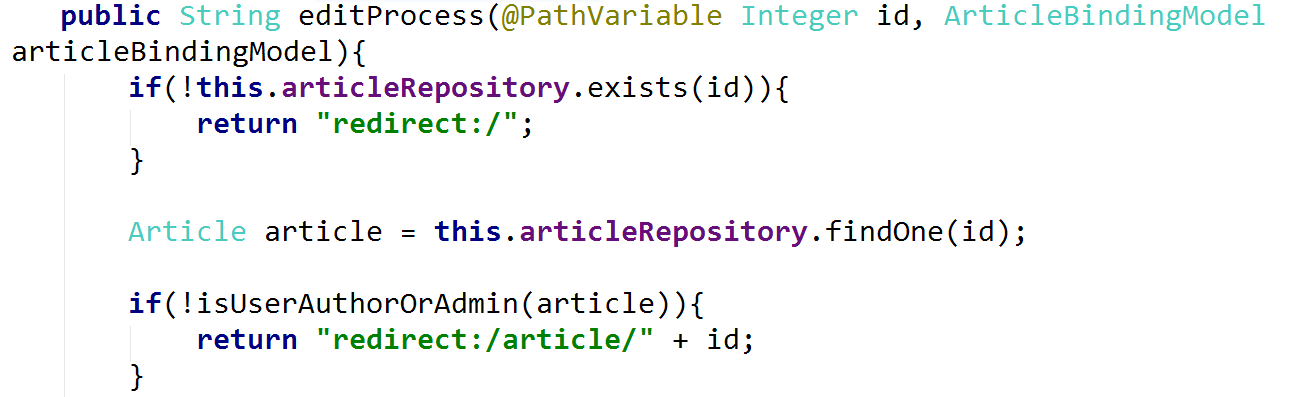
delete():



edit():



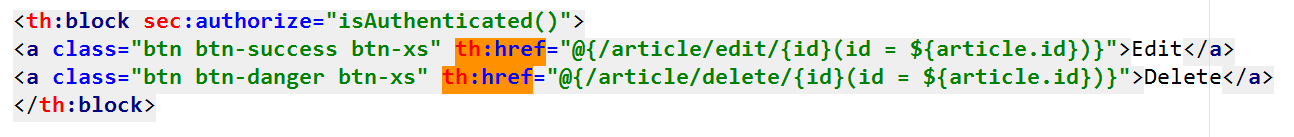
editProcess():



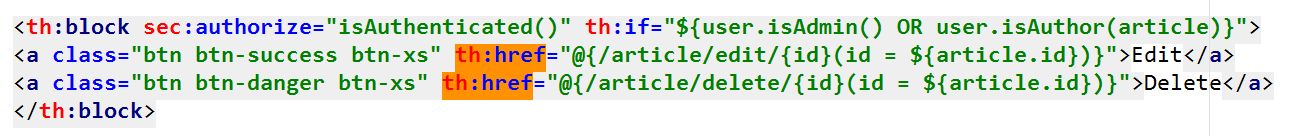
This will **stop unauthorized access** to our request methods, but we need to **show the edit and delete buttons** **only** **to** the **users** that have the **rights to edit or delete an article**.

## Creating View Helper for Article Details View

We've **validated our back-end**, but we need to do the same for the **front-end** part. Find our **edit and delete buttons** in the details view:

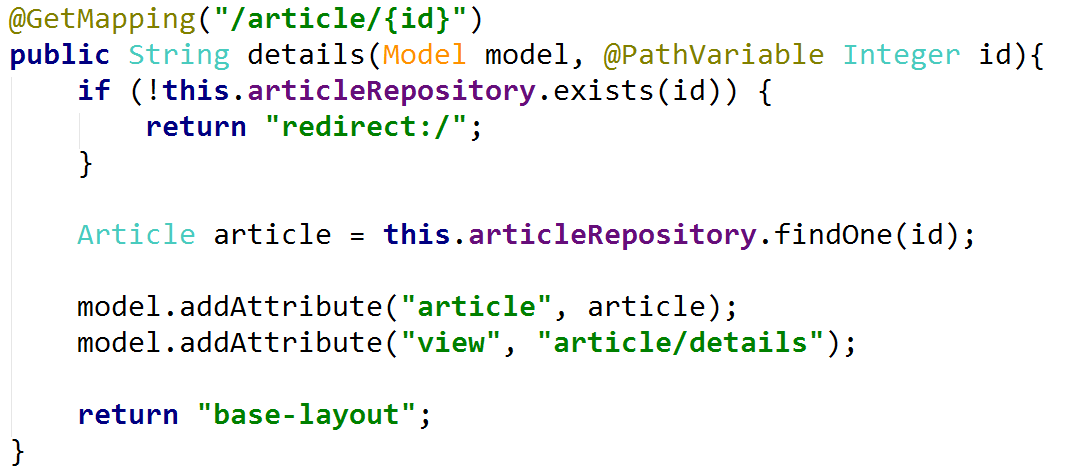


Currently, we are only checking if the user **is logged in**. We need to create **conditional statement** to verify that the **user** **is** **author** **or** **admin**:

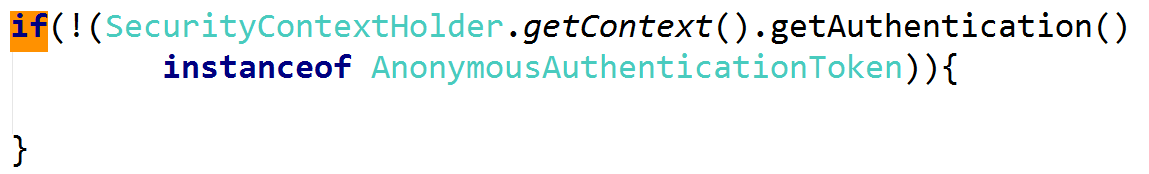


Here we are making use of something called [short-circuit evaluation](https://en.wikipedia.org/wiki/Short-circuit_evaluation). To explain it in our case let's take the following example – "user is reviewing our article". In that case **Thymeleaf Security** will check if there is **authorized user**. Then it will proceed to check if the **user is admin**. If that method returns true, the buttons **will be displayed** and **we won't make the last check**. If the **user isn’t admin**, we will do the **final check** if he is **author** and based on the result, we will display the buttons.

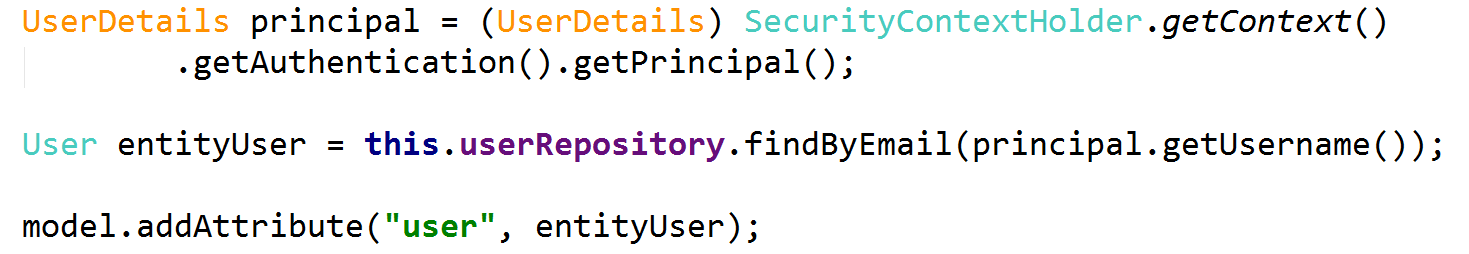
In order to use the "user" variable in our view, we need to send it, using our ArticleController. Find the details() method:



Here we have a **special case**. This method can be accessed **both by users** of our blog **and** **guests**. That's why we need to verify that we are only **trying to get the user** if the page is **accessed by authenticated user**. We will do that using the following if-statement:



This will get our **authentication token** and check if the type of **user is anonymous**. If the **authentication is anonymous** it means that **we don't have logged in user**. In the **body** of our if, we need to **get our current user**:



Overall, the method should look like this:



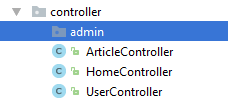
Now we can **start the application** and see if **everything works**.

# Admin Panel

Now, that everything else is done, we can focus on the **admin functionality**. We will give the admins the ability to **edit** and **delete users**.

## Listing Users

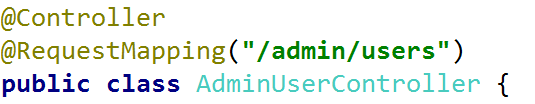
In our "controller" package, create a new package called "admin":



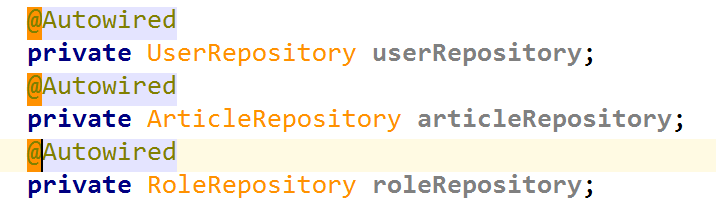
Now, create a new class called "AdminUserController":



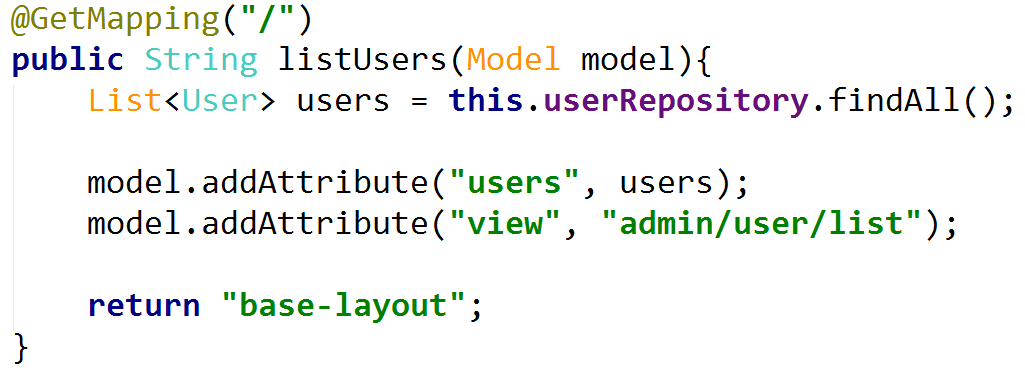
We want to do something additional to our controller – **create a default route** **for all methods**. Why? Because all of our methods will **start with** "/admin/users" and we **don't want to write this every time**. Use the following annotation:



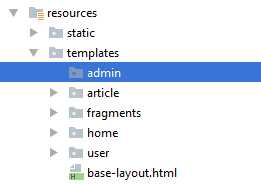
We will need all of our repositories, so we can create them now:



Now, we can create our first method that will **send all users to a view**:



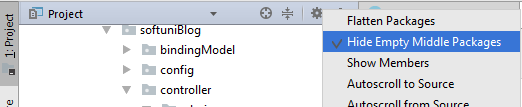
Let's create the view now. Create a new "admin" directory:



Inside, create a new folder called "user". You may get the following result:



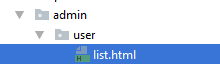
If you do, use the **gear icon** on top of the **Project** tab and **remove the checkmark** from "**Hide Empty Middle Packages**":



Now, your new folders should look like that:



Create "list" view:

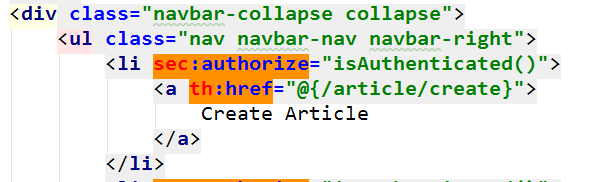


And use the following html:

|  |
| --- |
| <**main**>  <**div class="container body-content"**>  <**div class="well"**>  <**h2**>All Users</**h2**>  <**div class="row"**>  <**table class="table table-striped table-hover "**>  <**thead**>  <**tr**>  <**th**>#</**th**>  <**th**>Full Name</**th**>  <**th**>Email</**th**>  <**th**>Actions</**th**>  </**tr**>  </**thead**>  <**tbody**>   <**th:block th:each="user : ${users}"**>  <**tr th:class="${user.isAdmin() ? 'info' : null}"**>  <**td th:text="${user.id}"**></**td**>  <**td th:text="${user.fullName}"**></**td**>  <**td th:text="${user.email}"**></**td**>  <**td**>  <**a th:href="@{/admin/users/edit/{id}(id=${user.id})}"**>Edit</**a**>  <**a th:href="@{/admin/users/delete/{id}(id=${user.id})}"**>Delete</**a**>  </**td**>  </**tr**>  </**th:block**>   </**tbody**>  </**table**>  </**div**>  </**div**>  </**div**> </**main**> |

The new thing in this code is the "th:class" attribute. This attribute will **add any given class** based on **condition**. In our case, we are **checking if the user is admin** and if he is, we are using the "**info**" **class** from bootstrap. Otherwise, we don't use any class.

Because we are lazy and **don't want to write the URL manually**, we will add **drop-down menu** that will **contain our admin functionality**. Open our "header" fragment and find the following piece of code:



At the beginning of the list, add the following code:

|  |
| --- |
| <**li class="dropdown" sec:authorize="hasRole('ADMIN')"**>  <**a href="#" class="dropdown-toggle" data-toggle="dropdown" role="button" aria-expanded="false"**>Admin<**span class="caret"**></**span**></**a**>  <**ul class="dropdown-menu" role="menu"**>  <**li**><**a href="#"**>Categories</**a**></**li**>  <**li**><**a th:href="@{/admin/users/}"**>Users</**a**></**li**>  </**ul**> </**li**> |

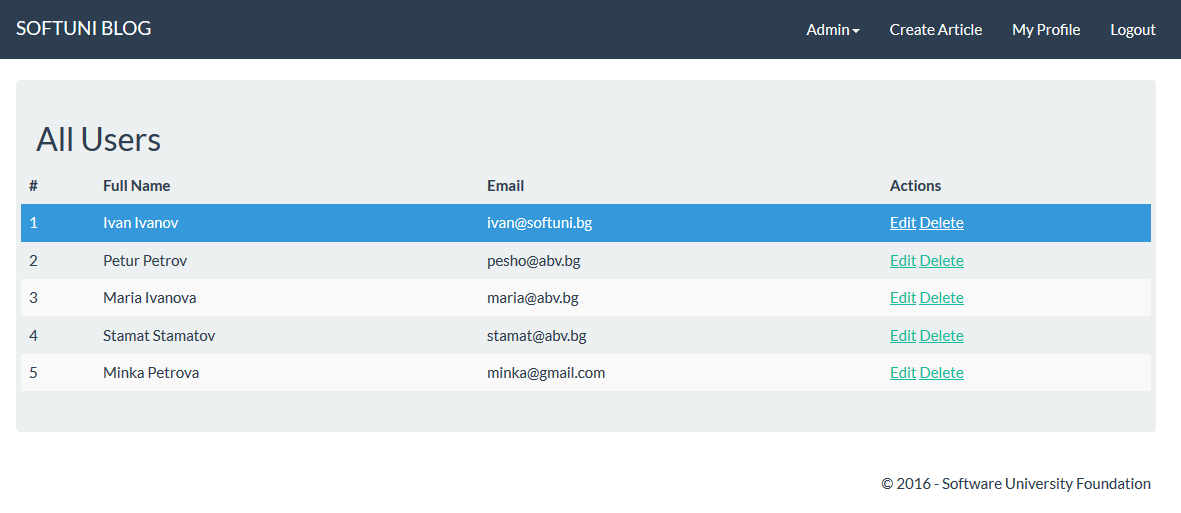
What are we doing here? Using the "sec:authorize" attribute, we verify that only users with the "ADMIN" role will see the menu. The code should look like this:



Run the blog and see if it works. Our **navigation bar**:



And our **list with users**:



Our listing is done, it's time to **edit the users**.

## Editing Users

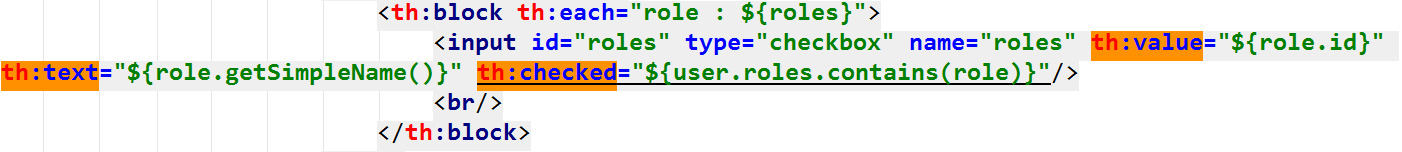
This will probably the hardest part in the entire blog. Well maybe **not the hardest**, but the "ugliest" one. We need to create a form that **looks like our register form**, but from it we can also **change the roles** each user has. We will start by creating the following method:



We are getting **all of the roles** from the **database** and sending them to the view. Then we are sending the **user we are going to edit** to the view as well. This method should be easy to understand. The next part is our view that we need to create in "admin/user" and call it "edit":

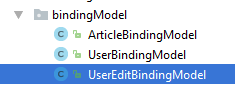
|  |
| --- |
| <**main**>  <**div class="container body-content span=8 offset=2"**>  <**div class="well"**>  <**form class="form-horizontal" th:action="@{/admin/users/edit/{id}(id=${user.id})}" method="post"**>  <**fieldset**>  <**legend**>Edit User</**legend**>  <**div class="form-group"**>  <**label class="col-sm-4 control-label" for="user\_email"**>Email</**label**>  <**div class="col-sm-4 "**>  <**input class="form-control" type="email" id="user\_email" placeholder="Email" name="email"  required="required" th:value="${user.email}"**/>  </**div**>  </**div**>  <**div class="form-group"**>  <**label class="col-sm-4 control-label" for="user\_fullname"**>Full Name</**label**>  <**div class="col-sm-4 "**>  <**input class="form-control" type="text" id="user\_fullname" placeholder="Full Name"  name="fullName" required="required" th:value="${user.fullName}"**/>  </**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"**/>  </**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"**/>  </**div**>  </**div**>   <**div class="form-group"**>  <**label class="col-sm-4 control-label" for="roles"**>Roles</**label**>  <**div class="col-sm-6"**>  <**th:block th:each="role : ${roles}"**>  <**input id="roles" type="checkbox" name="roles" th:value="${role.id}" th:text="${role.getSimpleName()}" th:checked="${user.roles.contains(role)}"**/>  <**br**/>  </**th:block**>  </**div**>  </**div**>   <**div class="form-group"**>  <**div class="col-sm-4 col-sm-offset-4"**>  <**a class="btn btn-default" th:href="@{/admin/users/}"**>Cancel</**a**>  <**input value="Edit" type="submit" class="btn btn-success"**/>  </**div**>  </**div**>  </**fieldset**>  </**form**>  </**div**>  </**div**> </**main**> |

The interesting part here is that we are creating our roles in a foreach loop:

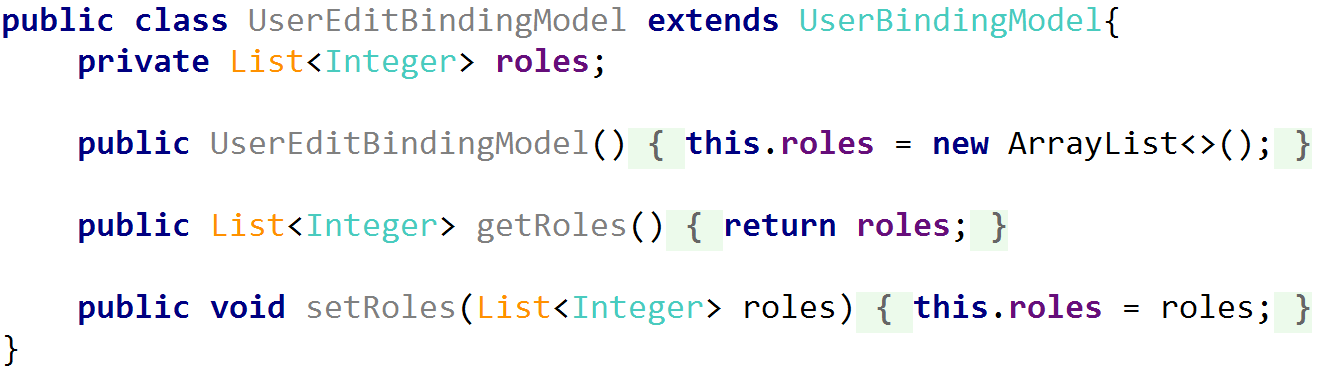


Here is the most complicated part in our view. The **checkboxes have several attributes** that we are using. First, let's say that the **checkbox** is something like a **key-value pair**. We are **displaying the keys to our admin** and **sending the values to our controller**. Using the "th:value" attribute we defining that **each textbox** will keep **the role id**. Later we will **use the values** as List<Integer>. The "th:text" is what the **user** is **going to see as a content** of the checkbox. We will use the helper method we've created earlier to display the name of the role. Finally, the "th:checked" attribute will create the **default state** of our **checkbox**. We are saying that the **checkbox should be checked** **if the user roles** collection **contains** the **current** **role**.

We've said that our checkboxes will give us **List<Integer>** when the **form is submitted**. That means that we have to create a **new binding model**:

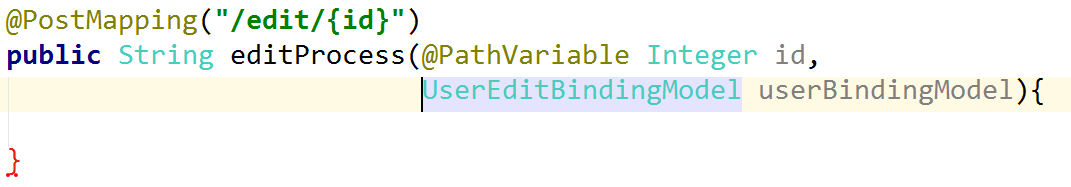


Write the following code:

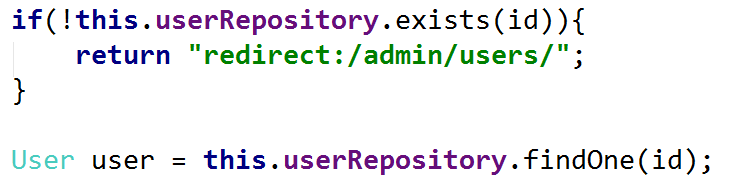


What do we have here? We have private field, constructor, getter and setter. Notice that our **private field** has the **same name**, **as the name of the input field** in the form. The interesting part here is "extends UserBindingModel". That means that **we will have access** to the email, fullName, password and confirmPassword from the other binding model. That way we can use them, without duplicating our code. We have to create our second method in the controller now.

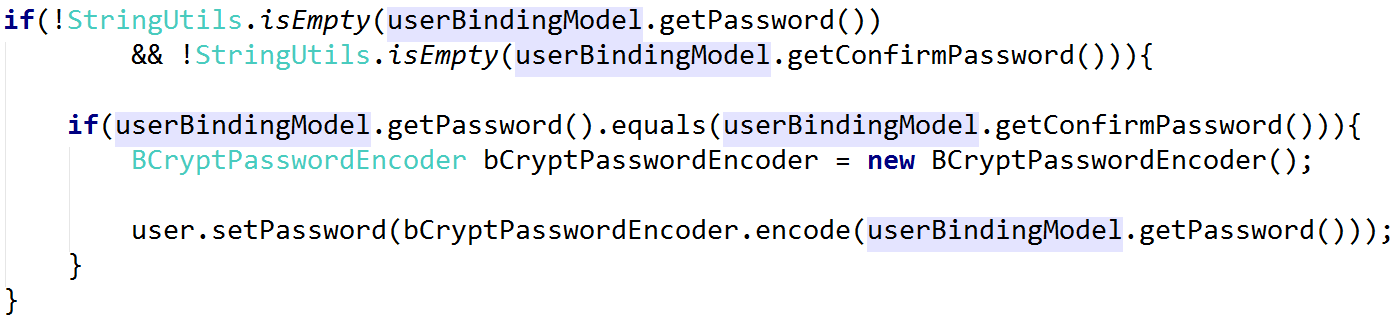
In the AdminUserController create a new method:



Here we are telling the method that it should receive **our newly created binding model** and the **id** of the **user** that **we are going to** **edit**. The first part of our code will be **getting the user from the database**:

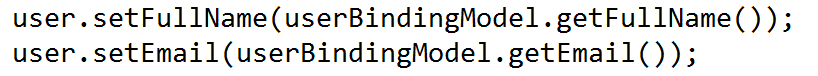


Here comes the tricky part. **We don't want** to **change the user password** if the admin left **the password fields empty**. Because of that we are going to write the following code:

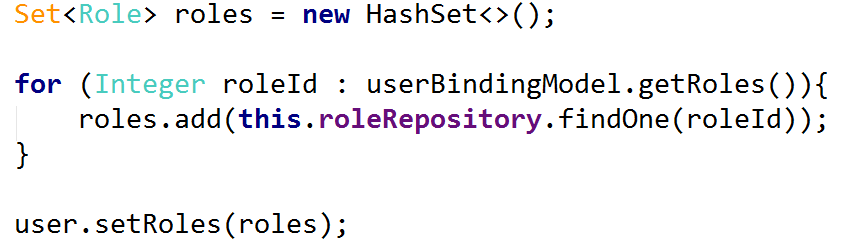


In the first if-statement we check **if the password fields are not empty**. Then, in the second if-statement we check **if the two passwords match**. If the match, we **encrypt the password** and **set it as a new password** for our user.

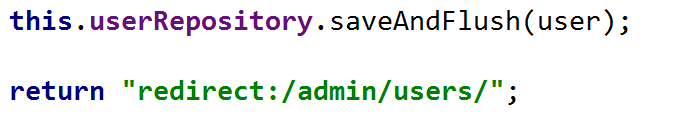
The third part of our code is **changing** the user **name** and **email**:



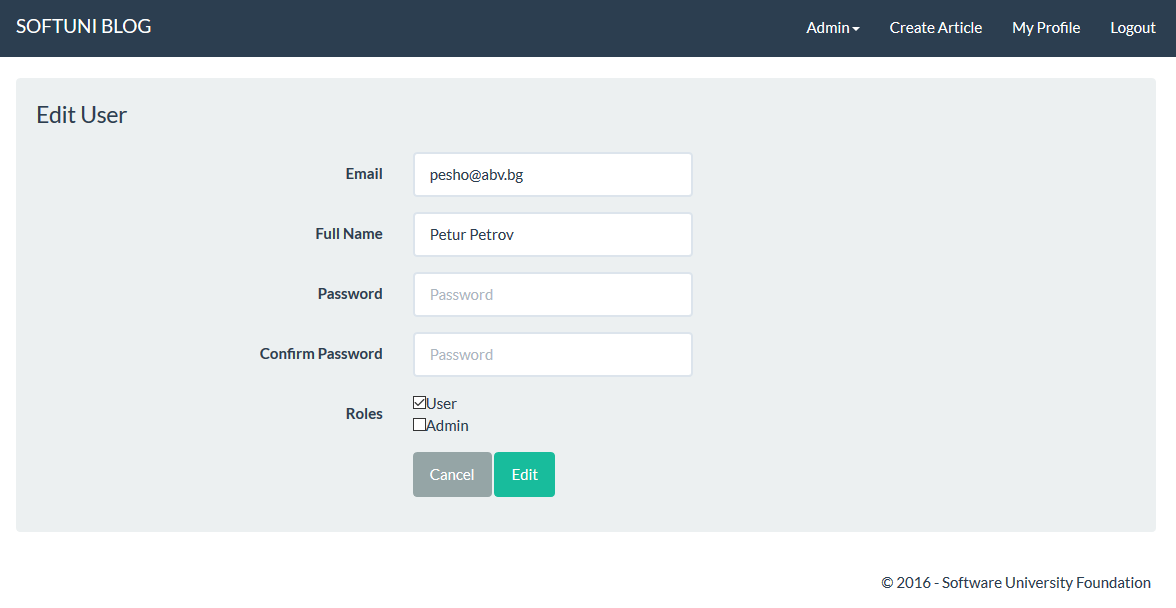
Then, we **iterate** over our **list of integers** that keeps the id for **each role** that the **user will have**. We **get the roles from the database** and when we're done, we **set the new roles** to our user:



Finally, we **save the user** and **redirect the admin** to the users listing:



The **edit should work now**, go ahead and test it. The view will look like this:



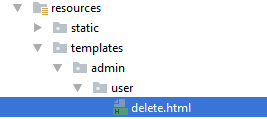
**Deleting the users** is our next target.

## Deleting Users

The final part of our user administration is the **delete**. In our controller create new "delete" method that will **return the view with the user** or **redirect to the listing** page:



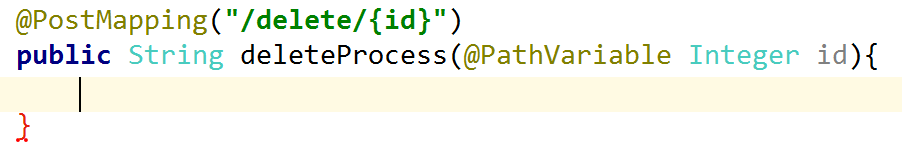
The view must be located in "admin/user" and must be called "delete":



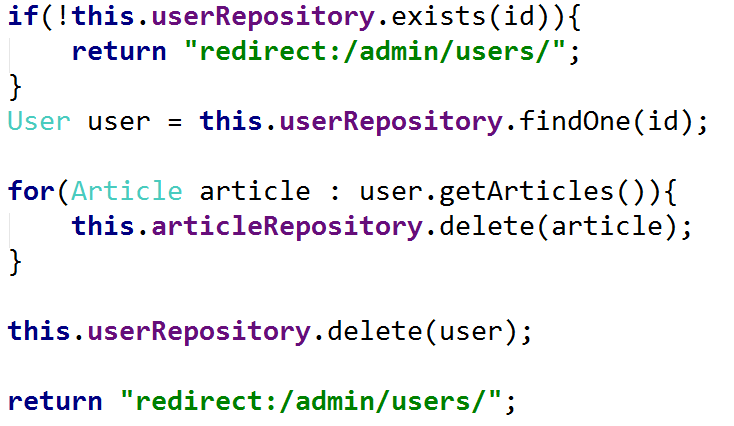
The form will be really simple with only 2 fields:

|  |
| --- |
| <**main**>  <**div class="container body-content span=8 offset=2"**>  <**div class="well"**>  <**form class="form-horizontal" th:action="@{/admin/users/delete/{id}(id=${user.id})}" method="post"**>  <**fieldset**>  <**legend**>Delete User</**legend**>  <**div class="form-group"**>  <**label class="col-sm-4 control-label" for="user\_email"**>Email</**label**>  <**div class="col-sm-4 "**>  <**input class="form-control" type="email" id="user\_email" placeholder="Email" name="email" th:value="${user.email}" disabled="disabled"**/>  </**div**>  </**div**>  <**div class="form-group"**>  <**label class="col-sm-4 control-label" for="user\_fullname"**>Full Name</**label**>  <**div class="col-sm-4 "**>  <**input class="form-control" type="text" id="user\_fullname" placeholder="Full Name" name="fullName" th:value="${user.fullName}" disabled="disabled"**/>  </**div**>  </**div**>   <**div class="form-group"**>  <**div class="col-sm-4 col-sm-offset-4"**>  <**a class="btn btn-default" th:href="@{/admin/users/}"**>Cancel</**a**>  <**input value="Delete" type="submit" class="btn btn-danger"**/>  </**div**>  </**div**>  </**fieldset**>  </**form**>  </**div**>  </**div**> </**main**> |

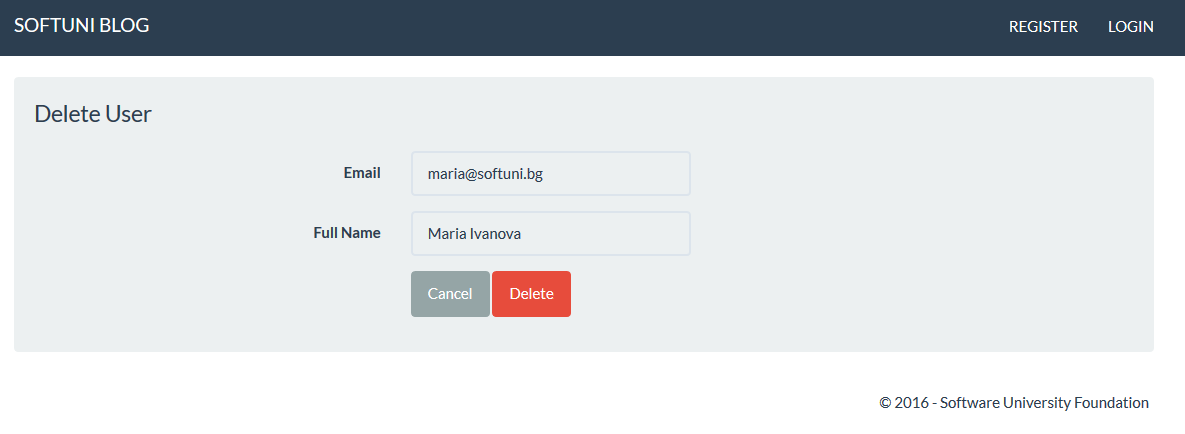
The unique thing about this form is that **the fields are disabled**. In the end, we need to implement the method that will work with the "POST" request:



This method will be **unique on its own**. We **can't delete the user** if he has created any posts. That's why we need to **delete all of his articles first**. **Then we can delete the user** as well:



The **delete is working** and it looks like this:



You have **spotted the problem**, haven't you? Take a look at the **navigation bar** in the last picture. We **can access** the **delete** and **edit** pages **even as guests**. We need to fix that.

## Adding Access Control

Making **routes** **limited** to a **given role** is easily done in Spring. Open WebSecurityConfig from the "config" package:



This method is what you should be looking for. Change it to this:



The row we added will **limit all routes** **starting** **with** "/admin" to the users, which have the "ADMIN" **role** (ignore the "…antPatterns" and "role:", those **hints** are coming from **the new IntelliJ Idea 2016.3**). Every other request will be allowed.

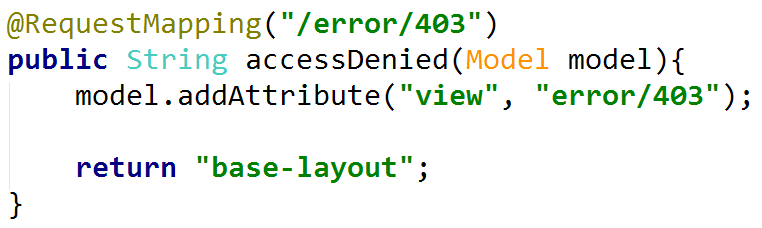
Now, if you try to **open the admins page** with **normal user**, you will receive this **strange page**:



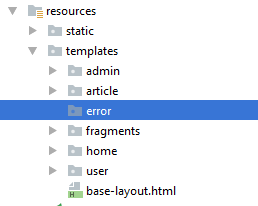
What makes it strange? We receive error "404 Not Found", instead of error "403 Forbidden". That is because in our **config file** we have the following line:



This means that **when we don't have access** to a given route**, Spring should redirect us to "**/error/403**"**. However, we haven't created that route anywhere in our blog. That's why we need to open our HomeController and create the following method:



We are just returning a view. Create a new folder called "error":



Now, create a file called "403". Use the following html:

|  |
| --- |
| <**main**>  <**div class="container body-content"**>  <**div class="row"**>  <**div class="col-md-12"**>  <**div class="alert alert-dismissible alert-danger"**>  <**h2**>Access Denied!</**h2**>  <**h4**>You need to <**a th:href="@{/logout}" class="alert-link"**>login</**a**> with different user.</**h4**>  </**div**>  </**div**>  </**div**>  </**div**> </**main**> |

If try it now, it should look like this:



This is **everything for this chapter**, in the next one we are going to implement **categories**.

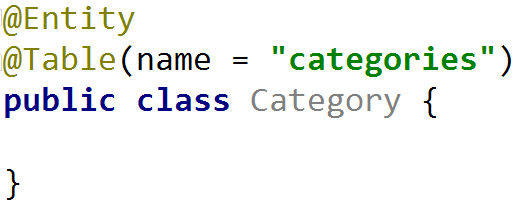
# Categories

## Delete Existing Articles

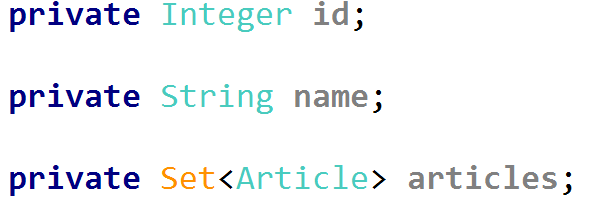
It's important to **delete all articles** from the database. You can use your blog functionality or use **HeidiSQL** to delete them.

## Creating Entity

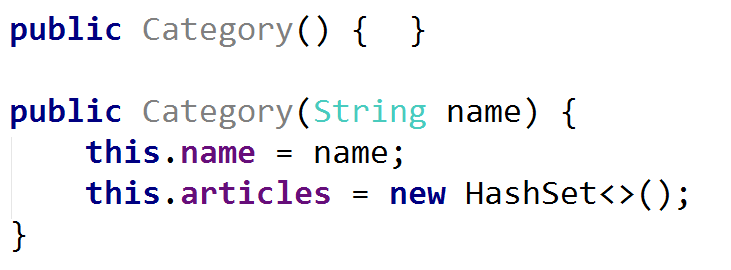
Our first step should be to create the **category** entity:



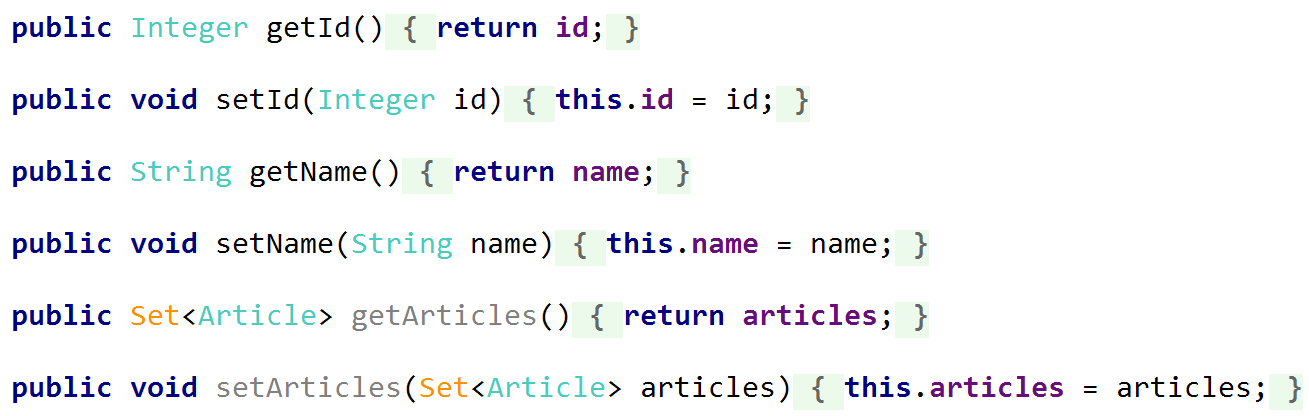
Now, we can create the private fields:



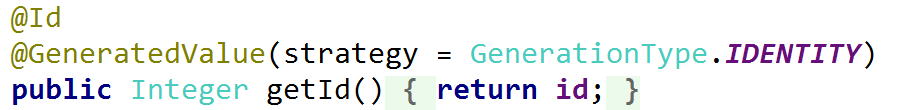
Each category will only have **name** and **articles** related to it. Next thing – **constructors**:

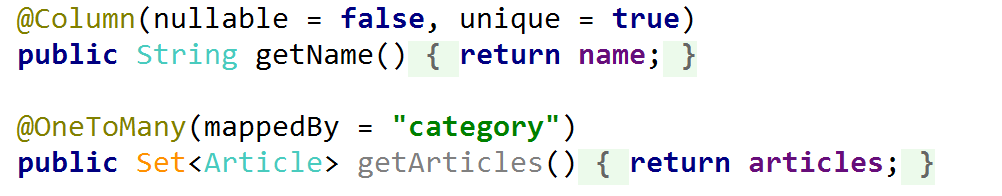


Again, we're creating the **second constructor** in order to **simplify** the **category creation** process. Create the getters and setters:



It's time for annotations:



The id will have the usual "@Id" and "@GeneratedValue" annotations. 

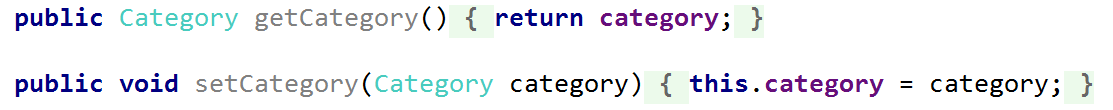
We want each **category to be unique**. We will define the **relation** with the **article** entity next.

## Creating Article-Category Relation

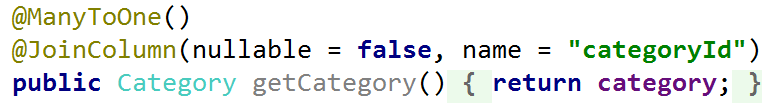
Open your Article entity. We need to add a new private field that will keep the article category:



And we will need getter and setter for it:



Time to talk about the relation between the two entities. **Each** **category** **can contain many articles**. That means that our relation will be **One-To-Many**. Create the following annotation:



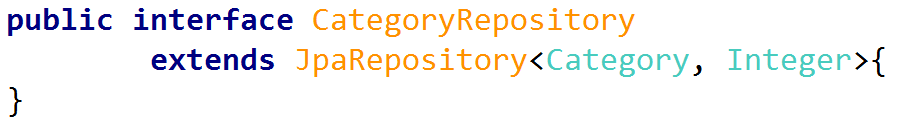
Here we define that our **article** **must** **have** **category**.

## Create Category Repository

Create a new interface called CategoryRepository:



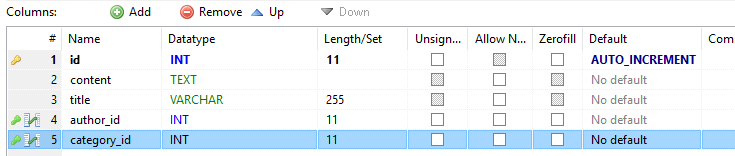
Extend JpaRepository:



Start your project now. In the database, you should see the **new table**:



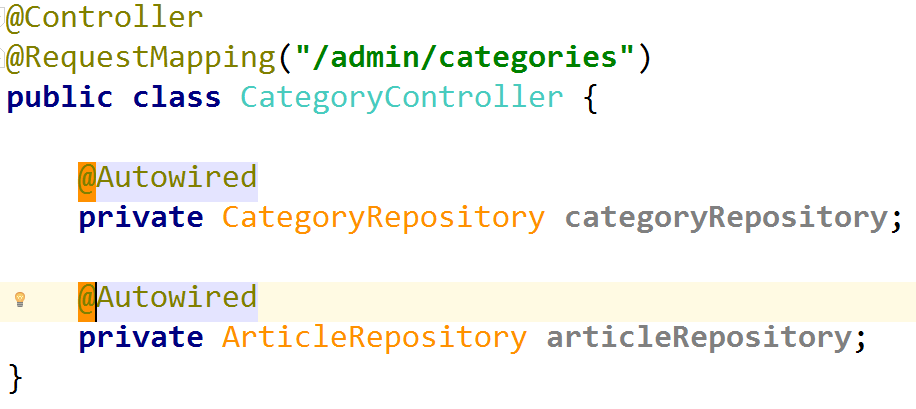
Additionally, in the articles table you will see the **new column**:



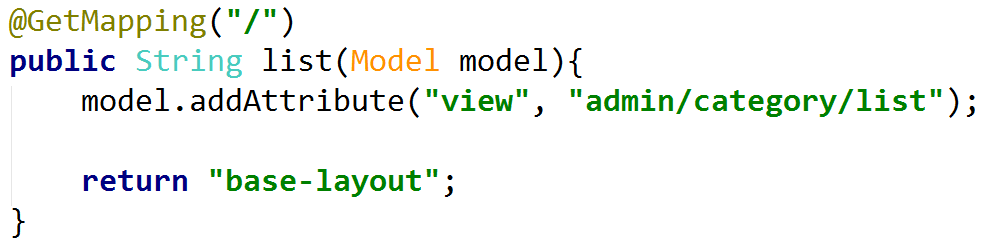
It looks like it should work.

## Listing Categories

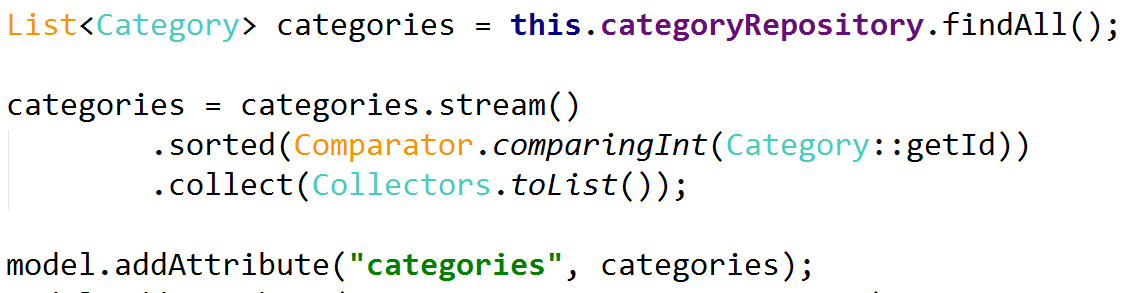
The time has come. We need to **extend our admin functionality** in order to use categories. In the admin package create a new CategoryController, which will have the following repositories:



We will start by creating the listing method:



Before we go to our view, we need to **get all categories** and send them as well:



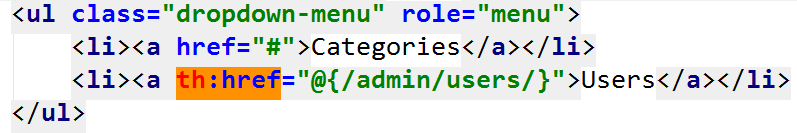
Here we are getting our categories and **sorting them** by id, because if we don't they will be **sorted alphabetically**.

For our view, we need to create **new folder** "category" in our existing "admin" folder. Inside, create a new html called "list" and delete its contents. Use the following code:

|  |
| --- |
| <**main**>  <**div class="container body-content"**>  <**div class="well"**>  <**h2**>All Categories -  <**a th:href="@{/admin/categories/create}" class="btn btn-warning"**>Create New</**a**>  </**h2**>  <**div class="row"**>  <**table class="table table-striped table-hover "**>  <**thead**>  <**tr**>  <**th**>#</**th**>  <**th**>Name</**th**>  <**th**>Actions</**th**>  </**tr**>  </**thead**>  <**tbody**>   <**tr th:each="category : ${categories}"**>  <**td th:text="${category.id}"**></**td**>  <**td th:text="${category.name}"**></**td**>  <**td**>  <**a th:href="@{/admin/categories/edit/{id}(id = ${category.id})}"**>Edit</**a**>  <**a th:href="@{/admin/categories/delete/{id}(id = ${category.id})}"**>Delete</**a**>  </**td**>  </**tr**>  </**tbody**>  </**table**>  </**div**>  </**div**>  </**div**> </**main**> |

This view will foreach the categories and it will **create a table**. Keep in mind that the **create**, **edit** and **delete** buttons **won't work** at this point in time.

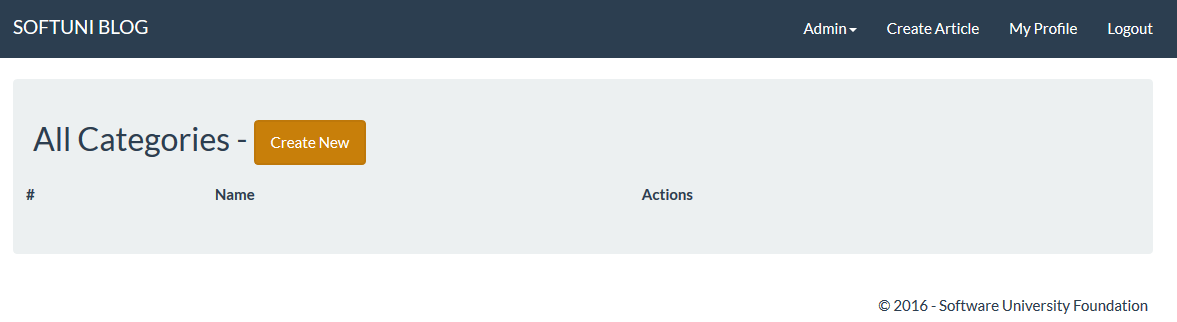
We need to fix our **admin drop-down** as well. Open the fragments/header file and find the following code:



Change the category hyperlink like that:



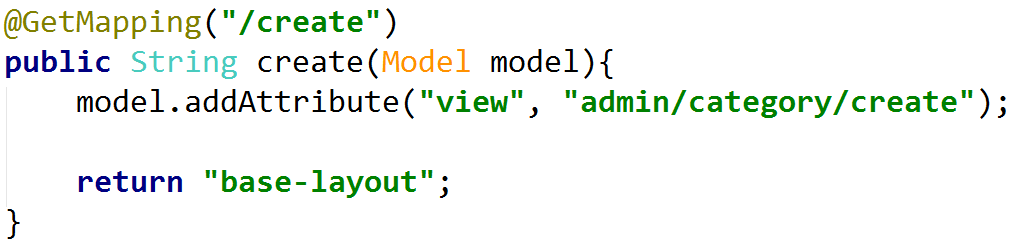
Run the blog. When you open the categories page, you will see this:



Our next step is to create some categories.

## Creating Categories

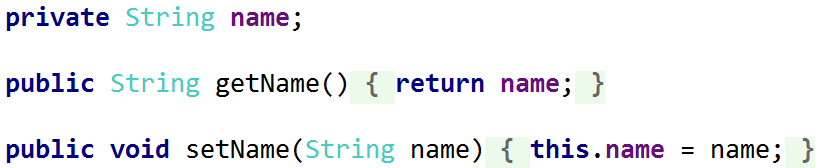
Create a new method in our CategoryController:



This method will only **return our creation form**. Before we try to create a new category, we need to create the binding model that will transfer the data from our form. Create a **new binding model** called "CategoryBindingModel":



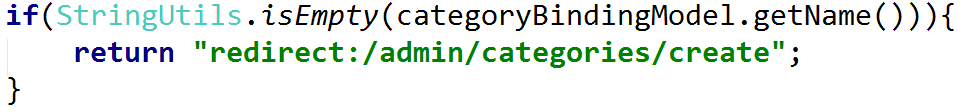
It should only contain **one field** for the category name and **getter/setter** for it:



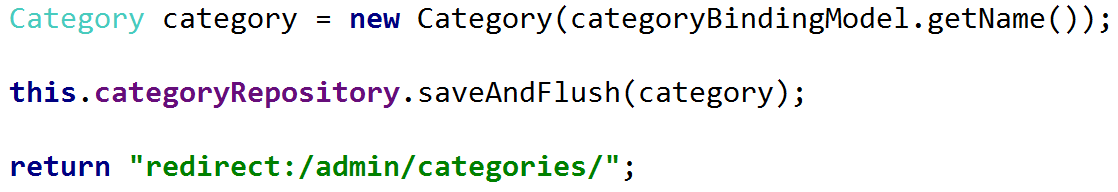
Now in our CategoryController, we should create a **new method** that will actually **create the category**:



First, we need to check if the **field is empty** and if it is, we should redirect the user:



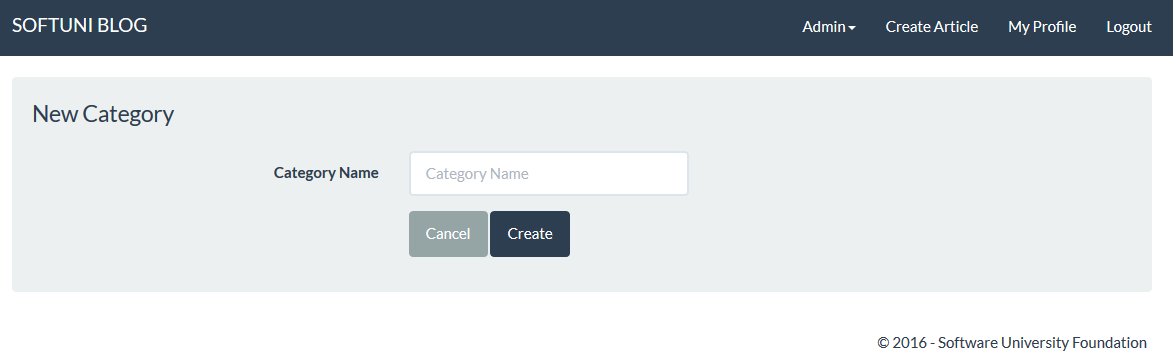
If the field contains a category name, we should **create a new category** and **save it to our database**:



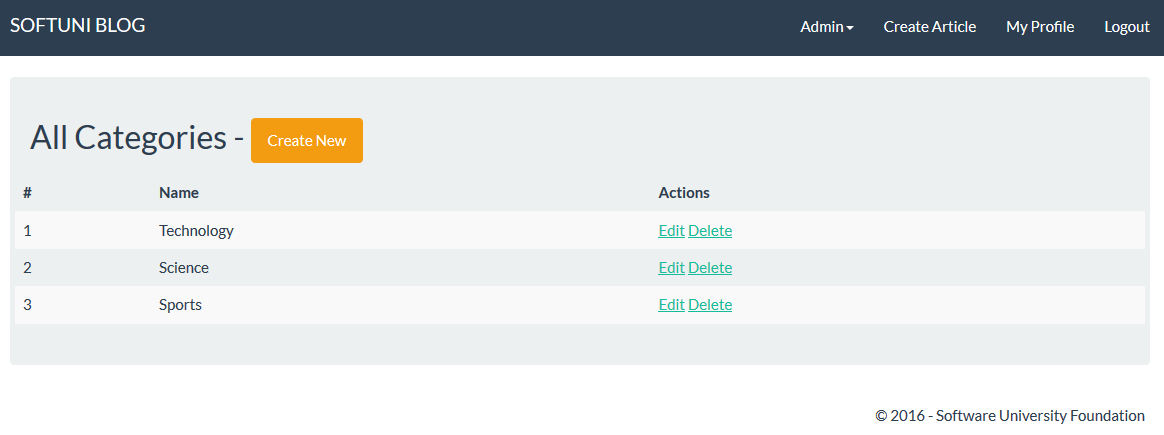
It's time to create the view. In the category folder create a new html called "create". Use the following code:

|  |
| --- |
| <**main**>  <**div class="container body-content span=8 offset=2"**>  <**div class="well"**>  <**form class="form-horizontal" th:action="@{/admin/categories/create}" method="POST"**>  <**fieldset**>  <**legend**>New Category</**legend**>   <**div class="form-group"**>  <**label class="col-sm-4 control-label" for="category\_name"**>Category Name</**label**>  <**div class="col-sm-4 "**>  <**input type="text" class="form-control" id="category\_name" placeholder="Category Name" name="name"**/>  </**div**>  </**div**>   <**div class="form-group"**>  <**div class="col-sm-4 col-sm-offset-4"**>  <**a class="btn btn-default" th:href="@{/admin/categories/}"**>Cancel</**a**>  <**input type="submit" class="btn btn-primary" th:value="Create"**/>  </**div**>  </**div**>  </**fieldset**>  </**form**>  </**div**>  </**div**> </**main**> |

Really basic form containing only one input field. Run the blog and you should see this:



You should be able to create categories:



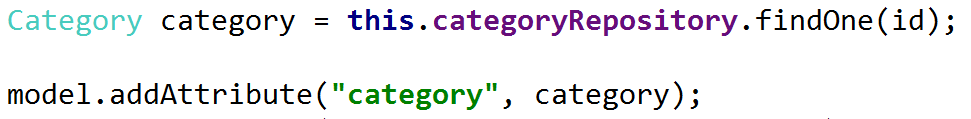
Now we need to create the **edit** functionality.

## Editing Categories

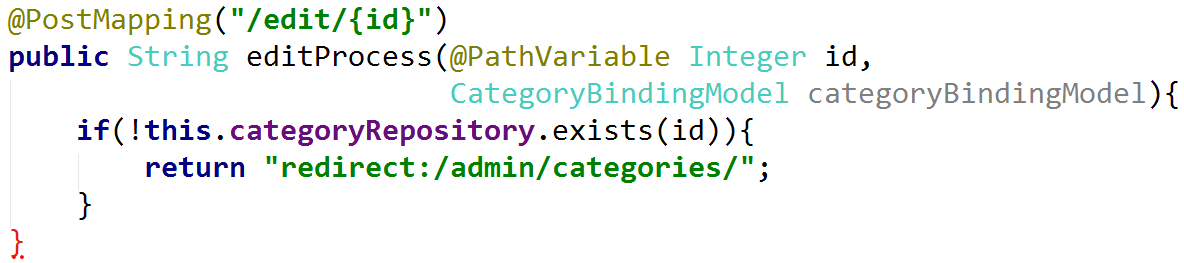
Now we need to **create another method** that will give us the **edit** form:



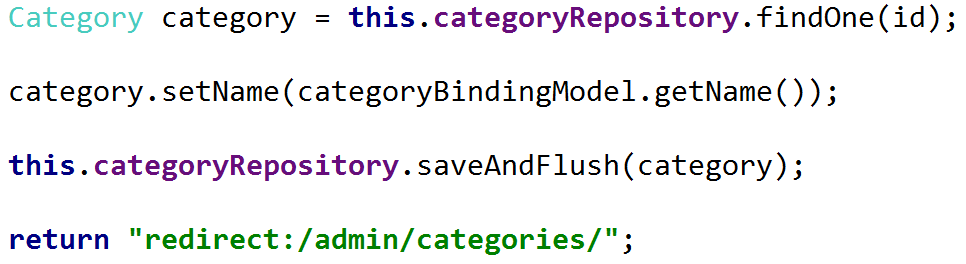
We are redirecting the admin if such category **doesn't exist**, but we need to get the category if it does:



Create **another method** that will **edit** our category:



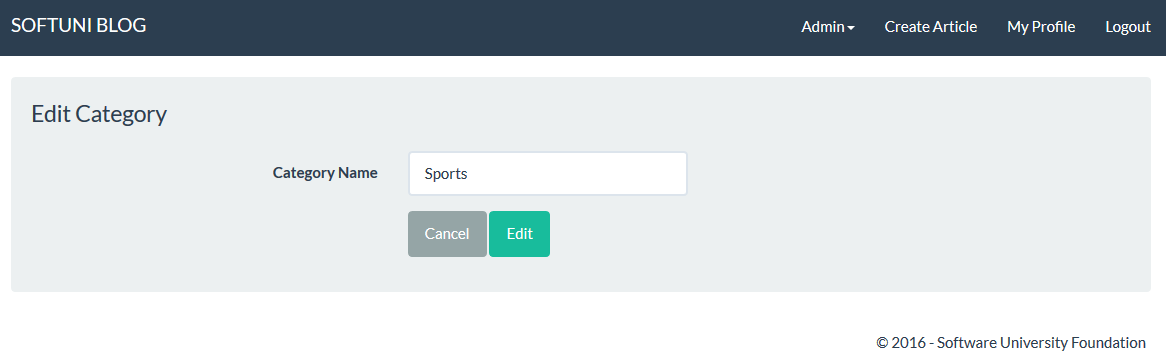
We are again verifying that the category exists. Now we need to **get the category**, **change the name** and **save it to our database**:



The only thing left to do is create the view. Create a new html called "edit" in our category folder. Replace the default html with this one:

|  |
| --- |
| <**main**>  <**div class="container body-content span=8 offset=2"**>  <**div class="well"**>  <**form class="form-horizontal" th:action="@{/admin/categories/edit/{id}(id=${category.id})}" method="POST"**>  <**fieldset**>  <**legend**>Edit Category</**legend**>   <**div class="form-group"**>  <**label class="col-sm-4 control-label" for="category\_name"**>Category Name</**label**>  <**div class="col-sm-4 "**>  <**input type="text" class="form-control" id="category\_name" placeholder="Category Name" name="name" th:value="${category.name}"**/>  </**div**>  </**div**>   <**div class="form-group"**>  <**div class="col-sm-4 col-sm-offset-4"**>  <**a class="btn btn-default" th:href="@{/admin/categories/}"**>Cancel</**a**>  <**input type="submit" class="btn btn-success" value="Edit"**/>  </**div**>  </**div**>  </**fieldset**>  </**form**>  </**div**>  </**div**> </**main**> |

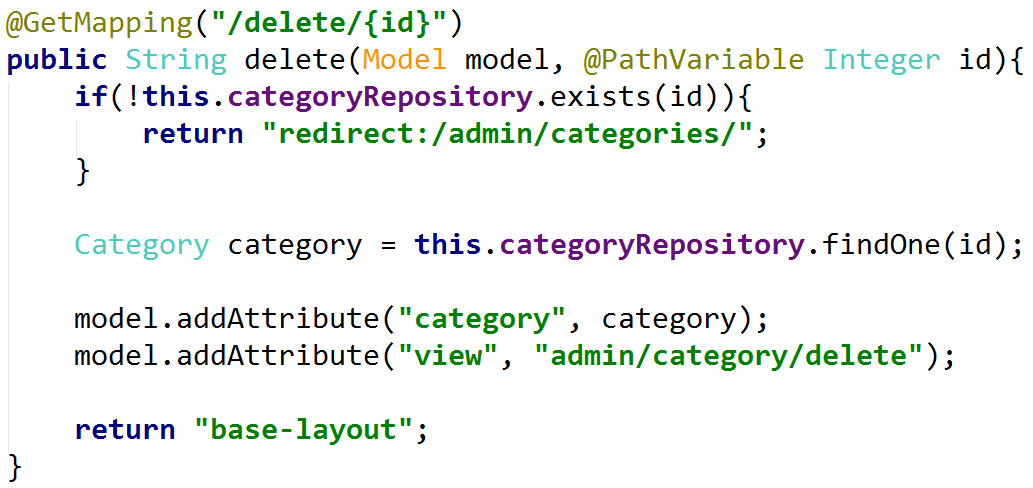
It should look like that:



It should work without any problems. Next on the list is the **delete functionality**.

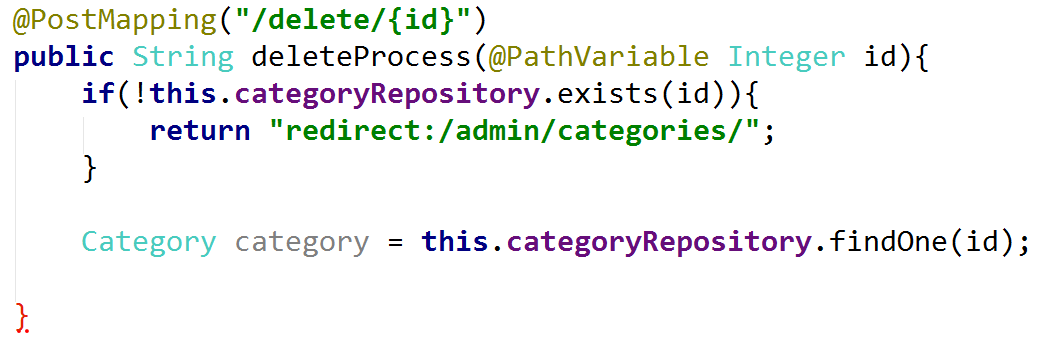
## Deleting Categories

Start by creating new method that will return our view:

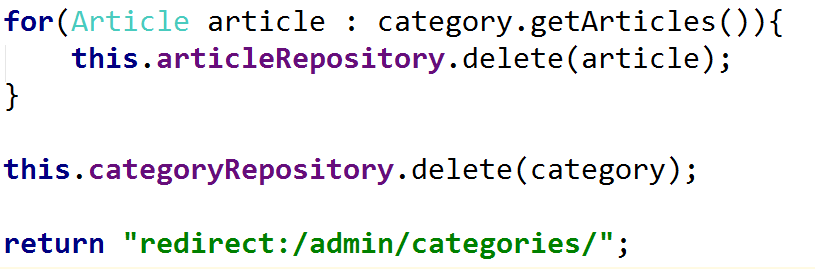


Basically, it's the **same method as the one that returns the edit view**. The only difference is the view that we are sending to the layout.

The method that will delete the category is **more interesting**. We start like usual:



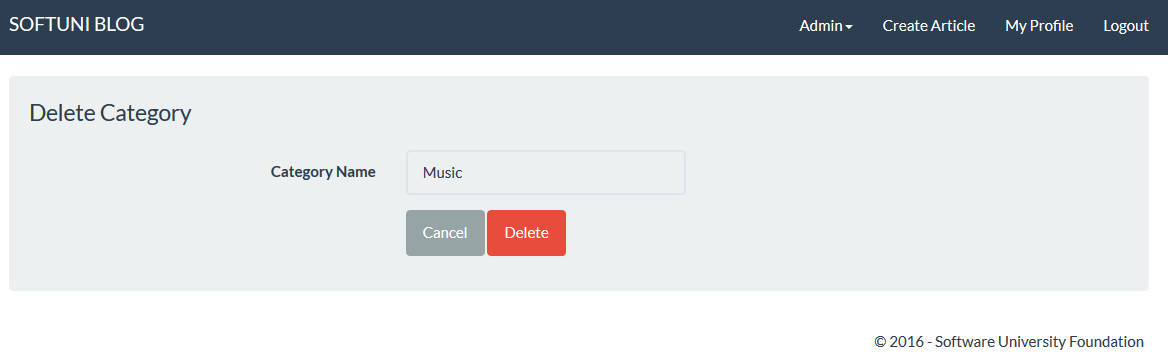
Why it's more interesting? Because we **cannot delete a category**, without **deleting all of its articles**. That's why we need to **first delete all of the articles** and **then delete the category** itself:



Now we need to create the view. Create a new view called delete and use the following code:

|  |
| --- |
| <**main**>  <**div class="container body-content span=8 offset=2"**>  <**div class="well"**>  <**form class="form-horizontal" th:action="@{/admin/categories/delete/{id}(id=${category.id})}" method="POST"**>  <**fieldset**>  <**legend**>Delete Category</**legend**>   <**div class="form-group"**>  <**label class="col-sm-4 control-label" for="category\_name"**>Category Name</**label**>  <**div class="col-sm-4 "**>  <**input type="text" class="form-control" id="category\_name" placeholder="Category Name" name="name" th:value="${category.name}" disabled="disabled"**/>  </**div**>  </**div**>   <**div class="form-group"**>  <**div class="col-sm-4 col-sm-offset-4"**>  <**a class="btn btn-default" th:href="@{/admin/categories/}"**>Cancel</**a**>  <**input type="submit" class="btn btn-danger" value="Delete"**/>  </**div**>  </**div**>  </**fieldset**>  </**form**>  </**div**>  </**div**> </**main**> |

The delete is working and it should look like this:



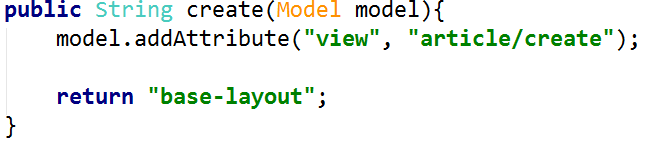
With that done, we need to fix the article creation process now. Currently we **can't create** or **edit** **articles** properly.

## Choosing Category on Article Create

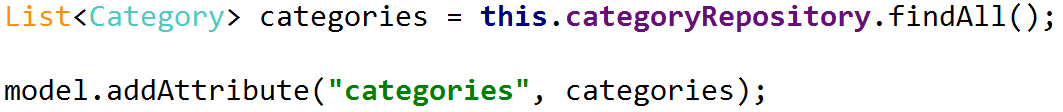
We need to show a **drop-down menu** with **all categories** each time we're trying to **edit** or **create** **article**. To do that, we need to go to our ArticleController. We will need to use categories and that's why we need our category repository:



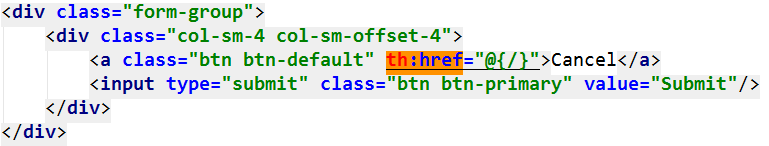
Our first stop will be the **method** that **returns** the **create view**:



Here, we need to get all of the categories and send them to the view:



Next, we need to **show all of the categories** in our view. Open the article/create view and find the following:

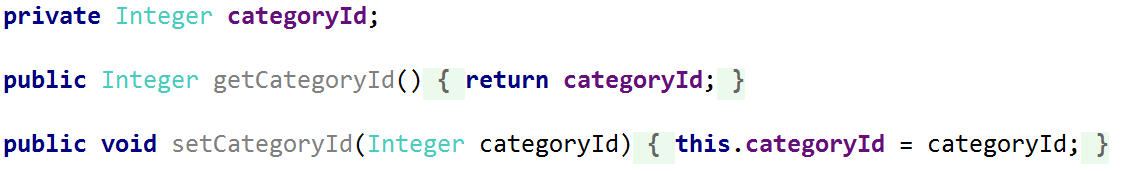


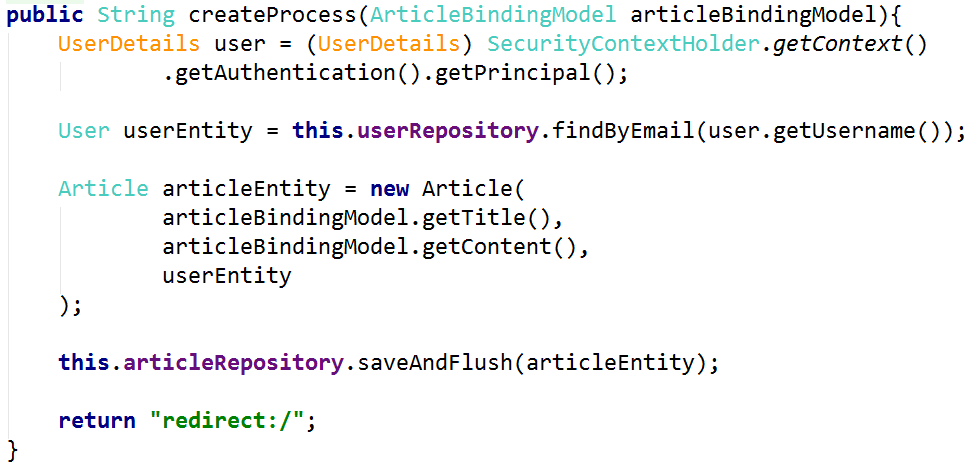
Those are the 2 buttons we have in our form. **Above** the "form-group" **div tag**, add the following code:

|  |
| --- |
| <**div class="form-group"**>  <**label class="col-sm-4 control-label" for="article\_category"**>Category</**label**>  <**div class="col-sm-6"**>  <**select class="form-control" id="article\_category" name="categoryId"**>  <**option th:each="category : ${categories}" th:value="${category.id}" th:text="${category.name}"**></**option**>  </**select**>  </**div**> </**div**> |

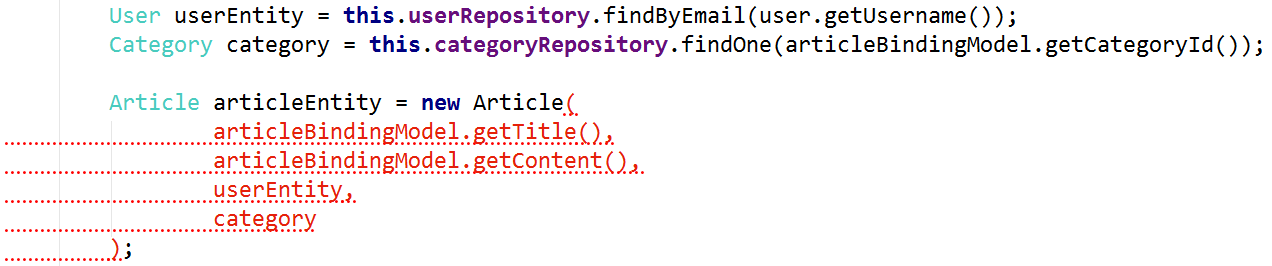
This code will foreach our categories and fill them as options of select box. The **value** of each option will be the id of the category and the **text** will be name.

In order to use the category from our form, we need to **edit** our ArticleBindingModel. We need to create a **private field** that will keep the chosen category id and **getter/setter** for it:

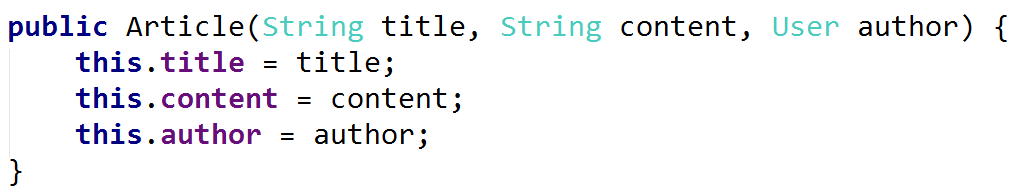
 Now, we need to change the how the article is being created. Find the method that is called on "POST" request:



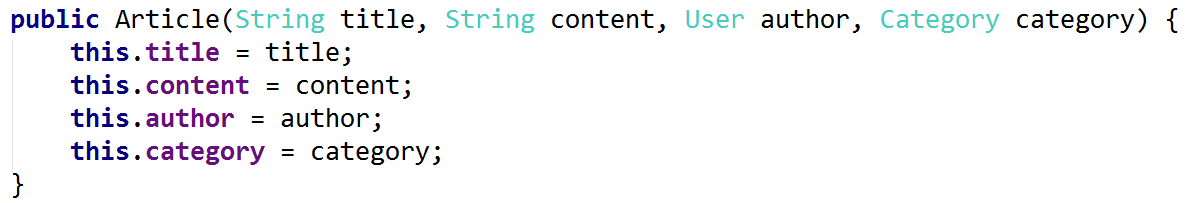
Here, when we get the user, we should **get the category** as well. Then, send it to the **constructor**:



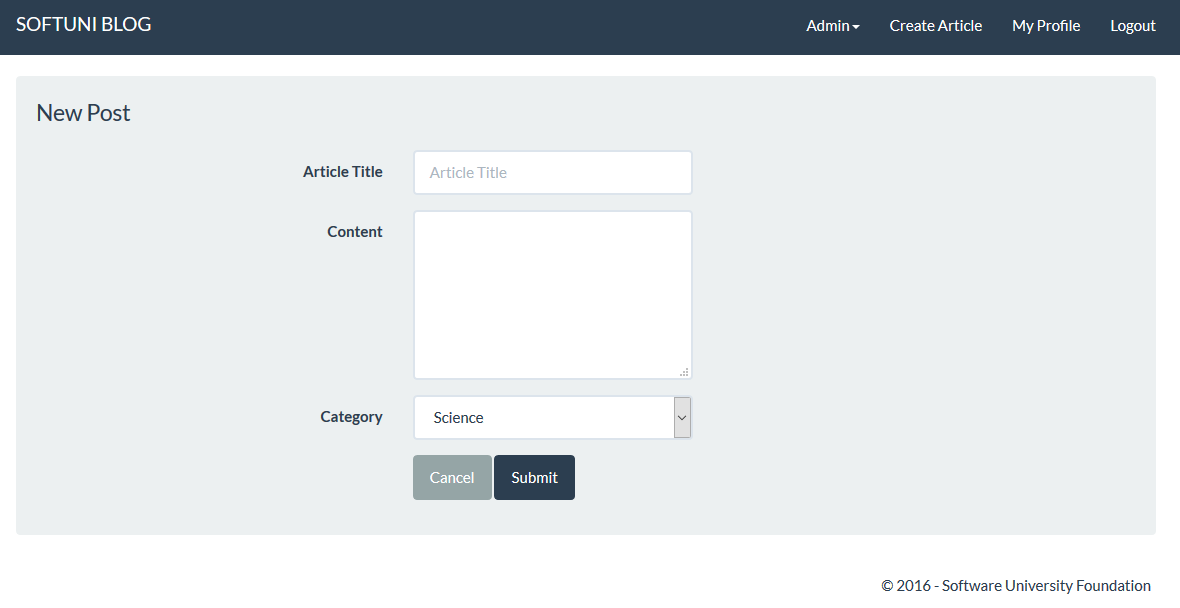
Don't worry that everything goes red. We will fix that in a moment. Open the Article entity and find this **constructor**:



We need to add the category to it:



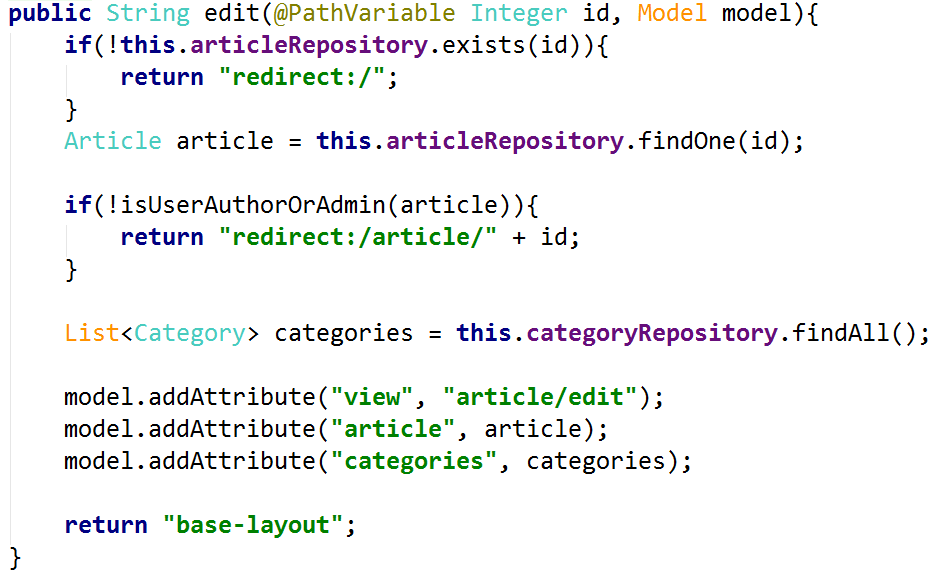
This will **fix the problem** we've had in the controller. Let's try to create a new article. The view should look like this:



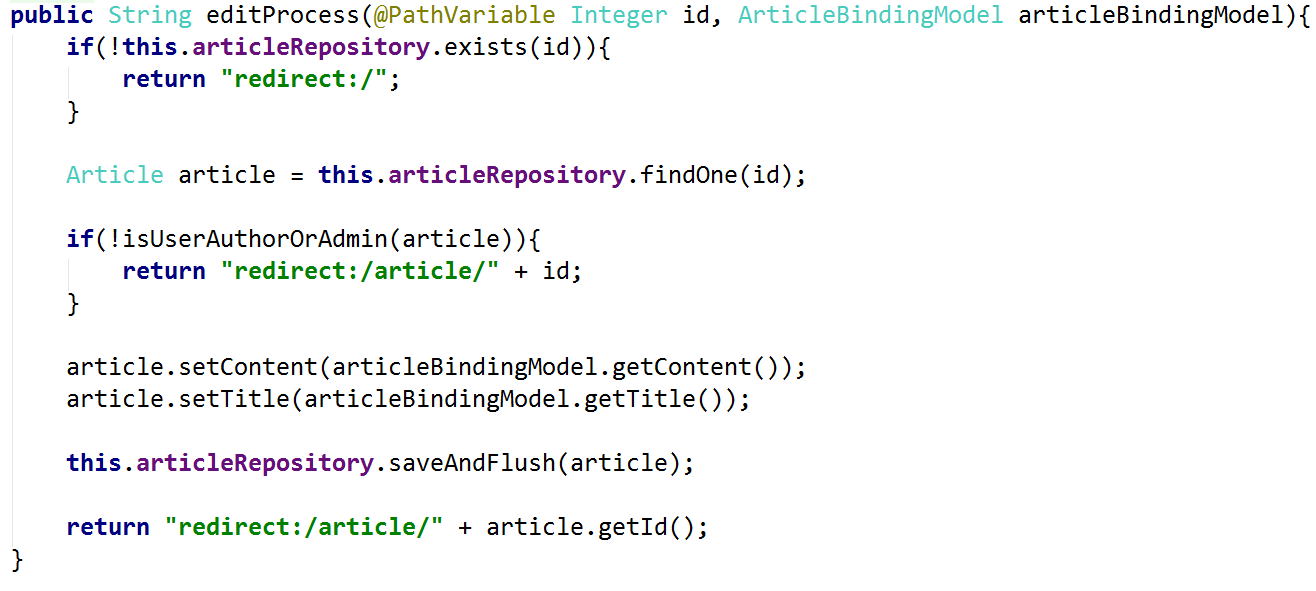
We have our drop-down that **contains all of our categories**. We can now **create articles** once again.

## Choosing Category on Article Edit

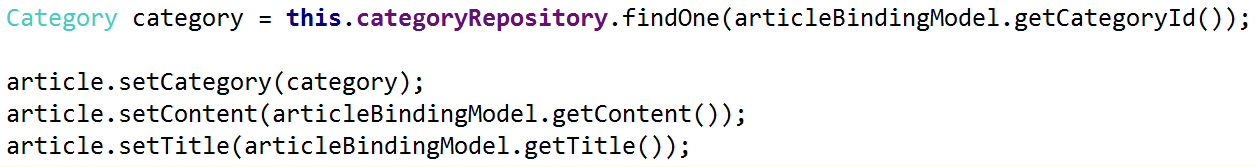
Our changes in this part will be similar to the changes we've made earlier. The **edit** method that return the view will need to **return all of the categories**:



The method that actually **edits the article** looks like this now:



The only thing that changes here is that before we set the new title and content we will **get the category** from our database. Then we will **set it to the article**:

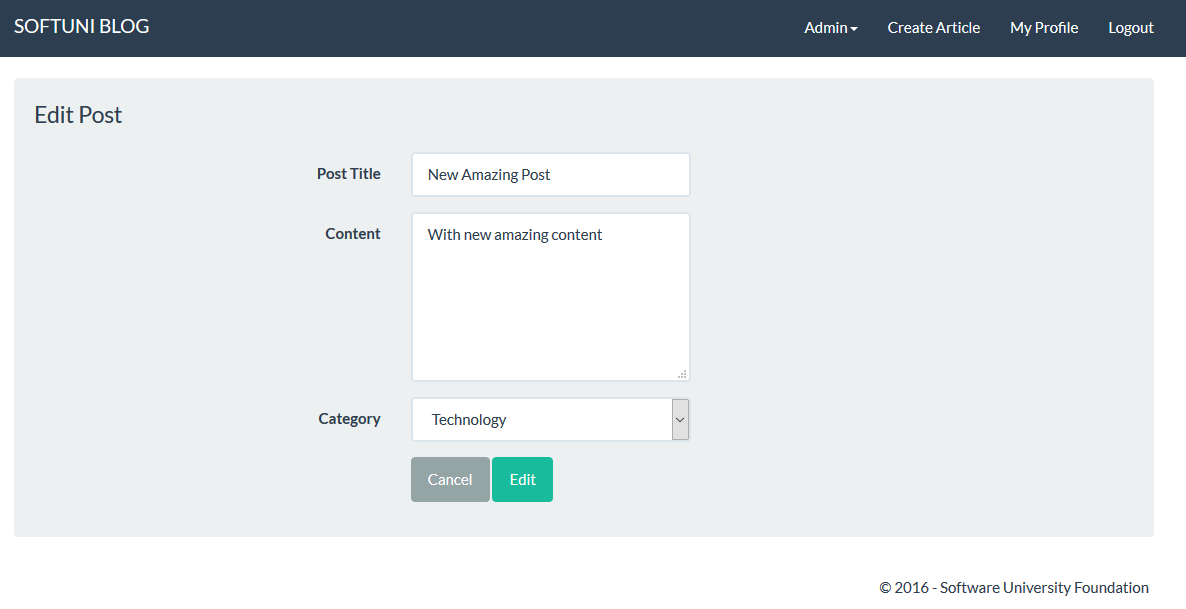


Finally, we need to **edit** the view. Open the article/edit view and above the last "form-group" **div** **tag**, add the following code:

|  |
| --- |
| <**div class="form-group"**>  <**label class="col-sm-4 control-label" for="article\_category"**>Category</**label**>  <**div class="col-sm-6"**>  <**select class="form-control" id="article\_category" name="categoryId"**>  <**option th:each="category : ${categories}" th:value="${category.id}" th:text="${category.name}" th:selected="${category.id == article.category.id}"**></**option**>  </**select**>  </**div**> </**div**> |

Again, we're adding the select with options. The difference here is the "th:selected" attribute. This attribute will make the **default value** of the **drop-down** the **article current category**. We are comparing the **article category id** with the id of the **category** we are currently adding to the list. **If they match**, the **current category should be selected by default**.

Try it out, it should look like this:

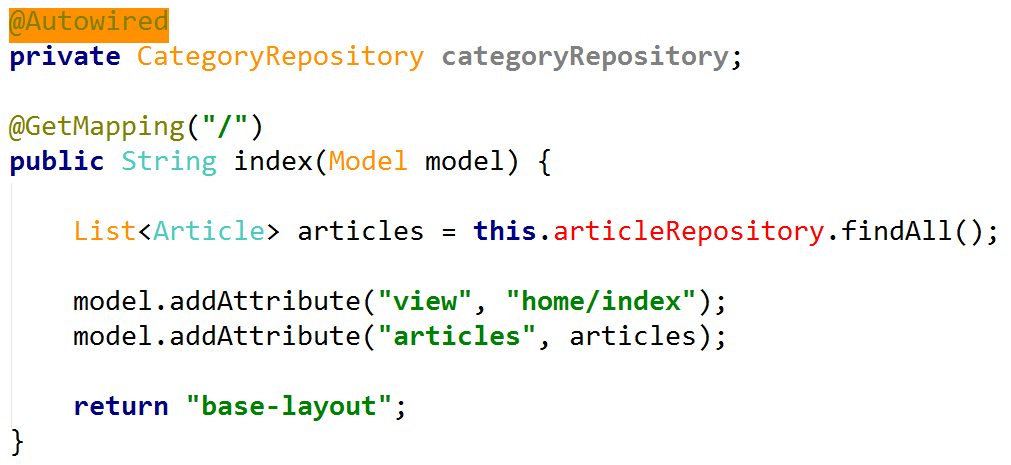


Not bad. Now we need to **separate the articles by category**.

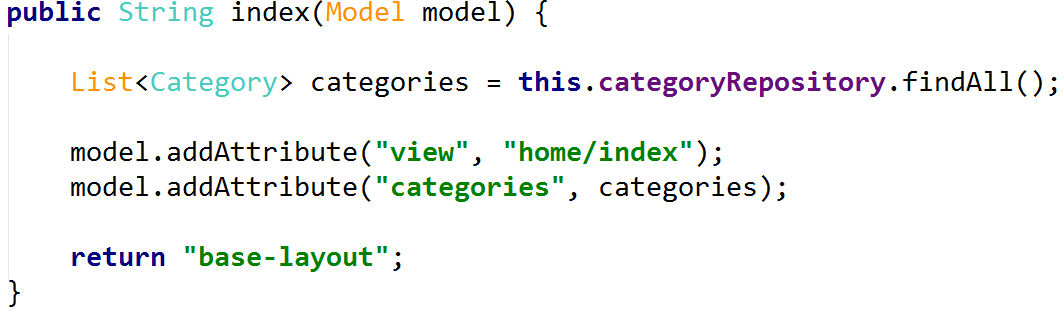
## Changing the Home Page

Our home page currently shows us all of the articles. However, we don't want that. We want to see **all of the categories**.

Open the HomeController. First, we can **replace** the ArticleRepository with CategoryRepository:



We need to edit that index() method. **Instead** of getting **all articles**, we will get **all categories** and send them to the view:

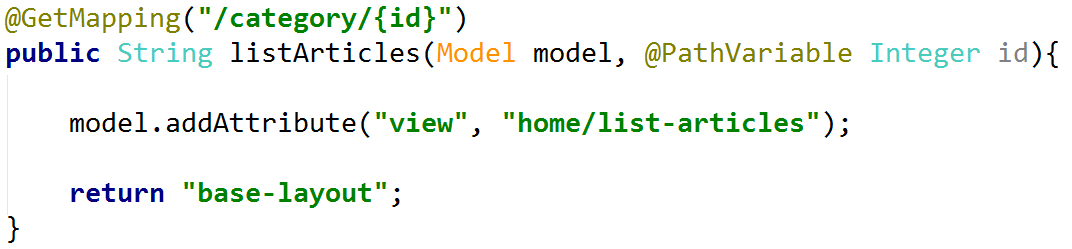


Now, that we've changed the articles with categories, our view won't work. Open the home/index view. **Replace the existing code** with this:

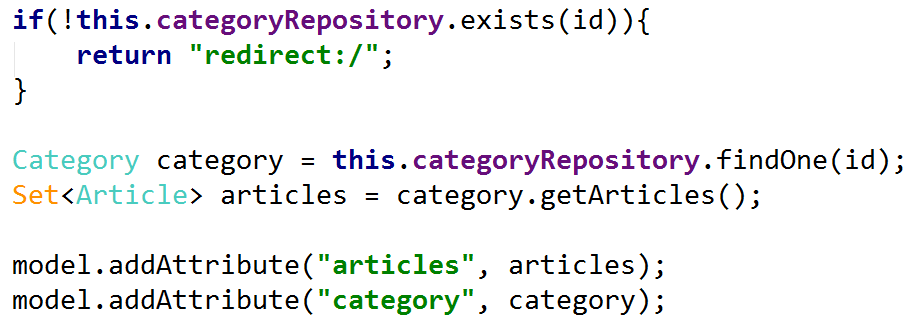
|  |
| --- |
| <**main**>  <**div class="container body-content"**>  <**div class="row"**>  <**div class="col-md-6" th:each="category : ${categories}"**>  <**header**>  <**h2**>  <**a th:href="@{/category/{id}(id=${category.id})}" th:text="${category.name + ' (' + #sets.size(category.articles) + ')'}"**>  </**a**>  </**h2**>  </**header**>  </**div**>  </**div**>  </**div**> </**main**> |

Take a closer look at "#sets.size()". This is a function in Thymeleaf that gives us the ability to get the **size of a collection** in our html. You can read more about [sets here](http://www.thymeleaf.org/apidocs/thymeleaf/2.1.4.RELEASE/org/thymeleaf/expression/Sets.html). There is also [lists](http://www.thymeleaf.org/apidocs/thymeleaf/2.0.2/org/thymeleaf/expression/Lists.html) helper.

Now, we need to **list the articles** in a given category. Create a new method called "listArticles" in our HomeController:



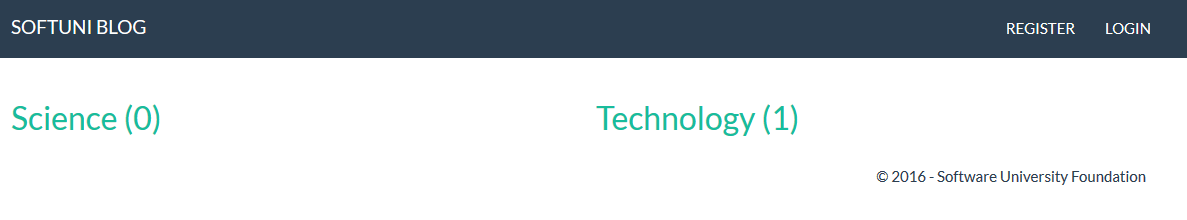
In this method, we need to get **all articles in a category**. That's why we are getting **category id** as a parameter. First, we need to **get the category if it exists** and then **get all of the articles**. Finally, **send** the **articles** and the **category** to the **view**:



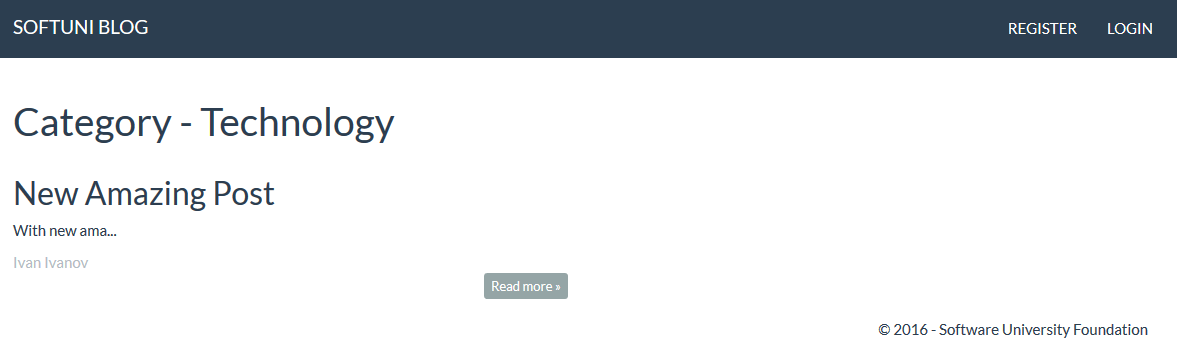
Speaking of the view, we need to create it. In the "home" folder create a new view called "list-articles". Use the following code:

|  |
| --- |
| <**main**>  <**div class="container body-content"**>  <**h1**>Category - <**th:block th:text="${category.name}"**></**th:block**></**h1**>  <**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 more **&raquo;**</**a**>  </**div**>  </**footer**>  </**article**>  </**div**>  </**th:block**>  </**div**>  </**div**> </**main**> |

We are done. Now, our **home page** should look like this:



When you **click on a category**, you should see this view:

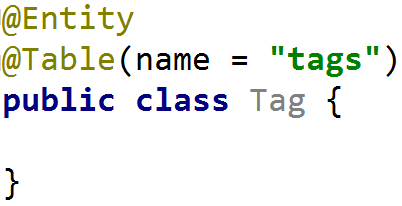


Looking good. We are ready with the categories. **Next** and **final** chapter – **tags**.

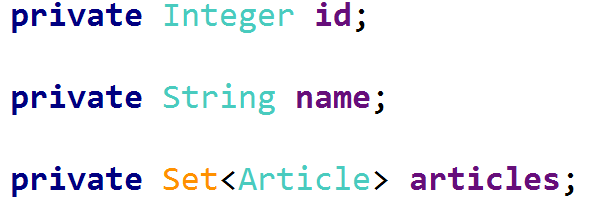
# Tags

## Create Tag Entity

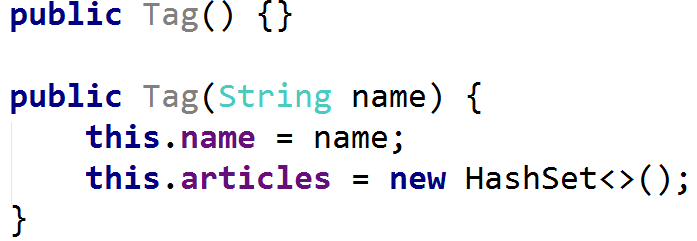
We need to create one last entity called Tag:



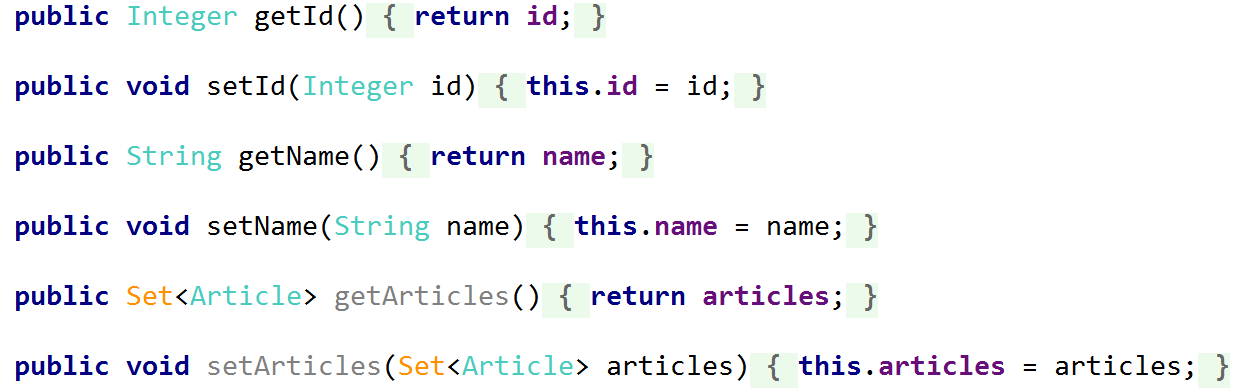
It will only have **name** and **collection of unique arti**cles:



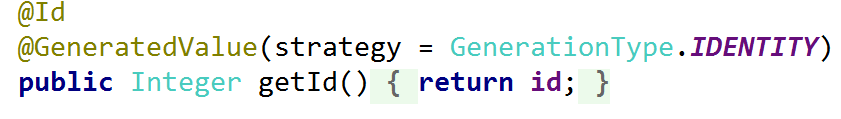
We will also **create constructors** that will make it easier to work with tags:



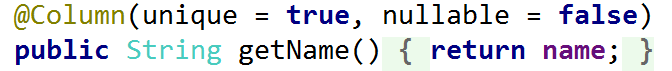
As you've probably figured out, we will need **getters and setters**:



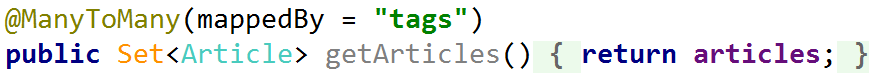
Now, we need the annotations. The id will have the usual ones:



The **name** of each tag **must be unique**:



Finally, the Many-To-Many relation with our **articles** will be defined like this:



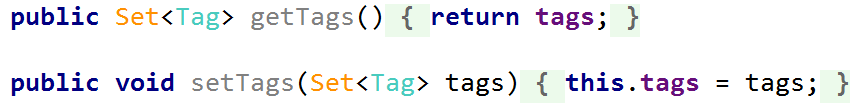
We need to finish the relation in our Article entity.

## Article-Tag Relation

In our Article entity, we need to add new **private field** that will keep the tags:



We will also need getter and setter for it:



Now, we need to specify the **table name** for our relation:



That's all. Now we need repository for our tags.

## Create Tag Repository

You should be familiar with the process. Create an interface called TagRepository that **extends** JpaRepository:



Take a look at the **import**. We will **only use this import** every time we are using tags! In the repository **define a new method** that will **search for tag** by **name**:



Now you can start the application. If you check the database, you should have **two new tables**:



That means that **everything is working** for now.

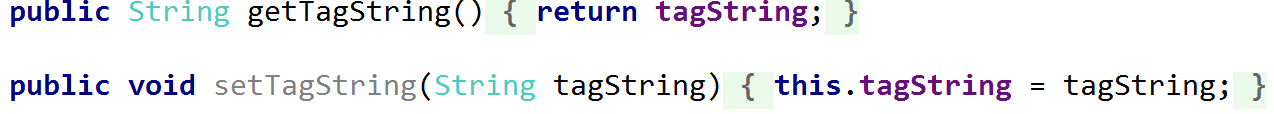
## Creating Tags

This is the **first entity** that **won't have controller** for **create**, **edit** and **delete** methods. That is because we will **create the tags** **when we create article**. We **won't edit** and **delete** **tags**, we will just **detach them from an article**.

First, we will add a **new field** to our ArticleBindingModel:



Create getter and setter for it:



We need to edit the view as well. We will **add a textbox** **below our category drop-down list** in both our **edit** and **create** views. Open the article/create view and **above the buttons** add the following code:

|  |
| --- |
| <**div class="form-group"**>  <**label class="col-sm-4 control-label" for="article\_tags"**>Tags</**label**>  <**div class="col-sm-6"**>  <**input type="text" class="form-control" id="article\_tags" placeholder="Tags" name="tagString"**/>  </**div**> </**div**> |

In the article/edit view add the following code, just above the buttons:

|  |
| --- |
| <**div class="form-group"**>  <**label class="col-sm-4 control-label" for="article\_tags"**>Tags</**label**>  <**div class="col-sm-6"**>  <**input type="text" class="form-control" id="article\_tags" placeholder="Tags" name="tagString" th:value="${tags}"**/>  </**div**> </**div**> |

Our views are ready. Let's move to the ArticleController.

First, create **new private field** for our **tag repository**:



Now, create a new method:



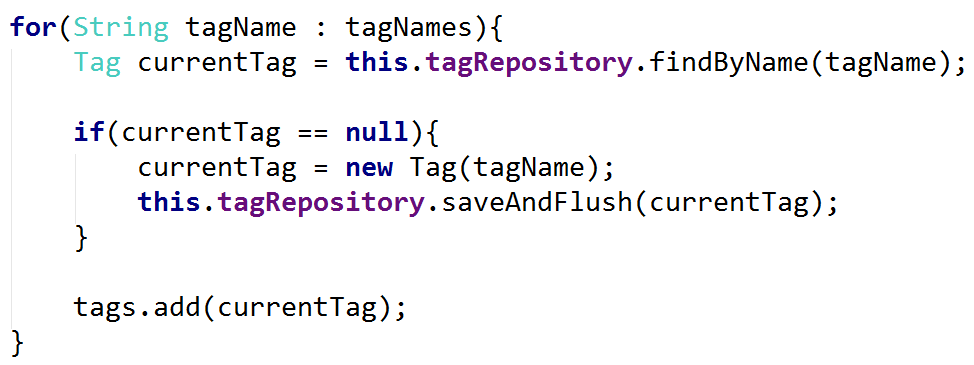
This will be a **helper method** that will **take a string** and **extract all tags** from it. Initialize new HashSet that will keep our tags:



Now **split our string** by **comma** and **empty space** (using RegEx):



Next step:



Here, we **try to find the tag** in our **database**. If such **tag** **doesn't exist**, we **create a new tag** with the given **name** and **add it to our collection**. Finally, we will **return our collection**:

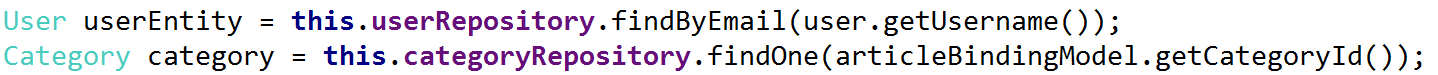


That's all **our method should work** now.

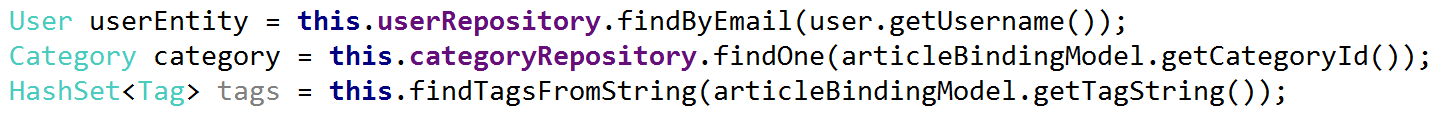
Find the method **that creates the article**:



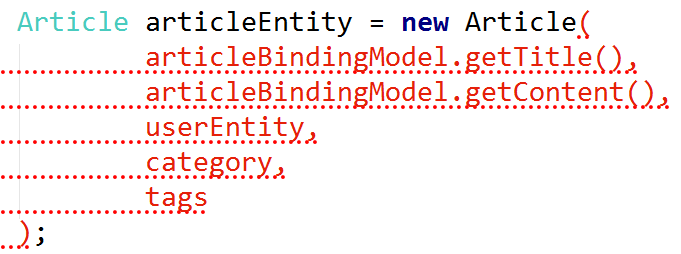
Currently, we are **getting** the **user** and the **category**:



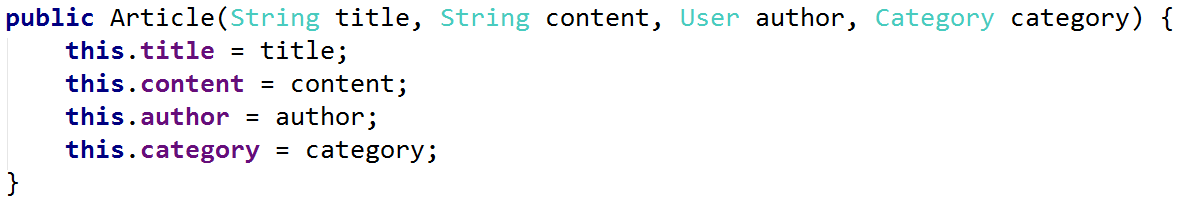
We need to add one more line that will get the **tags**:



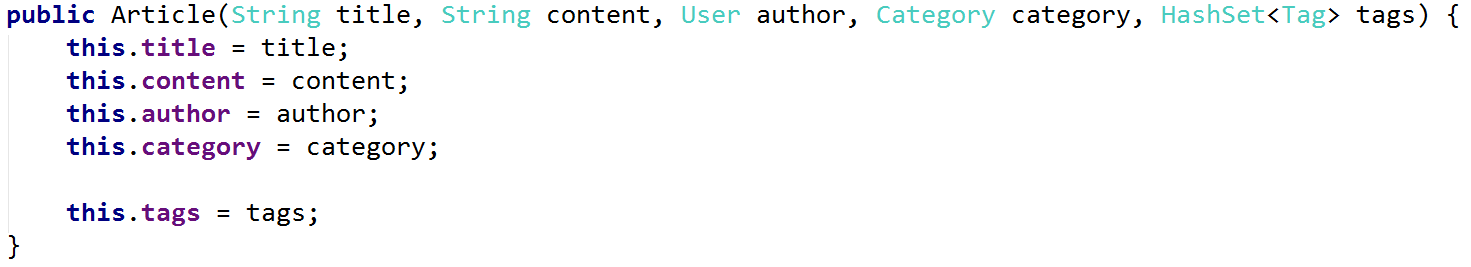
Then, we need to give it to the **constructor**:



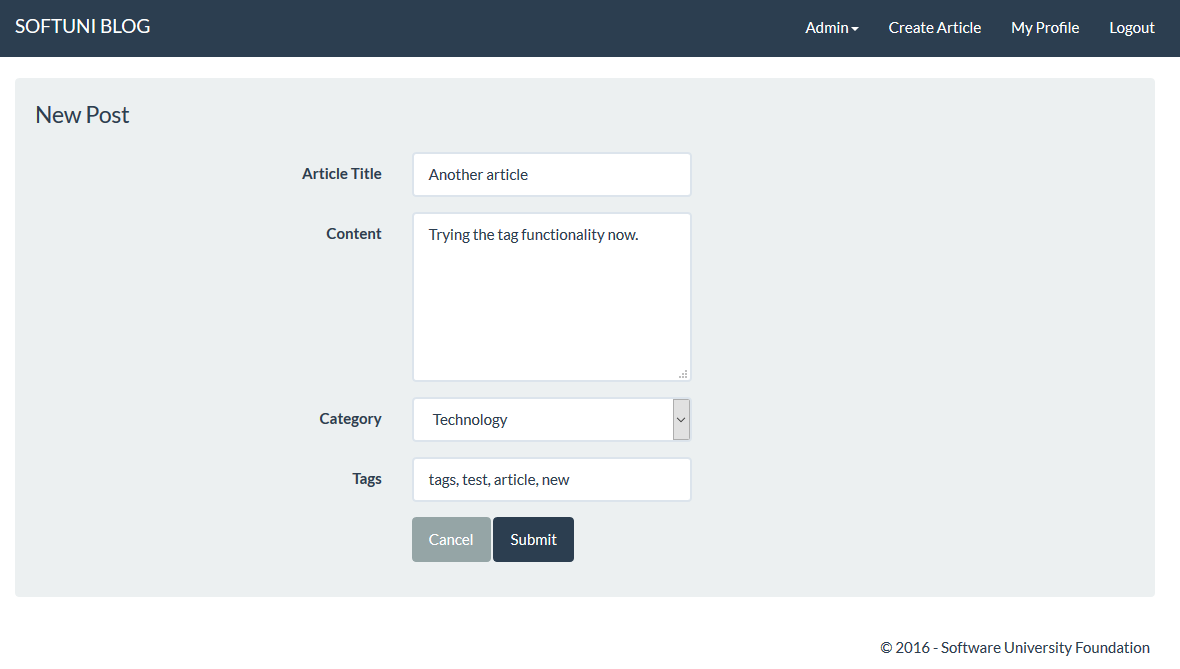
However, our **constructor** is **not suited** for that many parameters, so we need to **edit** **it**. Open the Article entity and **find** **the** **constructor**:



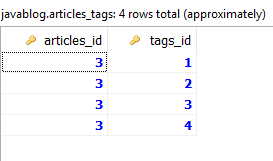
Add another parameter for our tags:



Let's try to create new article now:



Taking a **quick look at the database** will tell us that it works:



We need to fix the **article edit** as well. Find the **get method**:



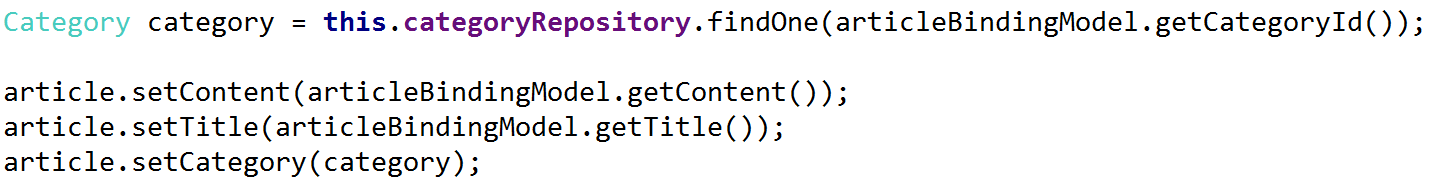
In this method, add the following code:



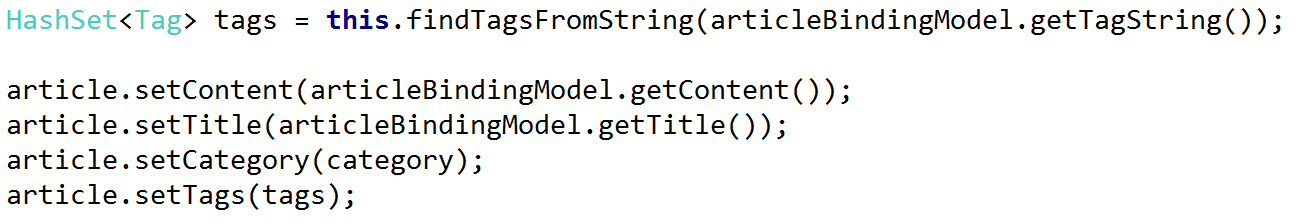
You should have **some of the model command**s from earlier. The important part here is that we are getting the **tags** for the **article** and **joining them together**, using the **Stream API** in **Java 8**. This should be similar to the string.Join() in **C#**. Finally, we need to find the **post edit method**:



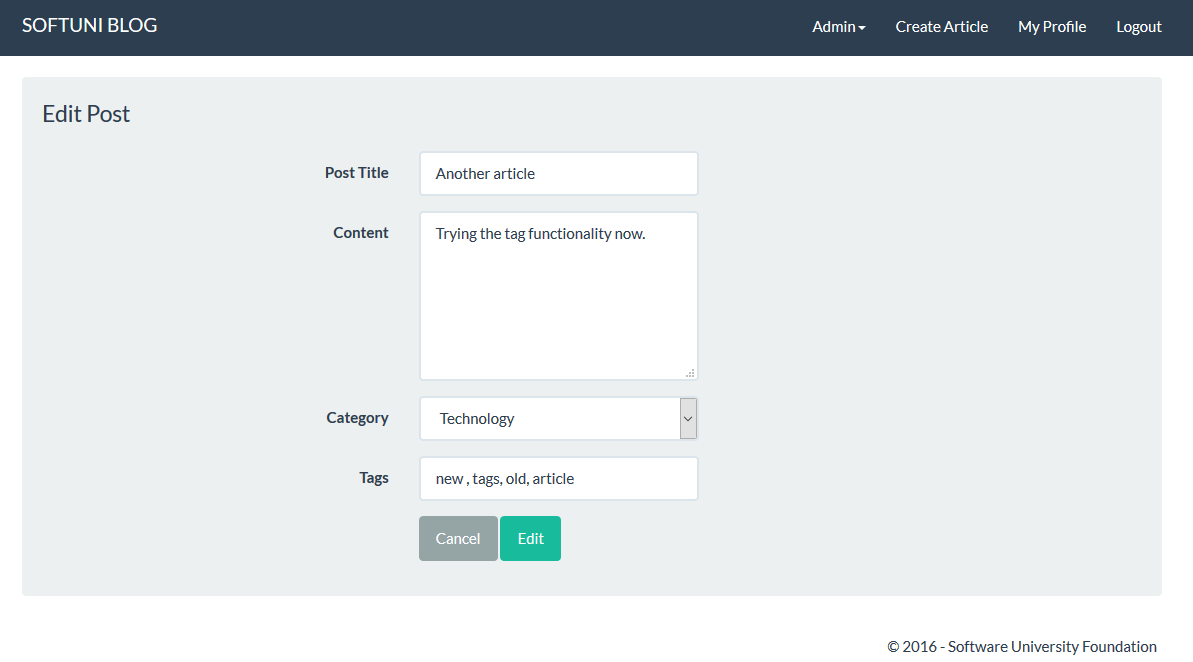
Here, we will do the same thing we've done in the create method, except for the fact that we are going to **set the tags**, not pass them using the **constructor**. Find this code:



Get the tags and set them to our article:



This should be everything. **Run the blog** and check if you have this:



It works. Let's **display the tags** in our blog.

## Displaying the Tags

Open the article/details view. Find this part:



Below it, add the following code:

|  |
| --- |
| <**p**>  <**a class="btn btn-default btn-xs" th:each="tag:${article.tags}" th:href="@{/tag/{name}(name=${tag.name})}" th:text="${tag.name}"**></**a**> </**p**> |

This should add our tags as buttons bellow the article.

Open the home/list-articles view and find the following line:



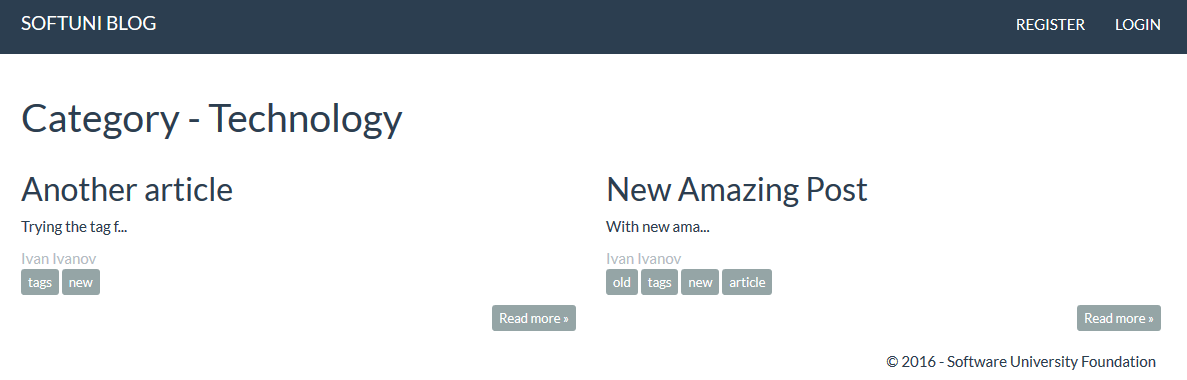
Below it, add the same code, we've used earlier:

|  |
| --- |
| <**p**>  <**a class="btn btn-default btn-xs" th:each="tag:${article.tags}" th:href="@{/tag/{name}(name=${tag.name})}" th:text="${tag.name}"**></**a**> </**p**> |

In this code, we are printing **each tag** as a **different button**, that will help us make the next functionality – **searching by tags**.

The views should be ready now.

**Category view**:



**Details view**:



Final part incoming!

## Searching by Tags

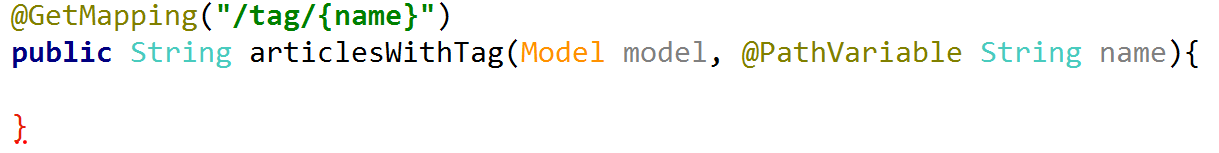
Create a new controller called TagController:



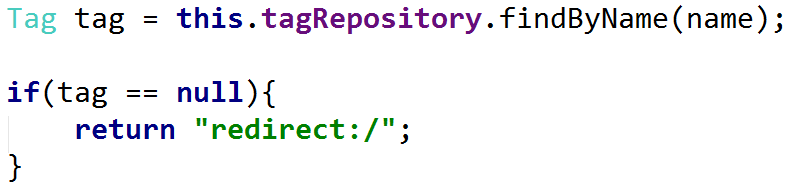
Inside, create a tag repository:



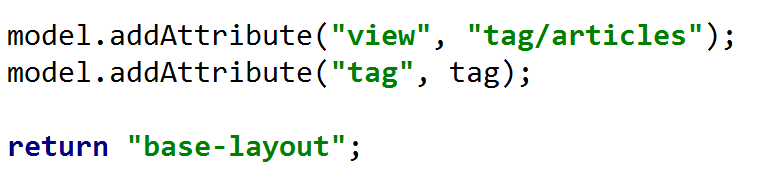
Now, create a new method that will **give us all of the articles** with a given tag:



Here we are using "@PathVariable" annotation again. However, this is the first time we are receiving a **string**. We need to **find the tag with given name**. If it **doesn't exist** we will **redirect the user** to the home page:



If it exists, we need to **send the tag to our view**:



Finally, we need to create the view. Create a new folder called "tag" in our templates folder. Create a new view, that will be called "articles". Use the following code:

|  |
| --- |
| <**main**>  <**div class="container body-content"**>  <**h1**>Search by Tag - <**th:block th:text="${tag.name}"**></**th:block**></**h1**>  <**div class="row"**>  <**th:block th:each="article : ${tag.articles}"**>  <**div class="col-md-6"**>  <**article**>  <**header**>  <**h2 th:text="${article.title}"**></**h2**>  </**header**>   <**p th:text="${article.summary}"**></**p**>   <**small class="author" th:text="${article.author.fullName}"**></**small**>   <**p**>  <**a class="btn btn-default btn-xs" th:each="tag:${article.tags}" th:href="@{/tag/{name}(name=${tag.name})}" th:text="${tag.name}"**></**a**>  </**p**>    <**footer**>  <**div class="pull-right"**>  <**a class="btn btn-default btn-xs"  th:href="@{/article/{id}(id=${article.id})}"**>Read more **&raquo;**</**a**>  </**div**>  </**footer**>  </**article**>  </**div**>  </**th:block**>  </**div**>  </**div**> </**main**> |

It is pretty much **the same code**, that we've used for the **article listing in category**.

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

We hope that **you've** **enjoyed working** with **Java, Spring, Hibernate** and **MySQL**. Farewell, friends! ☺