Blog: Java and Spring

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

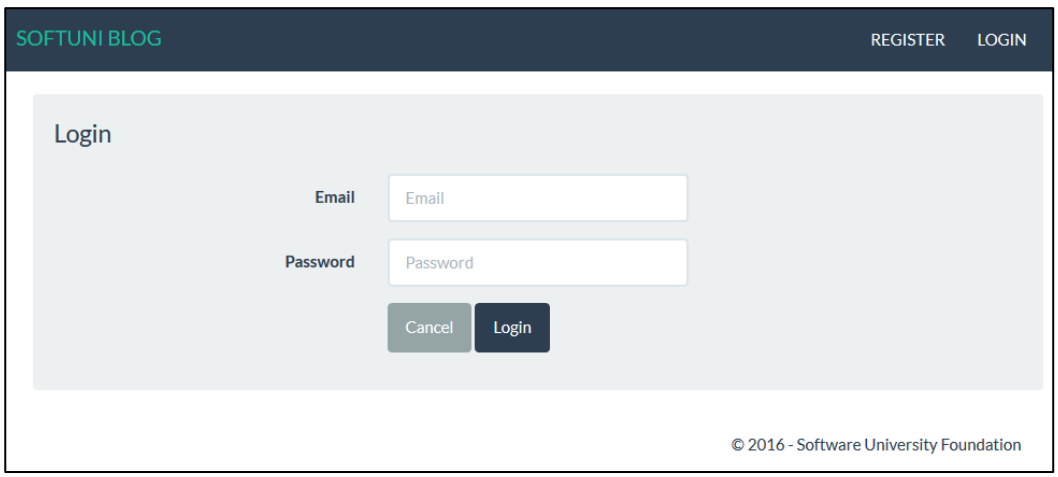
Chapters from III to V are for advanced users. There's a skeleton which you can use and start from chapter VI, after you set up the JDK in chapter II.

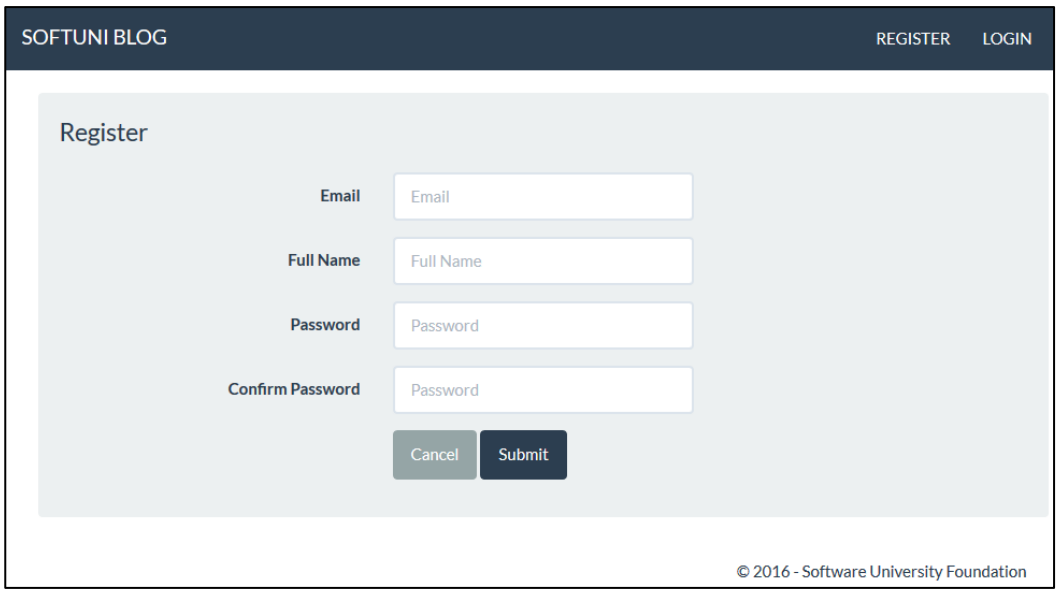
I. What Will You Create

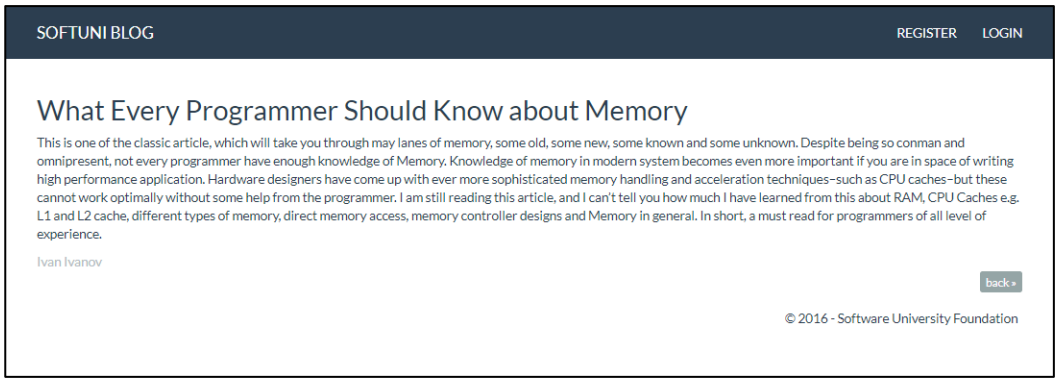
In the end of the whole tutorial, you should have blog which supports the following functionality:

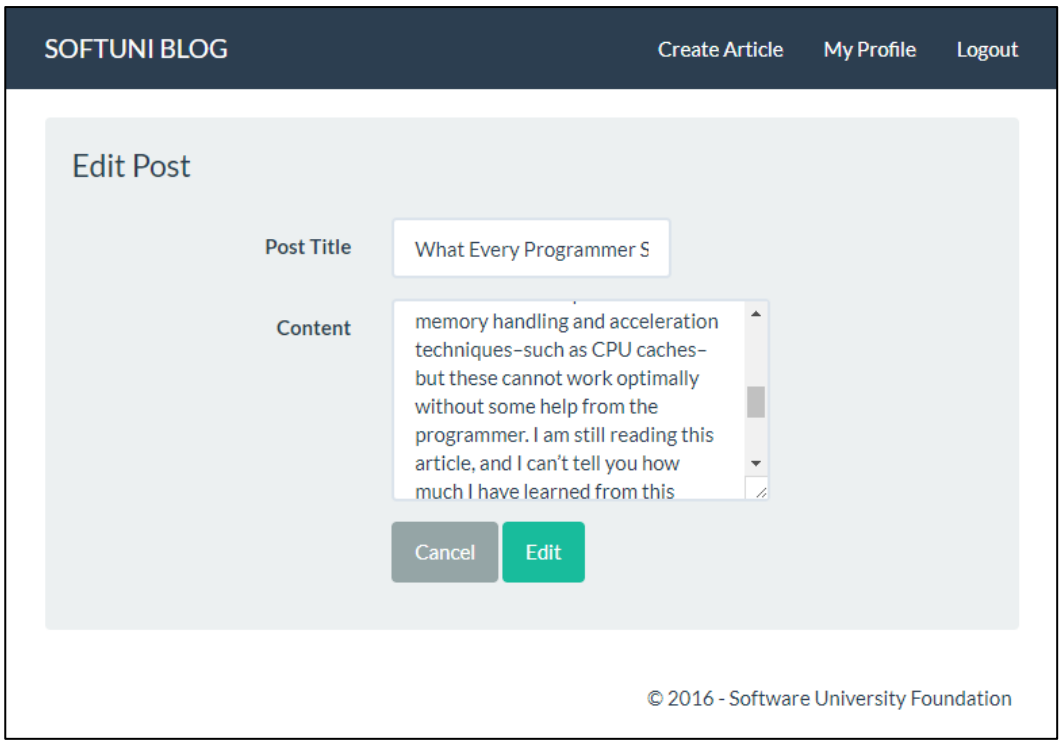
- **Register** and **Login** Users
- **Create** Articles
- **Edit** Articles
- **Delete** Articles
- **Responsive Design** using **Bootstrap**
- **More…**

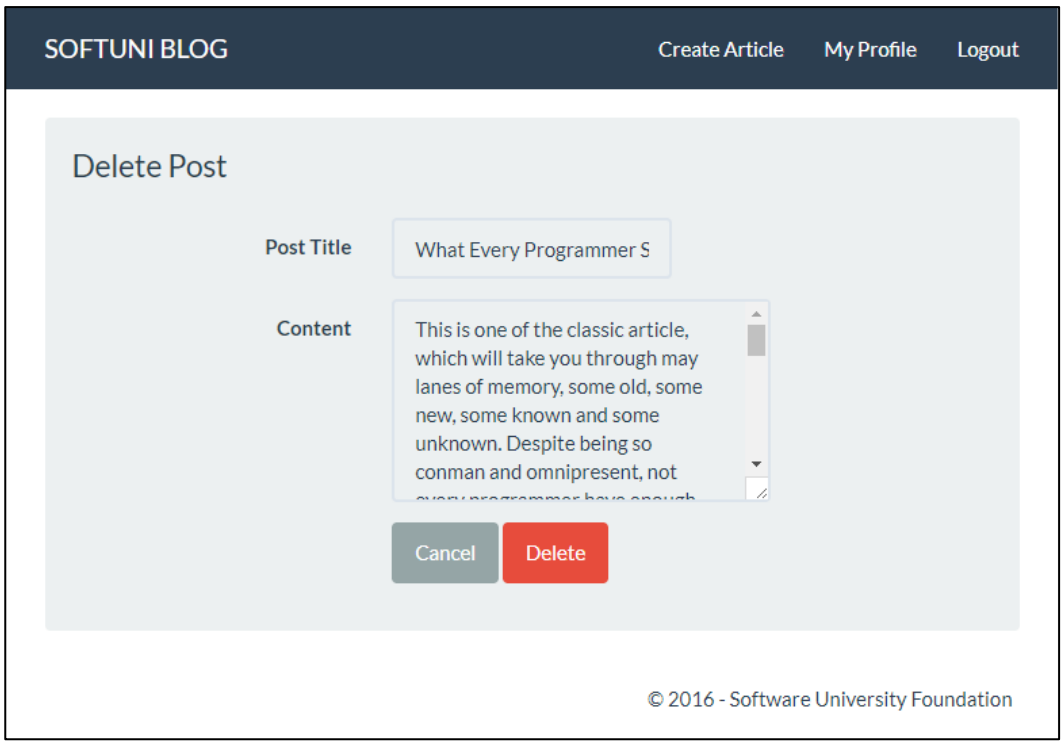
Pictures:











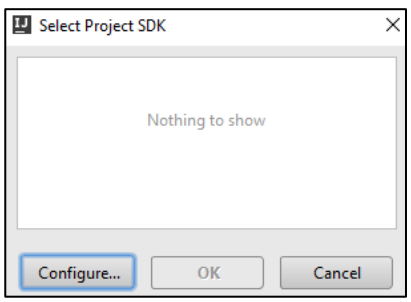
II. Set Up JDK and IntelliJ Idea Configuration

Before we start you need to download the Java Development Kit, also known as **JDK**. Download the "**Java SE Development Kit 8u(latest version number)**". After downloading it, install it **without changing the installation directory**. That will install it in the "**Program Files**" folder if you are on **Windows**.

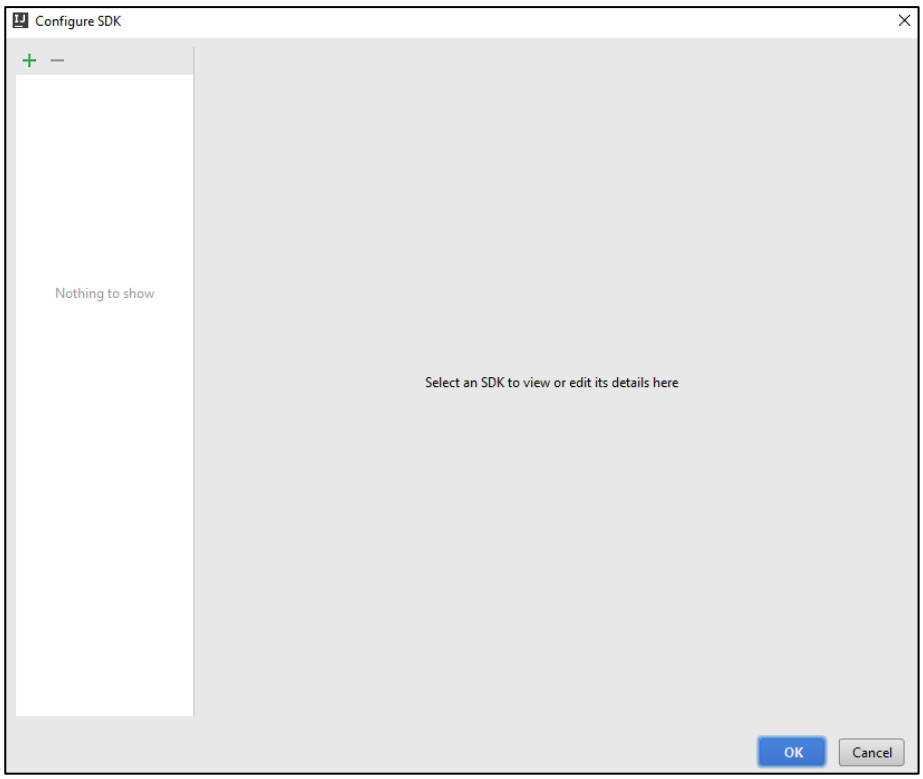
Using the Skeleton

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

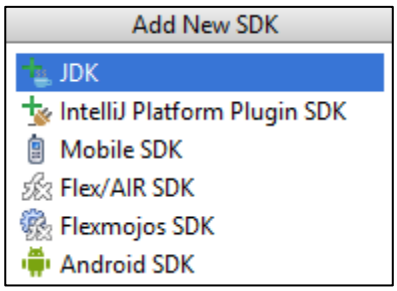
You should set-up the SDK. Click on "**Setup SDK**". You should see this screen:



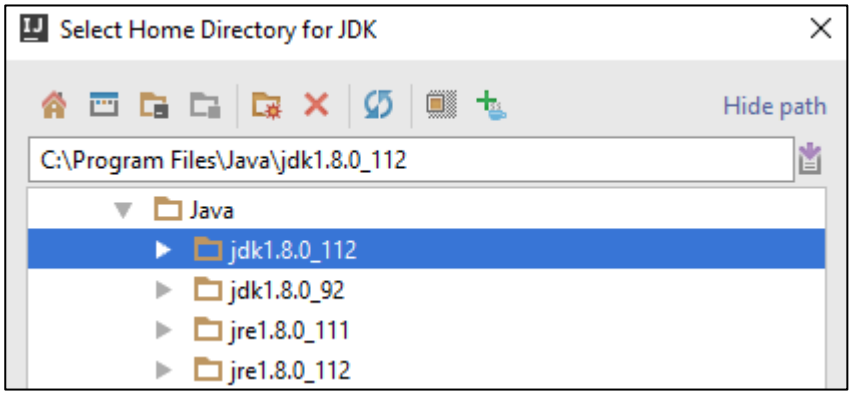
Click on "**Configure**" and see if you receive this screen:



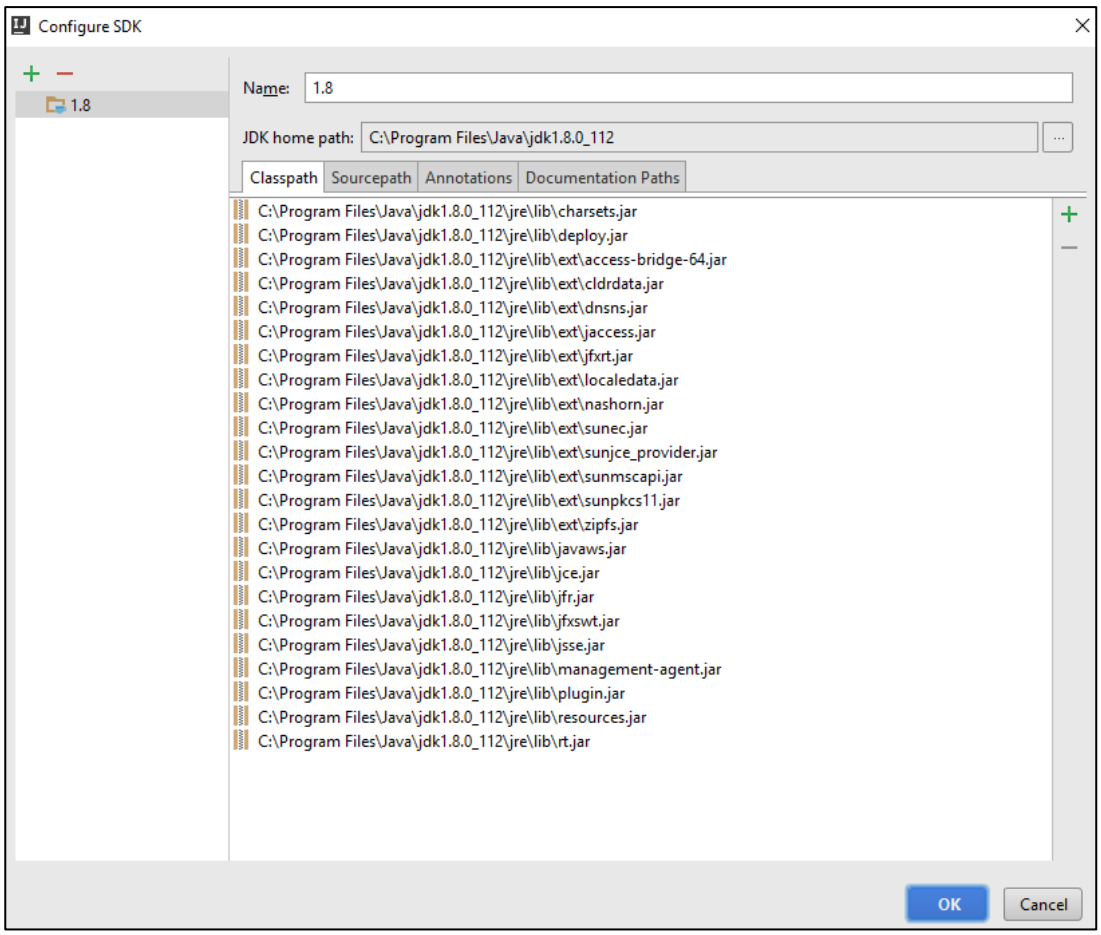
Click on the **green plus sign** in the top left corner of the window and choose **JDK**:



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:

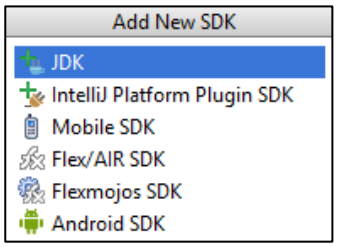


That is everything, your **JDK is now configured**.

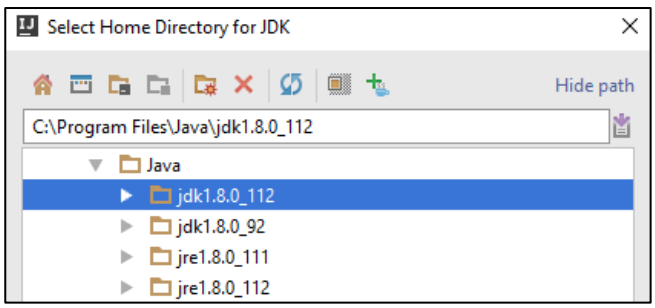
1. Creating New Project

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

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**:



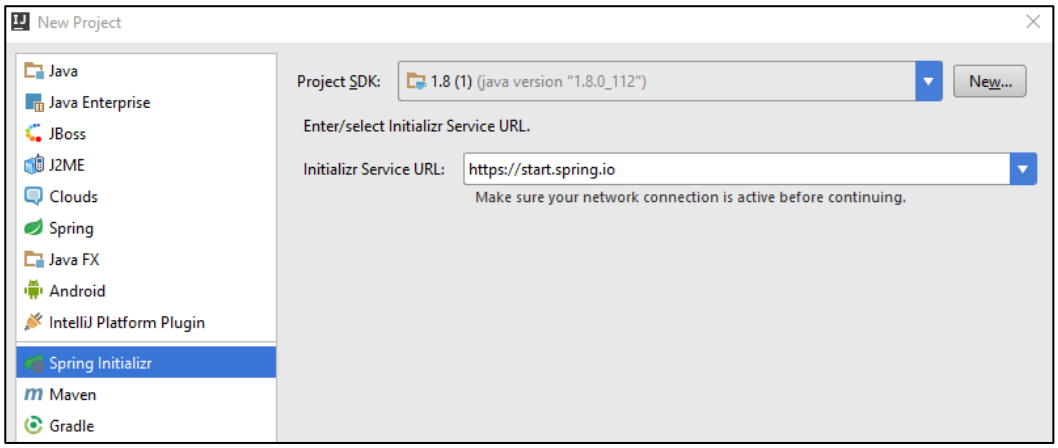
Click "**OK**" and you are **ready to create your project**.

III. Create Spring Project

1. Using Spring Initializr

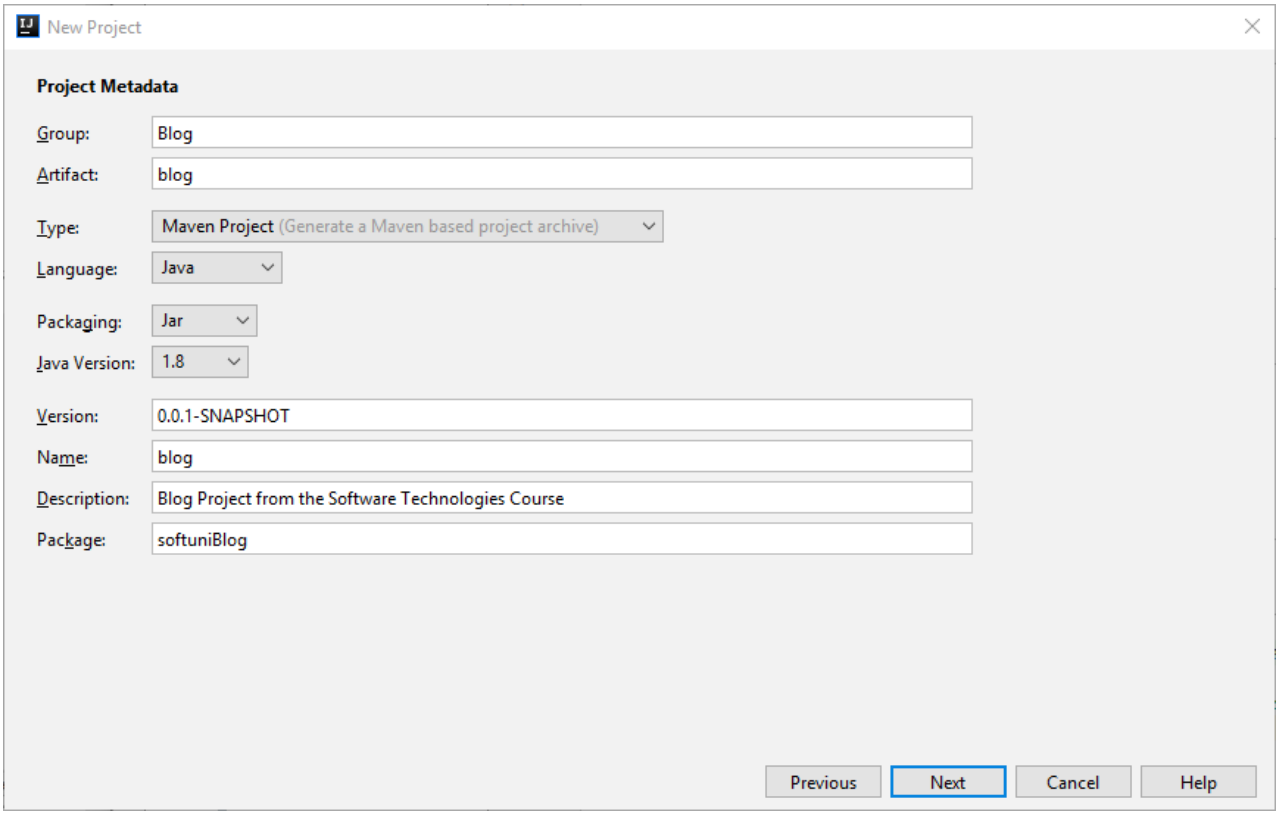
Setting Spring projects without any help is usually a time-consuming thing to do. That is because you need to search the internet for each module that you want to install. This is not always easy and thankfully there are tools that make our life easier. One of the tools is **Spring Initializr**. There is a [web version], 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**.

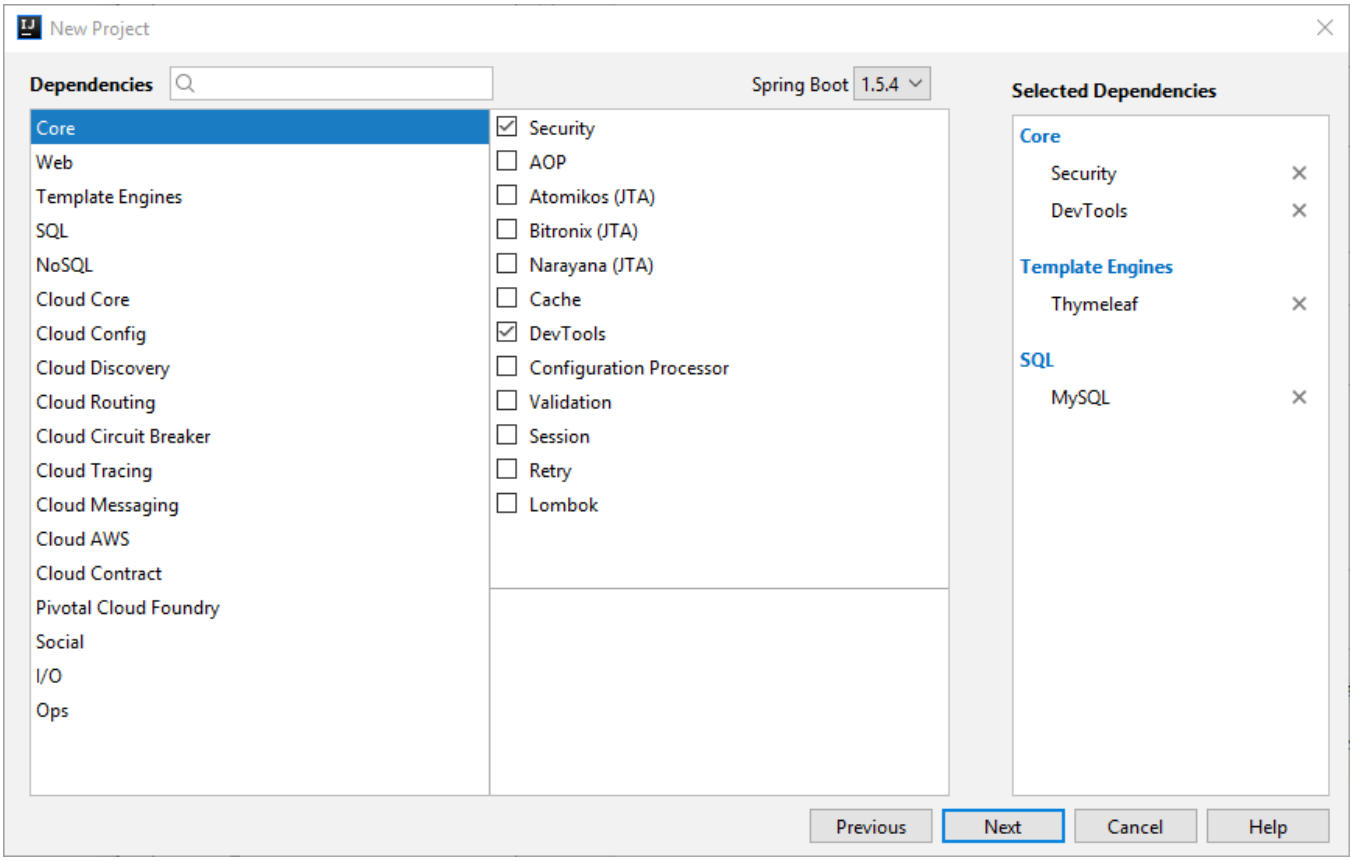
Click on "**Next**":



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:

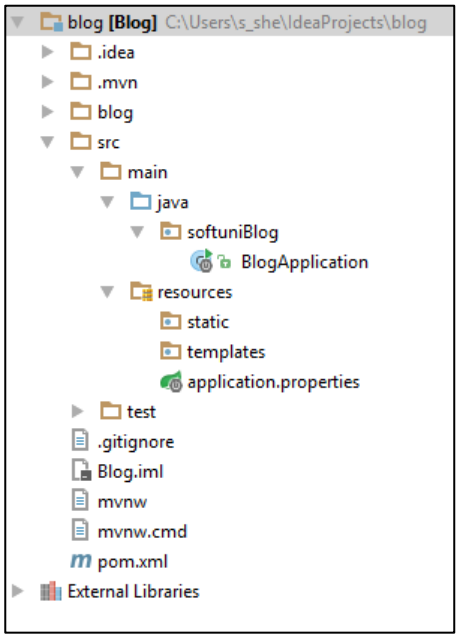
- From **Core** choose:
  - **Security**
  - **DevTools**
- From **Template Engines** choose:
  - **Thymeleaf**
- From SQL choose:
  - **MySQL**

You should have something like this:



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

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



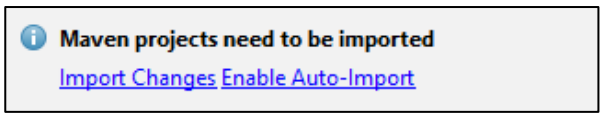
We will explain the project structure in the next chapter, but first we need to import something.

2. Import Additional Dependencies

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

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-security</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-thymeleaf</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-devtools</artifactId>
        <scope>runtime</scope>
    </dependency>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <scope>runtime</scope>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>
```

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>
```

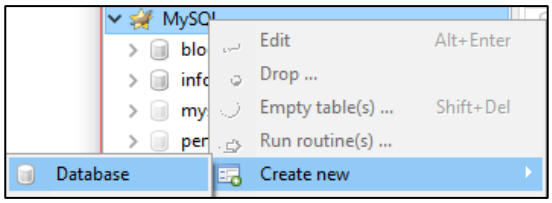
Insert this at the **bottom of the dependencies section**, and you have this:

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

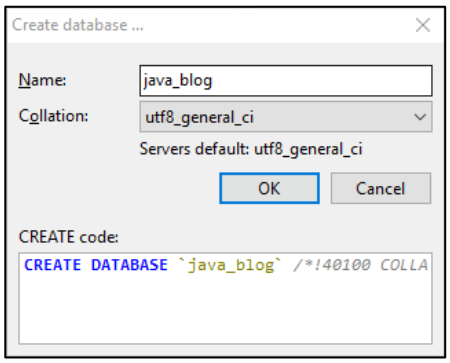
This will give us some **additional commands** that we are going to use in the **following chapters**.

3. Create the Database Connection

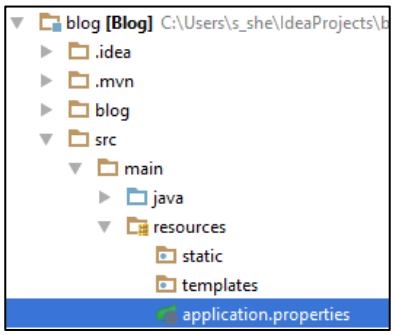
The last thing we are going to do in this chapter is create the DB connection. For database, we are going to use **MySQL**, the **same DB** we've used in the **PHP Blog**. That means that you will need to have **XAMPP** installed. Now you need to start the **MySQL module** in XAMPP and open HeidiSQL. 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/java_blog?createDatabaseIfNotExist=true&useSSL=false

# Username and password
spring.datasource.username = root
spring.datasource.password =

# Show or not log for each sql query
spring.jpa.show-sql = true

# Hibernate ddl auto (create, create-drop, update): with "update" the database
# schema will be automatically updated accordingly to java entities found in
# the project
# Using "create" will delete and recreate the tables every time the project is
started
spring.jpa.hibernate.ddl-auto = update
```

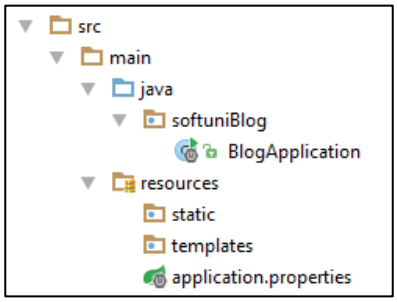
```
# Naming strategy
spring.jpa.hibernate.naming.strategy = org.hibernate.cfg.ImprovedNamingStrategy

# Allows Hibernate to generate SQL optimized for a particular DBMS
spring.jpa.properties.dialect = org.hibernate.dialect.MySQL5InnoDBDialect

#Turn off Thymeleaf cache
spring.thymeleaf.cache = false
```

Our connection is done. We will test it later.

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

IV. Reviewing the Project Structure

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



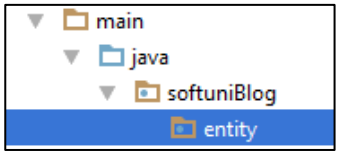
It contains main folder, which is then **separated into 2 different** folders. The first one is the "**java**" folder. This folder contains our **blog package**. **Inside** that **package,** we are going to **create** our **entities**, **controllers**, **configurations**, etc. The other folder is called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

V. Spring Security

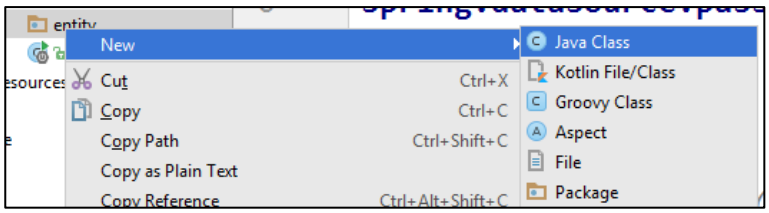
At the moment, you **cannot use your project**. Why? Because we've **imported the dependency for Spring**, that gives us the **authentication module**, but we haven't configured it, yet. To do that we will create **`User` entity** using Hibernate. Then we are going to tell **Spring Security** what to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

1. 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:

```
package softuniBlog.entity;

public class User {

}
```

Let's start with the first annotation:

```
import javax.persistence.Entity;

@Entity
public class User {
```

This annotation means that the **User** class will become entity that will get **saved into our database**. The next annotation is going to **define the table name** in our **database**:

```
import javax.persistence.Entity;
import javax.persistence.Table;

@Entity
@Table(name = "users")
public class User {
```

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

```
public class User {

    private Integer id;

    private String email;

    private String fullName;

    private String password;
```

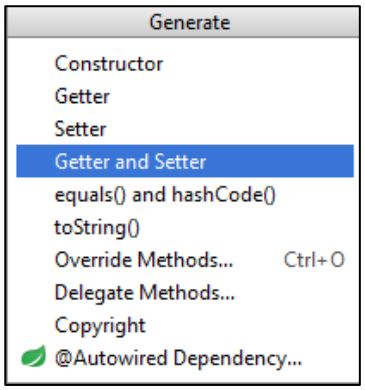
That is the information that we will keep in the database for our user. **ID**, which will be the **unique key**, **email**, **name** and **password**. The next thing that we are going to **create** is **constructor**, which should **help** us **with** the **user creation** later on:

```
public User(String email, String fullName, String password) {
    this.email = email;
    this.password = password;
    this.fullName = fullName;
}
```

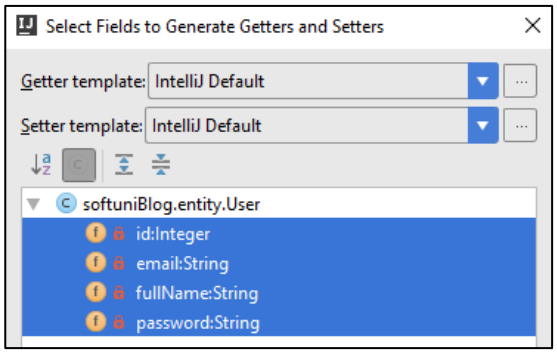
Spring Security will need **second constructor** in order to provide us with useful features. **It should be empty**:

```
public User() {    }
```

Now we need getters and setters. You should already be familiar with them. If you are curious why are we doing that, you can read more here. 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**:

```
public Integer getId() { return id; }

public void setId(Integer id) { this.id = id; }

public String getEmail() { return email; }

public void setEmail(String email) { this.email = email; }

public String getFullName() { return fullName; }

public void setFullName(String fullName) { this.fullName = fullName; }

public String getPassword() { return password; }

public void setPassword(String password) { this.password = password; }
```

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:

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
public Integer getId() { return id; }
```

The "**@Id**" annotation tells Hibernate 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:

```
@Column(name = "email", unique = true, nullable = false)
public String getEmail() { return email; }
```

Password:

```
@Column(name = "password", length = 60, nullable = false)
public String getPassword() { return password; }
```

Name:

```
@Column(name = "fullName", nullable = false)
public String getFullName() { return fullName; }
```

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

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

2. Creating the Role Entity

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

```
@Entity
@Table(name = "roles")
public class Role {

}
```

The next thing is the private fields:

```
private Integer id;

private String name;
```

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

```
public Integer getId() { return id; }

public void setId(Integer id) { this.id = id; }

public String getName() { return name; }

public void setName(String name) { this.name = name; }
```

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

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
public Integer getId() { return id; }
```

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

```
@Column(name = "name", nullable = false)
public String getName() { return name; }
```

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

3. Creating the Role-User Relation

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

```
private Set<User> users;
```

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

```
public Role() {
    this.users = new HashSet<>();
}
```

You can read more about the **HashSet** here.

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

```
public Set<User> getUsers() { return users; }

public void setUsers(Set<User> users) { this.users = users; }
```

And the annotation will be this:

```
@ManyToMany(mappedBy = "roles")
public Set<User> getUsers() { return users; }
```

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

We should jump to the **User** entity now.

4. Creating the User-Role Relation

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

```
private Set<Role> roles;
```

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

```
public Set<Role> getRoles() { return roles; }

public void setRoles(Set<Role> roles) { this.roles = roles; }
```

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

```
@ManyToMany(fetch = FetchType.EAGER)
@JoinTable(name = "users_roles")
public Set<Role> getRoles() { return roles; }
```

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. 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:

```
public User(String email, String fullName, String password) {
    this.email = email;
    this.password = password;
    this.fullName = fullName;
}
```

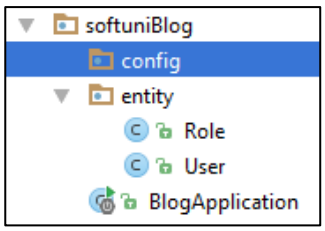
We are not **assigning default role** when we **create new user**. That's why we need to change the constructor like that:

```
public User(String email, String fullName, String password) {
    this.email = email;
    this.password = password;
    this.fullName = fullName;

    this.roles = new HashSet<>();
}
```

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

5. 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:

```
import org.springframework.security.core.userdetails.UserDetails;
import softuniBlog.entity.User;

public class BlogUserDetails extends User implements UserDetails {
}
```

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

```
@Override
public boolean isAccountNonExpired() { return true; }

@Override
public boolean isAccountNonLocked() { return true; }

@Override
public boolean isCredentialsNonExpired() { return true; }

@Override
public boolean isEnabled() { return true; }
```

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**:

```
private ArrayList<String> roles;
private User user;
```

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:

```
public BlogUserDetails(User user, ArrayList<String> roles) {
    super(user.getEmail(), user.getFullName(), user.getPassword());

    this.roles = roles;
    this.user = user;
}
```

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). For now, you can imagine that it **assigns** the **user email**, **name** and **password** to our class, using the **constructor** of our base class.

Now we need to override one more method:

```
@Override
public Collection<? extends GrantedAuthority> getAuthorities() {
    String userRoles =  StringUtils.collectionToCommaDelimitedString(this.roles);
    return AuthorityUtils.commaSeparatedStringToAuthorityList(userRoles);
}
```

This will get our **roles** (that we currently keep as strings) and **join them** into one string (we use the **`StringUtils`** class from **`org.springframework.util`**). Then it will return collection of authorities. 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**:

```
public User getUser() {
    return this.user;
}
```

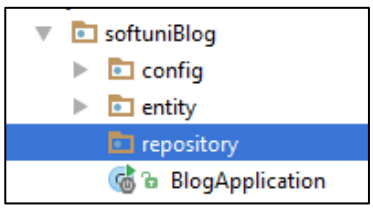
And **override** the last method, which we need to implement:

```
@Override
public String getUsername() {
    return this.user.getEmail();
}
```

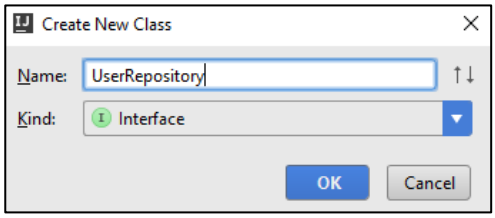
That is all, now we need to **find a way** to get our **users** and **roles** from the **database**.

6. Creating User Repository

Now, we are not exactly finding a way to get the users. There is a way called "**Repositories**". You can imagine that the repository 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. The interface is a special type, which **can't contain functional methods**. It can **only declare them**. You should have this:

```
public interface UserRepository {
}
```

We should quickly change it to:

```
import org.springframework.data.jpa.repository.JpaRepository;
import softuniBlog.entity.User;

public interface UserRepository extends JpaRepository<User, Integer> {
}
```

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**:

```
User findByEmail(String email);
```

As you can see, this **method is different**. It doesn't have body. Using magic (and 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 **query** to our **database**. Then it will take the **second part** which is "**Email**" in this case and it will understand that we want to get **user** by a **given email** address. The **generated query** will look like this "**SELECT id, email, full_name, password FROM users WHERE email={parameter}**". Anyway, let's move on.

7. Creating Role Repository

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

```
import org.springframework.data.jpa.repository.JpaRepository;
import softuniBlog.entity.Role;

public interface RoleRepository extends JpaRepository<Role, Integer> {

}
```

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

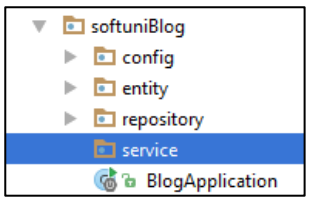
```
Role findByName(String name);
```

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

We are ready with our repositories for now.

8. Creating User Service

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



Create a new class called **BlogUserDetailsService**:

```
public class BlogUserDetailsService {

}
```

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

```
@Service("blogUserDetailsService")
public class BlogUserDetailsService {

}
```

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

```
@Service("blogUserDetailsService")
public class BlogUserDetailsService implements UserDetailsService {

}
```

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

```
private final UserRepository userRepository;

public BlogUserDetailService(UserRepository userRepository) {
    this.userRepository = userRepository;
}
```

This private field has the "**`final`**" keyword, which means that we will **not be able to change it** after **initialization**.

```
@Override
public UserDetails loadUserByUsername(String email) throws UsernameNotFoundException {

}
```

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**:

```
User user = userRepository.findByEmail(email);

if (user == null) {
    throw new UsernameNotFoundException("Invalid User");
} else {

}
```

The case, where the user exist is more interesting:

```
Set<GrantedAuthority> grantedAuthorities = user.getRoles()
        .stream()
        .map(role -> new SimpleGrantedAuthority(role.getName()))
        .collect(Collectors.toSet());

return new org
        .springframework
        .security
        .core
        .userdetails
        .User(user.getEmail(), user.getPassword(), grantedAuthorities);
```

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

9. Creating Web Security Configurer Adapter

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

```
public class WebSecurityConfig {

}
```

Now it will get really messy, really quick:

```
@Configuration
@EnableGlobalMethodSecurity(prePostEnabled = true)
@EnableWebSecurity
@Order(SecurityProperties.ACCESS_OVERRIDE_ORDER)
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {

}
```

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

- @Configuration
- @EnableGlobalMethodSecurity
- @EnableWebSecurity
- @Order

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**:

```
private UserDetailsService userDetailsService;
```

We will need to create the following annotation for it:

```
@Autowired
private UserDetailsService userDetailsService;
```

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**:

```
@Autowired
public void configAuthentication(AuthenticationManagerBuilder auth) throws Exception{
    auth.userDetailsService(this.userDetailsService).passwordEncoder(new BCryptPasswordEncoder());
}
```

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**:

```
@Override
protected void configure(HttpSecurity http) throws Exception {

}
```

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

```
http.authorizeRequests()
        .anyRequest().permitAll()
        .and()
        .formLogin().loginPage("/login")
        .usernameParameter("email").passwordParameter("password")
        .and()
        .logout().logoutSuccessUrl("/login?logout")
        .and()
        .exceptionHandling().accessDeniedPage("/error/403")
        .and()
        .csrf();
```

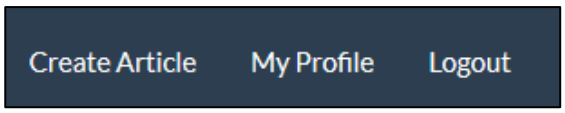
This code tells the authentication module, that **every page** can be **accessed** by **every user**. Then it tells us that the **login request** should be expected at the "**/login**" **route**. The parameter for login will be "**email**" and the parameter for password will be "**password**". The **logout** 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…

10. Creating Base Layout

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

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

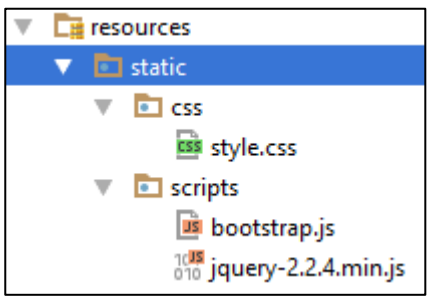
The other two parts as you can see for **logged users**. Using **Thymeleaf**, we need to check if whoever is opening our pages is logged in or not. **Take a look** at this code (**don't write** it anywhere):

```
<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>
```

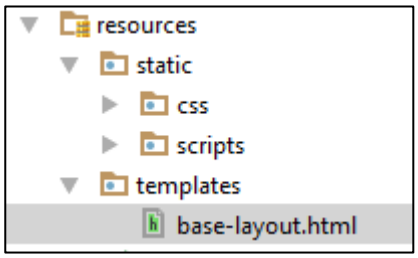
For every link in our navigation menu we are using "**`sec:authorize`**". This is coming from **Thymeleaf Security** and gives us the ability to **check if someone** is **logged in**, **has a specific role** or is just a **guest**.

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

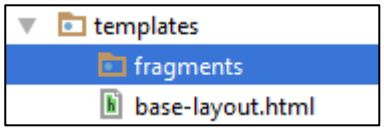
This is also used on **every single of our pages**. We will also **need scripts**, to **use bootstrap**. The final section is called "**main**" and it is unique for every page. It contains the content for any given page. However, three of four **sections can be reused**. First, we need to import our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



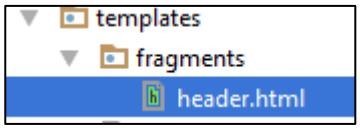
Now, we will create our base layout. Create a **new HTML file** in the **`templates`** package called "**`base-layout`**" and leave it there for now:



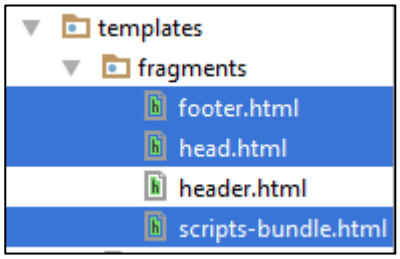
Now, create a **new directory** in the **`templates`** package called "**`fragments`**". It will contain the fragments(sections), we've talked earlier about:



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:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>

</body>
</html>
```

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:

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

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>
```

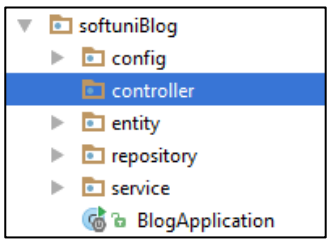
Its job is to import the 2 **JavaScript** files we are going to use. As you can see, all of our fragments have a specific role in our design, but let's combine them together in our "**base-layout.html**":

```
<!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**.

11. Creating User Controller

Create a new package called "**controller**":



Now create new class called **UserController**:

```
public class 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:

```
@Controller
public class UserController {
```

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:

```
@Autowired
private RoleRepository roleRepository;
@Autowired
private UserRepository userRepository;
```

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

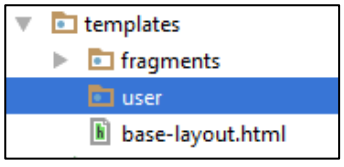
First, we need to be able to create users. Create the following method:

```
@GetMapping("/register")
public String register(Model model) {

}
```

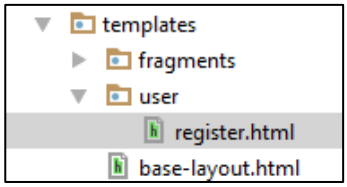
We are using the "**`@GetMapping`**" annotation. This annotation defines that the type of request we are going to **process in our method** is "**GET**". That means that if **someone sends data** (i.e. user data), this method won't be called. **This method** will **only be called** if someone **tries 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**:

```
@GetMapping("/register")
public String register(Model model) {
    model.addAttribute("view", "user/register");

    return "base-layout";
}
```

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**:

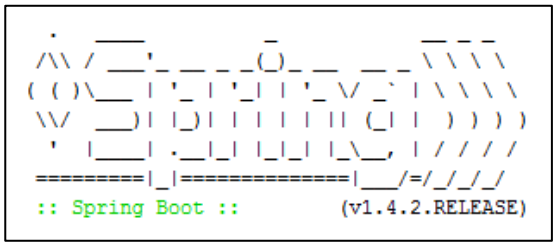
```
<main>
    <div class="container body-content span=8 offset=2">
        <div class="well">
            <form class="form-horizontal" th:action="@{/register}"
method="post">
                <fieldset>
                    <legend>Register</legend>
                    <div class="form-group">
                        <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**.

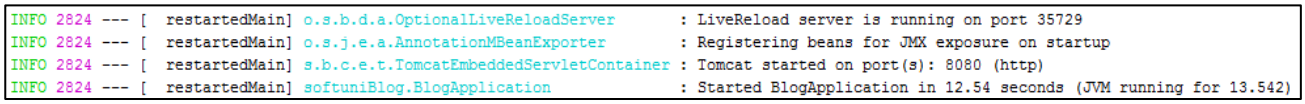
12. Starting the Project for the First Time

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

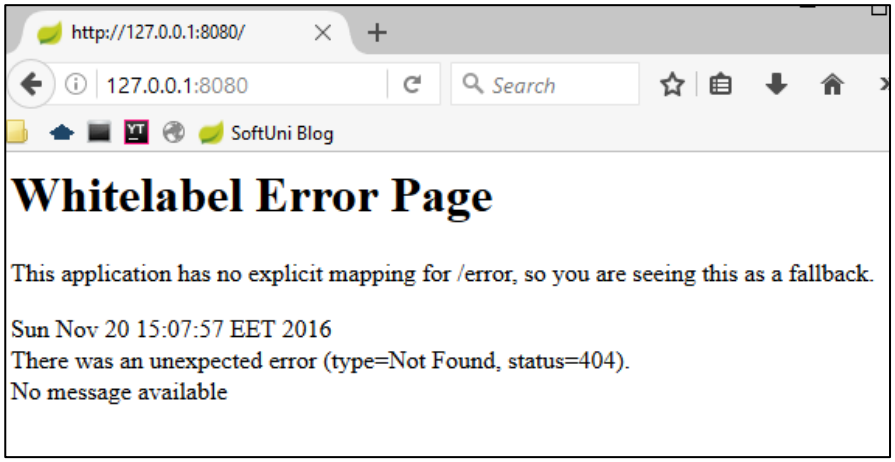
Click on the green arrow () and soon you should see something like this:



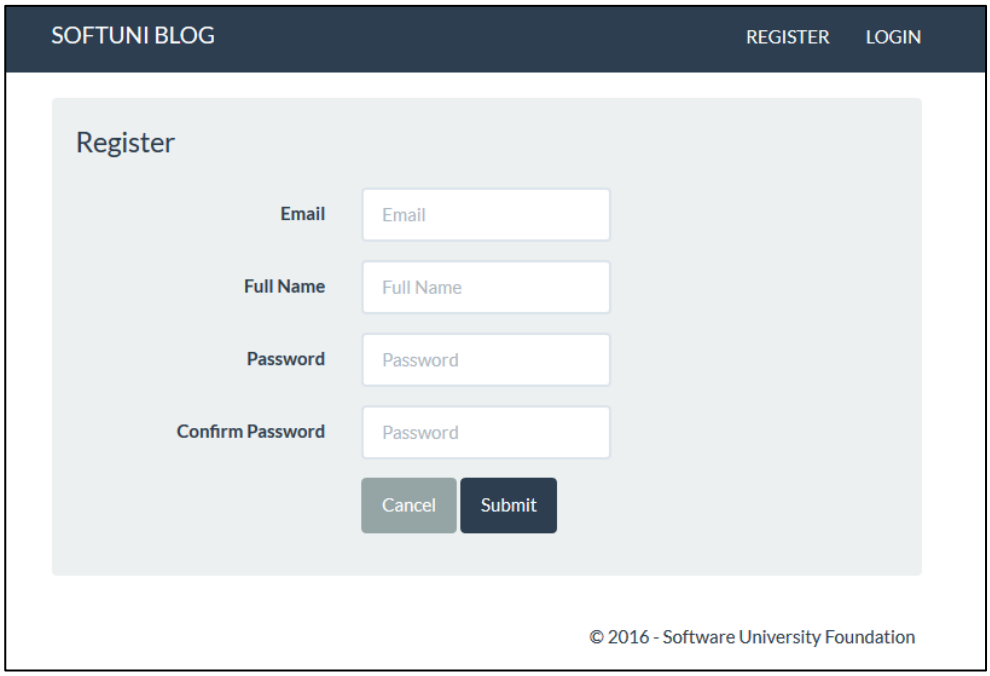
Wait until you see this:



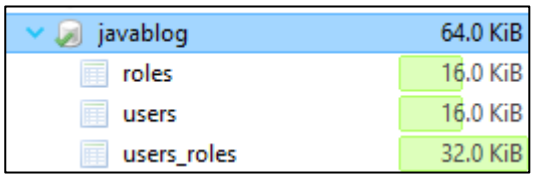
The last row means that the **server is running**. Open http://127.0.0.1:8080/ and see what you get. You should **receive this error message**:



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:

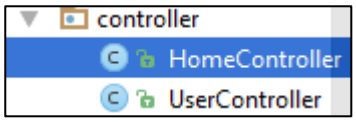


Stop the blog using the  icon.

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

13. Creating Home Contoller

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



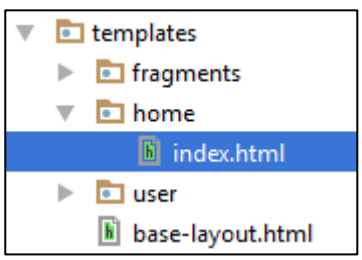
This controller will **list all of our articles** later on, but for now it will just **return an empty page**. Create a new function called "**`index`**":

```
@Controller
public class HomeController {

    @GetMapping
    public String index(Model model) {

    }
}
```

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

```
@GetMapping("/")
public String index(Model model) {
    model.addAttribute("view", "home/index");
    return "base-layout";
}
```

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



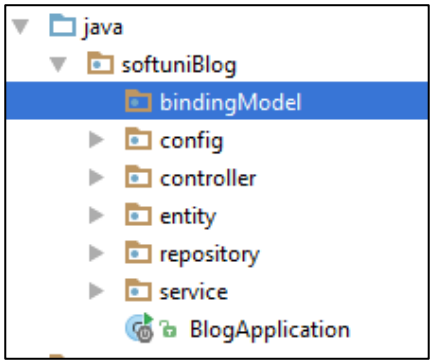
Delete everything from the file and **leave it empty**. When we create our articles, we will edit it.

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

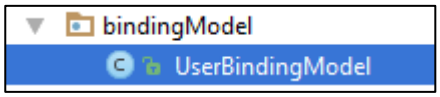
When you click on the **register button**, the hyperlink should lead you to the **register form**. That is everything for our **`HomeController`** at this point in time.

14. Finishing the User Register

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



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:

```
import javax.validation.constraints.NotNull;

public class UserBindingModel {
    @NotNull
    private String email;

    @NotNull
    private String fullName;

    @NotNull
    private String password;

    @NotNull
    private String confirmPassword;
}
```

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:

```
public String getEmail() { return email; }

public void setEmail(String email) { this.email = email; }

public String getFullName() { return fullName; }

public void setFullName(String fullName) { this.fullName = fullName; }

public String getPassword() { return password; }

public void setPassword(String password) { this.password = password; }

public String getConfirmPassword() { return confirmPassword; }

public void setConfirmPassword(String confirmPassword) { this.confirmPassword = confirmPassword; }
```

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

```
public String registerProcess(UserBindingModel userBindingModel){

}
```

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**:

```
@PostMapping("/register")
public String registerProcess(UserBindingModel userBindingModel){
```

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**:

```
if(!userBindingModel.getPassword().equals(userBindingModel.getConfirmPassword())){
    return "redirect:/register";
}
```

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:

```
BCryptPasswordEncoder bCryptPasswordEncoder = new BCryptPasswordEncoder();

User user = new User(
        userBindingModel.getEmail(),
        userBindingModel.getFullName(),
        bCryptPasswordEncoder.encode(userBindingModel.getPassword())
);
```

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:

```
public void addRole(Role role) {
    this.roles.add(role);
}
```

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

```
Role userRole = this.roleRepository.findByName("ROLE_USER");
```

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

```
user.addRole(userRole);
```

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

```
this.userRepository.saveAndFlush(user);

return "redirect:/login";
```

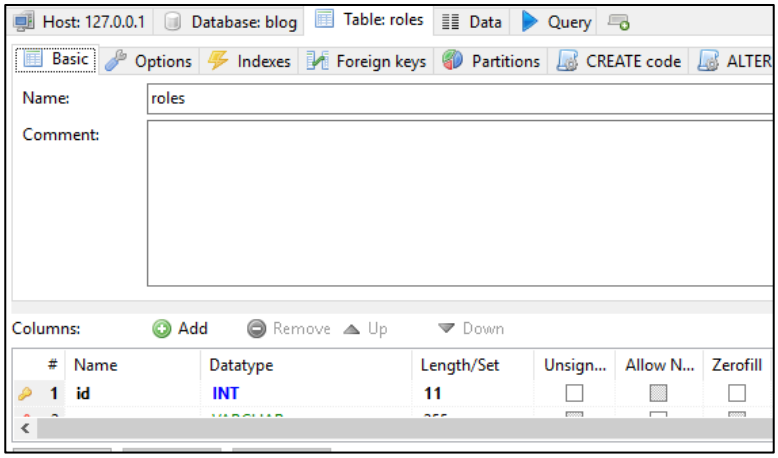
Before we test if it works, we want to change something. Maybe you have noticed that **every time** you start the application it **drops the old database** and **creates new one**. **We don't want that**. Find your **`application.properties`** file. Edit the following line:

```
spring.jpa.hibernate.ddl-auto = create
```

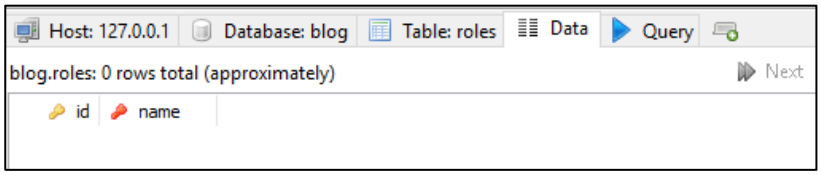
To

```
spring.jpa.hibernate.ddl-auto = update
```

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:



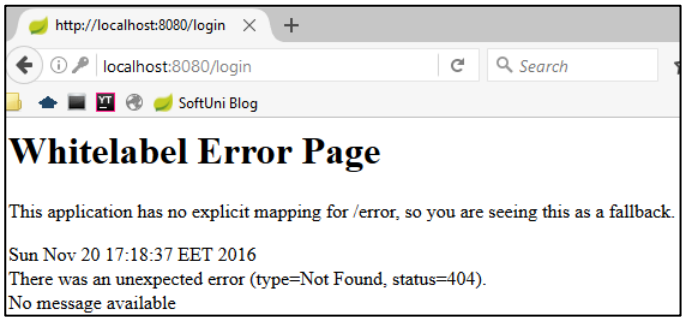
In the navigation, you will see "**Data**" tab. Open it and you should see this:



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:

id	name
1	ROLE_USER

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**:

id	email	full_name	password
1	ivan@softuni.bg	Ivan Ivanov	$2a$10$7VtFLFd9ow3d...

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

15. Implement User Login

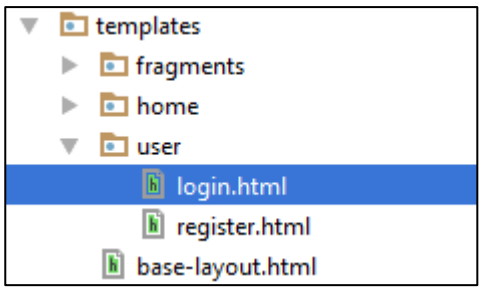
To create a login, we need **only 2 things**. A **method** and a **view**. **Spring Security will take care of the rest**. Let's start with the method in our **`UserController`**:

```
@GetMapping("/login")
public String login(Model model) {

}
```

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

```
@GetMapping("/login")
public String login(Model model) {
    model.addAttribute("view", "user/login");

    return "base-layout";
}
```

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.

16. Implement User Logout

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

```
@RequestMapping(value = "/logout", method = RequestMethod.GET)
public String logoutPage(HttpServletRequest request, HttpServletResponse response) {

}
```

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:

```
Authentication auth = SecurityContextHolder.getContext().getAuthentication();

if (auth != null) {
    new SecurityContextLogoutHandler().logout(request, response, auth);
}

return "redirect:/login?logout";
```

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

17. Create User Profile Page

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

```
@GetMapping("/profile")
public String profilePage(Model model) {

}
```

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**:

```
@GetMapping("/profile")
@PreAuthorize("isAuthenticated()")
public String profilePage(Model model) {

}
```

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:

```
UserDetails principal = (UserDetails) SecurityContextHolder.getContext()
                                                           .getAuthentication()
                                                           .getPrincipal();
```

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:

```
User user = this.userRepository.findByEmail(principal.getUsername());
```

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

```
model.addAttribute("user", user);
```

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

```
model.addAttribute("view", "user/profile");

return "base-layout";
```

Overall the code should look like this:

```
@GetMapping("/profile")
@PreAuthorize("isAuthenticated()")
public String profilePage(Model model) {

    UserDetails principal = (UserDetails) SecurityContextHolder.getContext()
                                                               .getAuthentication()
                                                               .getPrincipal();

    User user = this.userRepository.findByEmail(principal.getUsername());
    model.addAttribute("user", user);
    model.addAttribute("view", "user/profile");
    return "base/layout";
}
```

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**! ☺

VI. Create Articles

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

0. Start MySQL

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

If you are still reading:

Download the project skeleton, extract it in a shortest path you can make, e.g. in **c:\project**.

Before we start using our blog, we need to **create** a database. We will use MySQL, which you are given in the skeleton. To start using MySQL, just **double-click mysql_start.bat** from the root directory (e.g. **c:\project**). You will see a window like this one:

```
Diese Eingabeforderung nicht waehrend des Running beenden
Please dont close Window while MySQL is running
MySQL is trying to start
Please wait  ...
MySQL is starting with mysql\bin\my.ini (console)
2016-11-01  7:51:04 1556 [Note] mysql\bin\mysqld (mysqld 10.1.13-MariaDB) starting as process 1372 ...
2016-11-01  7:51:04 1556 [Note] InnoDB: Using mutexes to ref count buffer pool pages
2016-11-01  7:51:04 1556 [Note] InnoDB: The InnoDB memory heap is disabled
2016-11-01  7:51:04 1556 [Note] InnoDB: Mutexes and rw_locks use Windows interlocked functions
2016-11-01  7:51:04 1556 [Note] InnoDB: Memory barrier is not used
2016-11-01  7:51:04 1556 [Note] InnoDB: Compressed tables use zlib 1.2.3
2016-11-01  7:51:04 1556 [Note] InnoDB: Using generic crc32 instructions
2016-11-01  7:51:04 1556 [Note] InnoDB: Initializing buffer pool, size = 128.0M
2016-11-01  7:51:04 1556 [Note] InnoDB: Completed initialization of buffer pool
2016-11-01  7:51:04 1556 [Note] InnoDB: Highest supported file format is Barracuda.
2016-11-01  7:51:04 1556 [Note] InnoDB: The log sequence numbers 2607638 and 2607638 in ibdata files do nc
g sequence number 2624064 in the ib_logfiles!
2016-11-01  7:51:04 1556 [Note] InnoDB: Database was not shutdown normally!
2016-11-01  7:51:04 1556 [Note] InnoDB: Starting crash recovery.
2016-11-01  7:51:04 1556 [Note] InnoDB: Reading tablespace information from the .ibd files...
2016-11-01  7:51:04 1556 [Note] InnoDB: Restoring possible half-written data pages
2016-11-01  7:51:04 1556 [Note] InnoDB: from the doublewrite buffer...
2016-11-01  7:51:05 1556 [Note] InnoDB: 128 rollback segment(s) are active.
2016-11-01  7:51:05 1556 [Note] InnoDB: Waiting for purge to start
2016-11-01  7:51:05 1556 [Note] InnoDB:  Percona XtraDB (http://www.percona.com) 5.6.28-76.1 started; log
r 2624064
2016-11-01  7:51:05 13432 [Note] InnoDB: Dumping buffer pool(s) not yet started
2016-11-01  7:51:05 1556 [Note] Plugin 'FEEDBACK' is disabled.
2016-11-01  7:51:05 1556 [Note] Server socket created on IP: '::'.
2016-11-01  7:51:05 1556 [Note] mysql\bin\mysqld: ready for connections.
Version: '10.1.13-MariaDB'  socket: ''  port: 3306  mariadb.org binary distribution
```

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

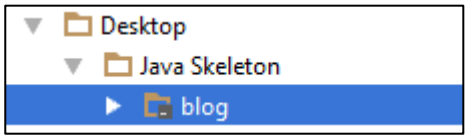
1. Open the Project

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

If you are still reading:

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

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.

2. Create the User Role

Using **HeidiSQL** import new **`Role`** into the **database** with **name** "**ROLE_USER**". If you don't know how to do it, refer to chapter V part 14.

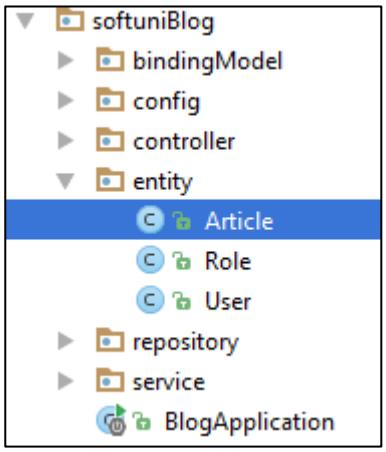
3. Start the project

You can find how to **start the project** in chapter V part 12.

4. Article Entity

It's time to create our first **`entity`**. We are using Hibernate for **ORM**. That means we are going to define our entities with annotations. 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:

```
package softuniBlog.entity;

public class Article {
}
```

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

```
import javax.persistence.Entity;

@Entity
public class Article {
```

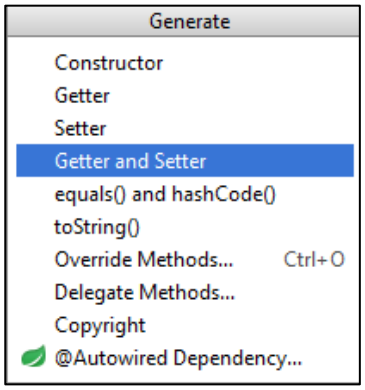
Now that our class is an entity we need to give our database **proper table name**:

```
@Entity
@Table(name = "articles")
public class Article {
```

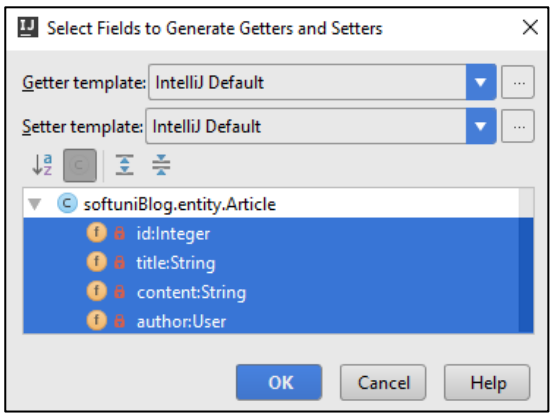
The next important thing is the **table columns**. We need columns for **`id`**, **`title`**, **`content`** and **`author`**. Create the following private fields:

```
private Integer id;

private String title;

private String content;

private User author;
```

Before we explain each column, let's create getters and setters for our fields. You should already be familiar with them. If you are curious why are we doing that, you can read more here. 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**:

```
public Integer getId() { return id; }

public void setId(Integer id) { this.id = id; }

public String getTitle() { return title; }

public void setTitle(String title) { this.title = title; }

public String getContent() { return content; }

public void setContent(String content) { this.content = content; }

public User getAuthor() { return author; }

public void setAuthor(User author) { this.author = author; }
```

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**:

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
public Integer getId() { return id; }
```

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**:

```
@Column(nullable = false)
public String getTitle() { return 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:

```
@Column(columnDefinition = "text", nullable = false)
public String getContent() { return content; }
```

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**:

```
public Article(String title, String content, User author) {
    this.title = title;
    this.content = content;
    this.author = author;
}
```

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

```
public Article() {    }
```

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

5. Article-User Relation

Remember that we've left the **author** field in the **Article** entity for later? Find the getter. Before we create the annotation, let's talk about the relation between our **Article** and the **User** entity. Our relation will be of type OneToMany. 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:

```
@ManyToOne()
@JoinColumn(nullable = false, name = "authorId")
public User getAuthor() { return author; }
```

The first one is the "**ManyToOne**" annotation. Many to one relationship represents OneToMany relationship from the side of the "many". Because we are working with the **Article** entity, we are telling **Hibernate** that **many of our articles** will correspond **to one user**. The other annotation is "**JoinColumn**", which tells **Hibernate** that it should **create a column** called "**authorId**" that will keep our relation and can't be **null**. This is everything from this side of the relation.

6. User-Article Relation

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

```
private Set<Article> articles;
```

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:

```
public User(String email, String fullName, String password) {
    this.email = email;
    this.password = password;
    this.fullName = fullName;

    this.roles = new HashSet<>();
}
```

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

```
this.roles = new HashSet<>();
this.articles = new HashSet<>();
```

Here we are telling **Java** that our specific type of **Set** should be the HashSet. 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:

```
public Set<Article> getArticles() {
    return articles;
}

public void setArticles(Set<Article> articles) {
    this.articles = articles;
}
```

Let's add the annotation for our relation:

```
@OneToMany(mappedBy = "author")
public Set<Article> getArticles() { return articles; }
```

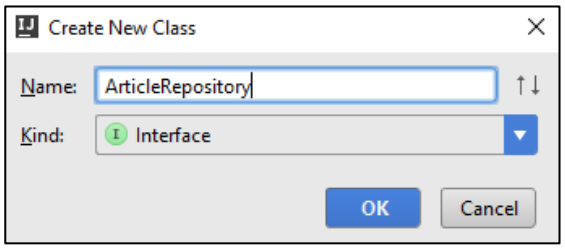
It is pretty simple. It means that **Hibernate** should go to our **Article** entity and find the "**author**" field that we've created earlier. Then it will get the **properties** of the relation from there and use them as a base when creating the foreign key **constraints** in the database.

Our relation is ready now.

7. Create Article Repository

If you want to read more about **repositories**, you can do it in chapter V part 6. 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:

```
package softuniBlog.repository;

public interface ArticleRepository {
}
```

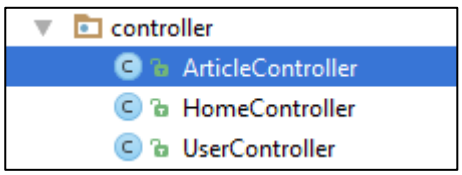
The only thing we want to do is tell **Spring** that our **repository** will be of **type JpaRepository<Entity, Primary Key>**:

```
public interface ArticleRepository extends JpaRepository<Article, Integer> {
}
```

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

8. Create Article Controller

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



Add the following annotation:

```
import org.springframework.stereotype.Controller;

@Controller
public class ArticleController {
}
```

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**:

```
private ArticleRepository articleRepository;

private UserRepository userRepository;
```

Add the following annotation to both of them:

```
@Autowired
private ArticleRepository articleRepository;
@Autowired
private UserRepository userRepository;
```

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. Using the "**@Autowired**" annotation, we tell **Spring** that **it should initialize** and **configure** our **repositories automatically**. We are ready to start creating articles.

9. Creating Articles Part I

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

```
public String create(Model model) {

}
```

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:

```
@GetMapping("/article/create")
@PreAuthorize("isAuthenticated()")
public String create(Model model) {
```

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. In our method write the following code:

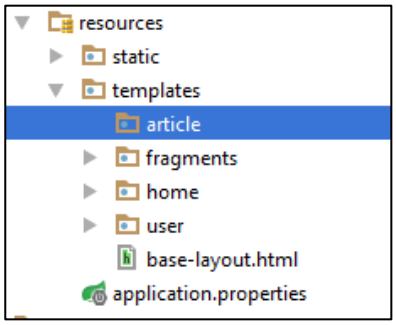
```
model.addAttribute("view", "article/create");

return "base-layout";
```

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

10. Creating the View

In order to use **loops** and **logical statements** in our **HTML** we will once again use **templating engine**. You should be **familiar** with **Twig** and **Handlebars** by now. Today you are going to use **Thymeleaf**. Thymeleaf 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:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>

</body>
</html>
```

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

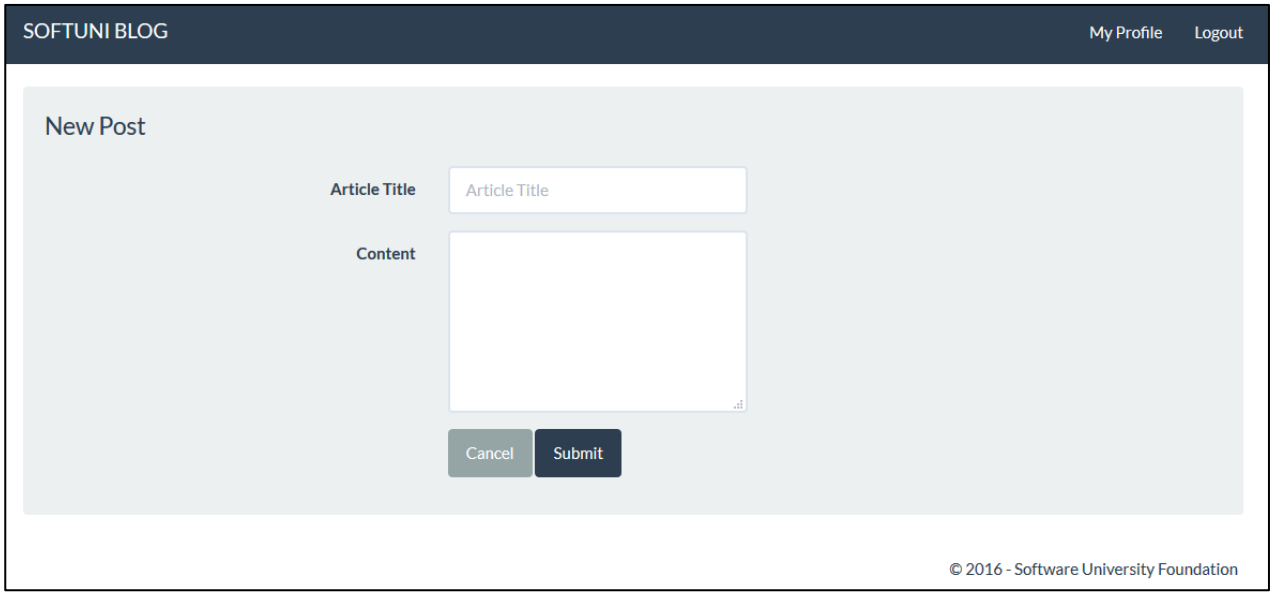
```
<main>
    <div class="container body-content span=8 offset=2">
        <div class="well">
            <form class="form-horizontal" th:action="@{/article/create}"
method="POST">
                <fieldset>
                    <legend>New Post</legend>

                    <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**.

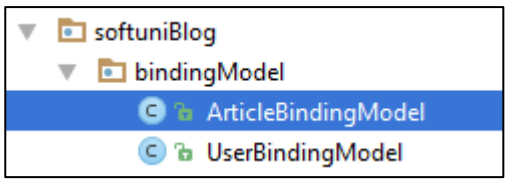
The next and last Thymeleaf attribute is "**`th:href`**". It will also **redirect** us **to** a **given route**. We can now **test the form** and see how it looks. **Start the application** and go to http://localhost:8080/article/create:



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

11. Creating Article Binding Model

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



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

```
public class ArticleBindingModel {
    @NotNull
    private String title;

    @NotNull
    private String content;
}
```

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:

```
public String getTitle() { return title; }

public void setTitle(String title) { this.title = title; }

public String getContent() { return content; }

public void setContent(String content) { this.content = content; }
```

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

12. Creating Articles Part II

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

```
@PreAuthorize("isAuthenticated()")
public String createProcess(ArticleBindingModel articleBindingModel){

}
```

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:

```
@PostMapping("/article/create")
@PreAuthorize("isAuthenticated()")
public String createProcess(ArticleBindingModel articleBindingModel){
```

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**:

```
UserDetails user = (UserDetails) SecurityContextHolder.getContext()
        .getAuthentication().getPrincipal();
```

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:

```
User userEntity = this.userRepository.findByEmail(user.getUsername());
```

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**:

```
Article articleEntity = new Article(
        articleBindingModel.getTitle(),
        articleBindingModel.getContent(),
        userEntity
);
```

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

```
this.articleRepository.saveAndFlush(articleEntity);
```

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

```
return "redirect:/";
```

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:

```
@PreAuthorize("isAuthenticated()")
@PostMapping("/article/create")
public String createProcess(ArticleBindingModel articleBindingModel) {
    UserDetails user = (UserDetails) SecurityContextHolder.getContext()
            .getAuthentication().getPrincipal();

    User userEntity = this.userRepository.findByEmail(user.getUsername());

    Article articleEntity = new Article(
            articleBindingModel.getTitle(),
            articleBindingModel.getContent(),
            userEntity
    );

    this.articleRepository.saveAndFlush(articleEntity);

    return "redirect:/";
}
```

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

13. Add Button to the Base Layout

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

```
<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>
```

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**:

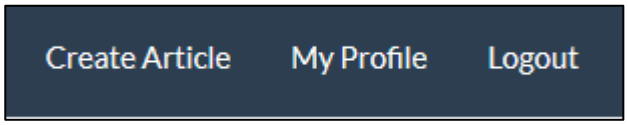
```
<li sec:authorize="isAuthenticated()">
    <a th:href="@{/article/create}">
        Create Article
```

```
    </a>
</li>
```

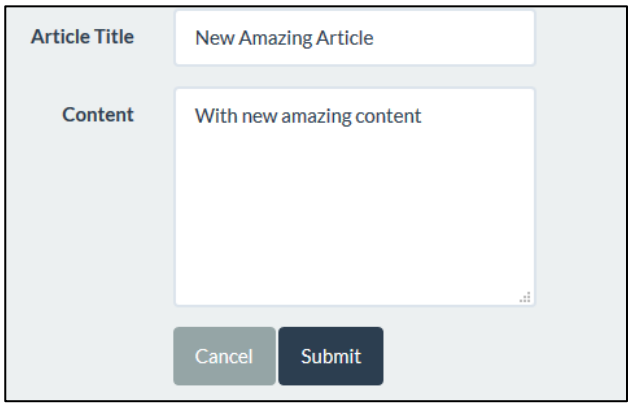
It should look like this now:

```
<ul class="nav navbar-nav navbar-right">
    <li sec:authorize="isAuthenticated()">
        <a th:href="@{/article/create}">
            Create Article
        </a>
    </li>
    <li sec:authorize="isAuthenticated()">
        <a th:href="@{/profile}">
            My Profile
        </a>
    </li>
```

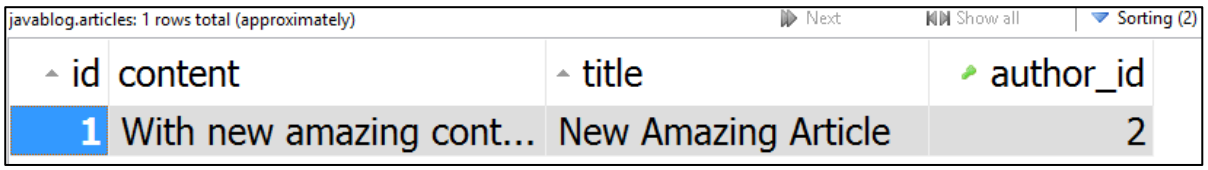
Let's test our application:



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**.

VII. List Articles

1. Listing All Articles

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

```
public String getSummary(){
}
```

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

```
@Transient
public String getSummary(){
```

There are other annotations that can manipulate what goes into the database. Now we just need return half of our content:

```
return this.getContent().substring(0, this.getContent().length() / 2) + "...";
```

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

Open the **`HomeController`** and find the **`index()`** method:

```
@GetMapping("/")
public String index(Model model) {
    model.addAttribute("view", "home/index");
    return "base-layout";
}
```

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

```
@Autowired
private ArticleRepository articleRepository;
```

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

```
List<Article> articles = this.articleRepository.findAll();
```

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

```
@GetMapping("/")
public String index(Model model) {

    List<Article> articles = this.articleRepository.findAll();

    model.addAttribute("view", "home/index");
    model.addAttribute("articles", articles);
    return "base-layout";
}
```

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**.

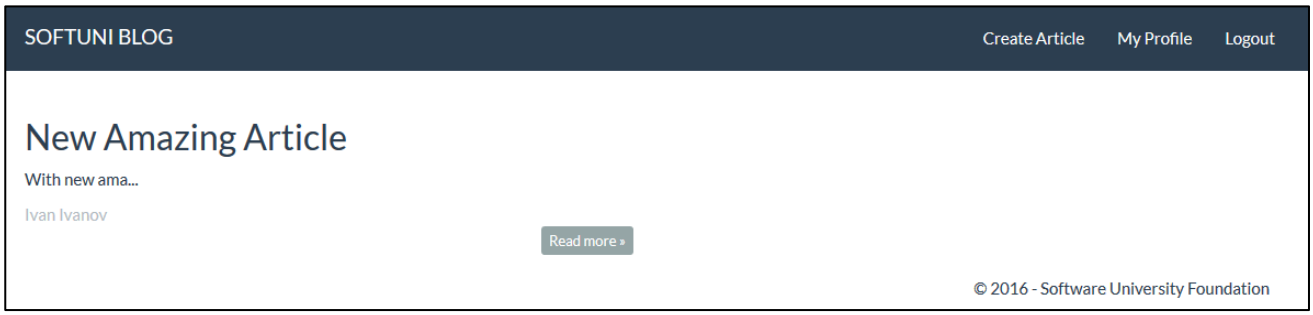
Inside, there is other attribute - "**th:each="article : ${articles}"**". This is a **foreach** loop. Thy syntax is really similar to the **foreach** loop in Java. We are getting each **article** from the **articles** collection that we've sent to our view. One thing that has to be mentioned is that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

```
{id}(id=${article.id})
```

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.

2. Single Article Details

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

Create new method in our **ArticleController** called "**details**":

```
@GetMapping("/article/{id}")
public String details(Model model, @PathVariable Integer id){

}
```

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:

```
if (!this.articleRepository.exists(id)) {
    return "redirect:/";
}
```

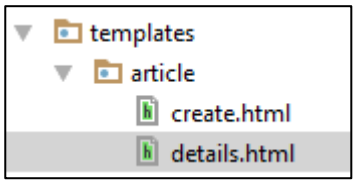
The next thing is to get the article from the database using our repository:

```
Article article = this.articleRepository.findOne(id);
```

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:

```
model.addAttribute("article", article);
model.addAttribute("view", "article/details");

return "base-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. ☺

VIII. Editing Articles

1. Creating the Get Method

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

```
@GetMapping("/article/edit/{id}")
@PreAuthorize("isAuthenticated()")
public String edit(@PathVariable Integer id, Model model) {

}
```

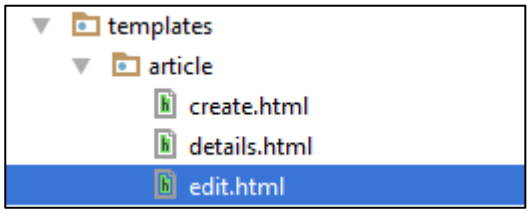
This method will be **similar** to our **details** method:

```
if (!this.articleRepository.exists(id)) {
    return "redirect:/";
}
Article article = this.articleRepository.findOne(id);

model.addAttribute("view", "article/edit");
model.addAttribute("article", article);

return "base-layout";
```

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

2. Creating the View

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

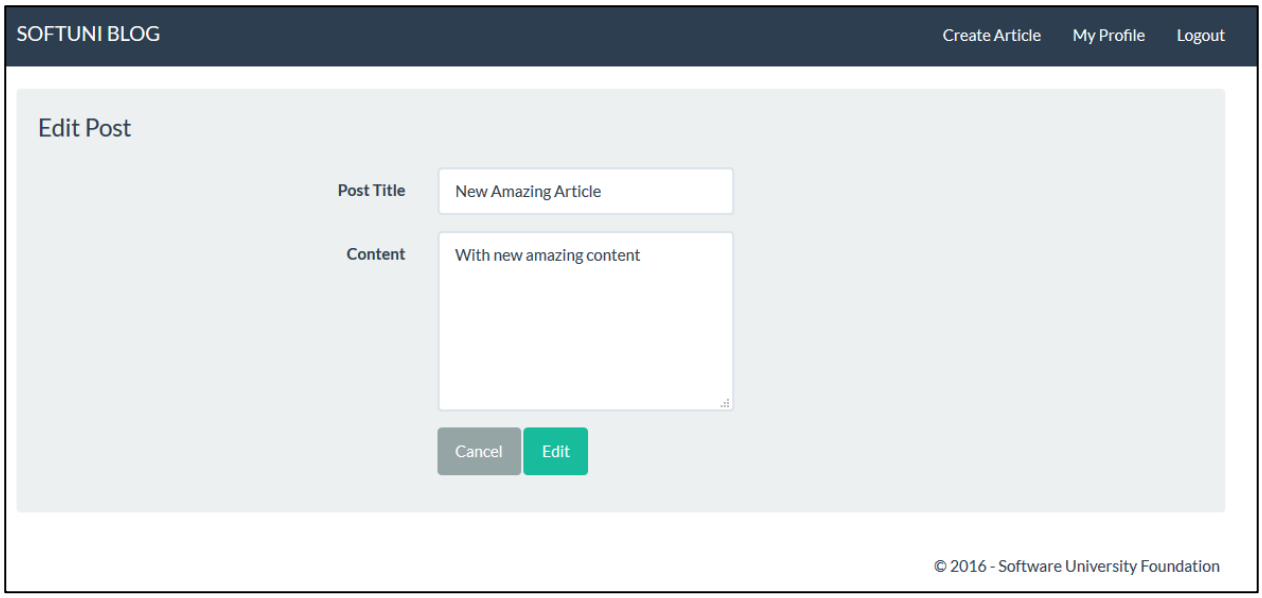


Use the following code:

```
<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.

3. Creating the Post Method

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

```
@PostMapping("/article/edit/{id}")
@PreAuthorize("isAuthenticated()")
public String editProcess(@PathVariable Integer id, ArticleBindingModel articleBindingModel) {

}
```

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:

```
if (!this.articleRepository.exists(id)) {
    return "redirect:/";
}
Article article = this.articleRepository.findOne(id);
```

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

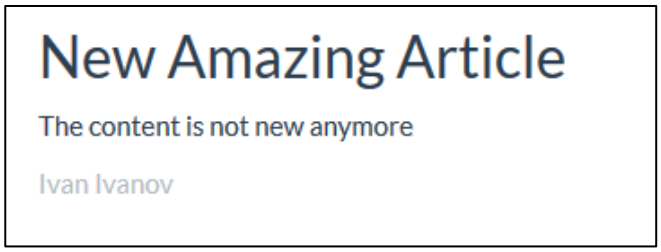
```
article.setContent(articleBindingModel.getContent());
article.setTitle(articleBindingModel.getTitle());

this.articleRepository.saveAndFlush(article);
```

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

```
return "redirect:/article/" + article.getId();
```

Test it and see if it works:

It should be working just fine:



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

4. Hiding Buttons

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

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

```
<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>
```

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

```
<th:block sec:authorize="isAuthenticated()">
    <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>
</th:block>
```

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

IX. Deleting Articles

1. Creating the Get Method

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

```
@GetMapping("article/delete/{id}")
@PreAuthorize("isAuthenticated()")
public String delete(Model model, @PathVariable Integer id) {

}
```

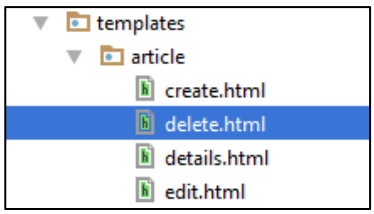
You should write the following code, that may look familiar:

```
if (!this.articleRepository.exists(id)) {
    return "redirect:/";
}
Article article = this.articleRepository.findOne(id);

model.addAttribute("view", "article/delete");
model.addAttribute("article", article);

return "base-layout";
```

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

2. Creating the View

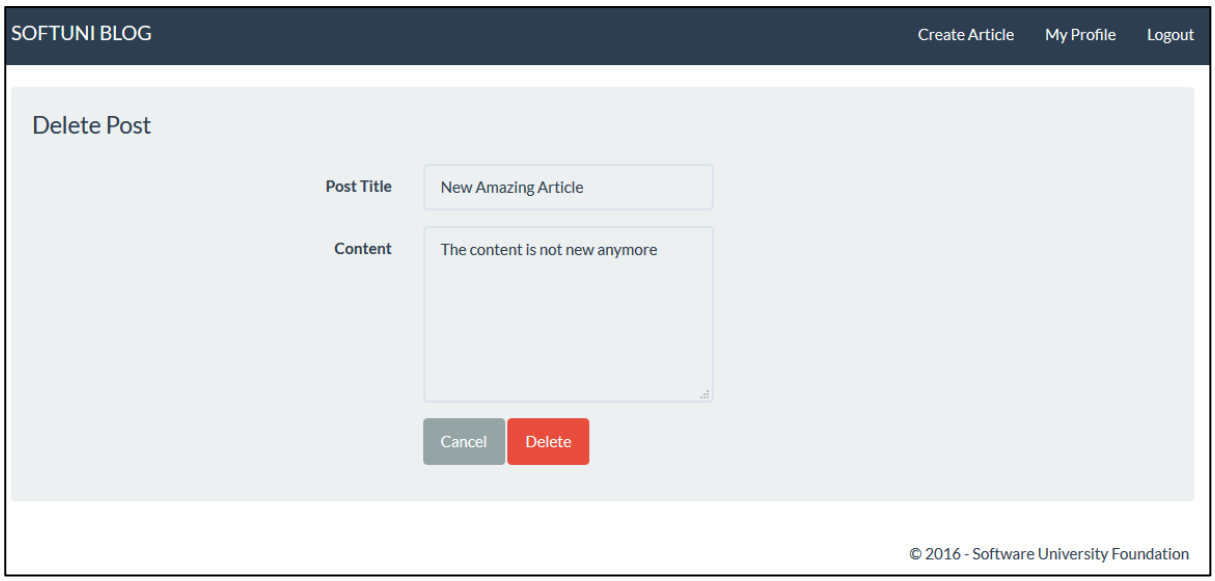
You need to **create a new view** called "**`delete`**" in the "**`article`**" folder:



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.

3. Creating the Post Method

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

```
@PostMapping("/article/delete{id}")
@PreAuthorize("isAuthenticated()")
public String deleteProcess(@PathVariable Integer id) {

}
```

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:

```
if (!this.articleRepository.exists(id)) {
    return "redirect:/";
}

Article article = this.articleRepository.findOne(id);
```

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:

```
this.articleRepository.delete(article);
```

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

```
return "redirect:/";
```

Overall, the code should look like this:

```
@PostMapping("/article/delete{id}")
@PreAuthorize("isAuthenticated()")
public String deleteProcess(@PathVariable Integer id) {
    if (!this.articleRepository.exists(id)) {
        return "redirect:/";
    }

    Article article = this.articleRepository.findOne(id);
    this.articleRepository.delete(article);

    return "redirect:/";
}
```

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

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