# E-Bank Documentation

*In this tutorial, we are going to create a banking software – E-Bank, Online banking system. After we are done with our application we are going to have the following functionality: User register, login, logout, a logged in user can create a bank account/s, deposit and withdraw given amount, transfer money to anouther user’s bank account, get information about transactions.*

**Chapters 2 is for advanced users. There’s a** [**skeleton**](http://svn.softuni.org/admin/svn/soft-tech/Sept-2018/Java/11-Web-Project/ebankSkeleton.zip) **which you can use and start from chapter 3.**

## What are we going to use?

### IntelliJ

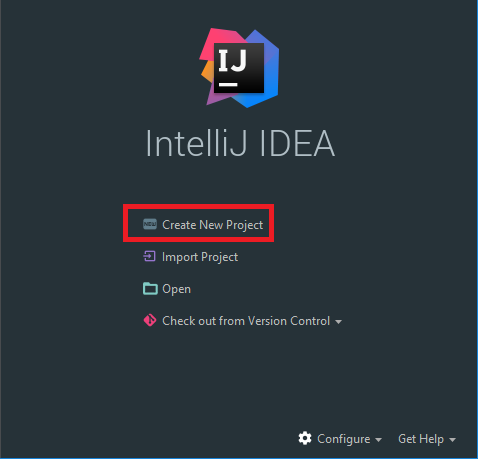
You probably already have it but if you don’t you can download it from here <https://www.jetbrains.com/idea/download/#section=windows> .

### Spring Framework

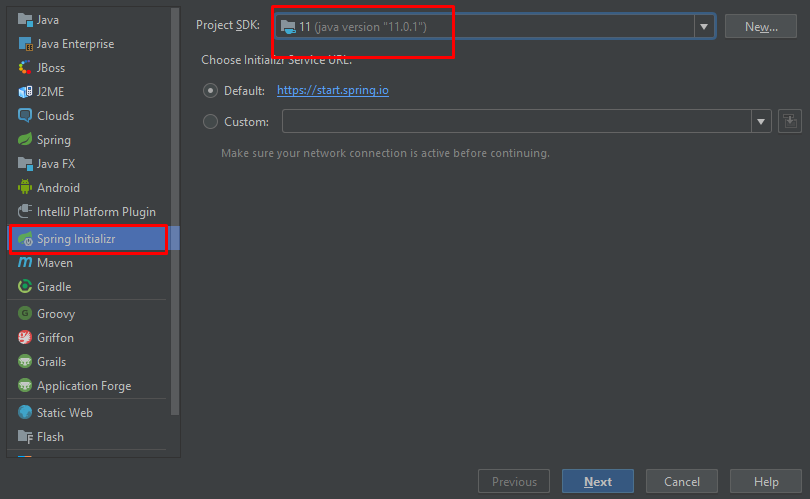
You don’t need to install anything, but if you are interested you can read more at <https://en.wikipedia.org/wiki/Spring_Framework> .

## Setting up the project

First, we start **IntelliJ** and click on “**Create New Project**”.

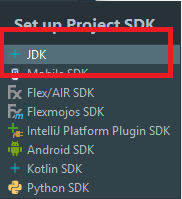


Next, we select **Spring Initializr**. After that, we must check out **Java** version and click N**ext**.

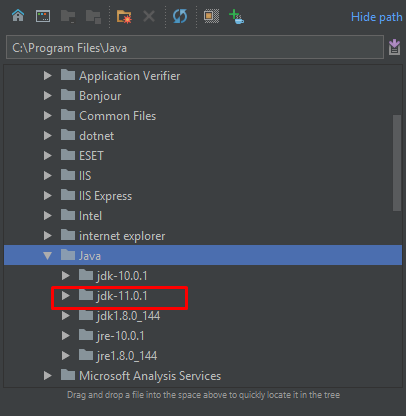


If your "**Project SDK**" field is empty follow these steps:

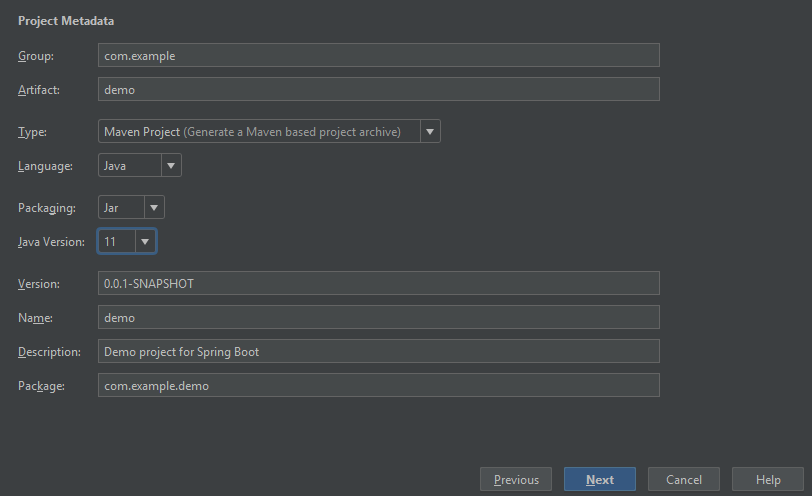
Click on "**New**". From the drop-down choose **JDK**:



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



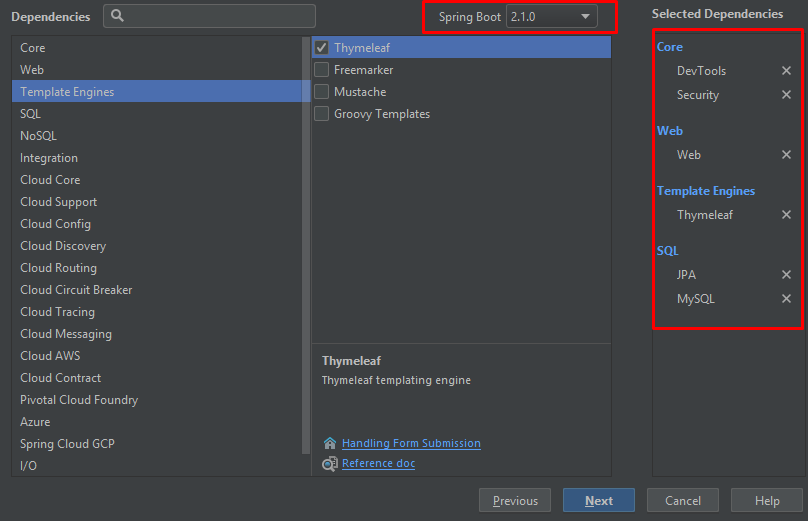
Your java should be set now, so we can continue by using the **values** from the **picture** **below**.



Now we will see **all of the things** that we **can** **include** in **our project**. We want to include only the following:

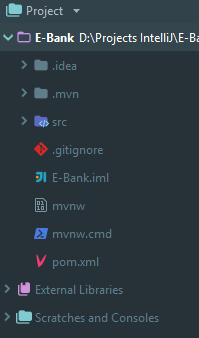
* From **Core** choose:
  + **Security**
  + **DevTools**
* From **Web** choose:
* **Web**
* From **Template Engines** choose:
  + **Thymeleaf**
* From SQL choose:
  + **MySQL**
  + **JPA**

You should have something like this:

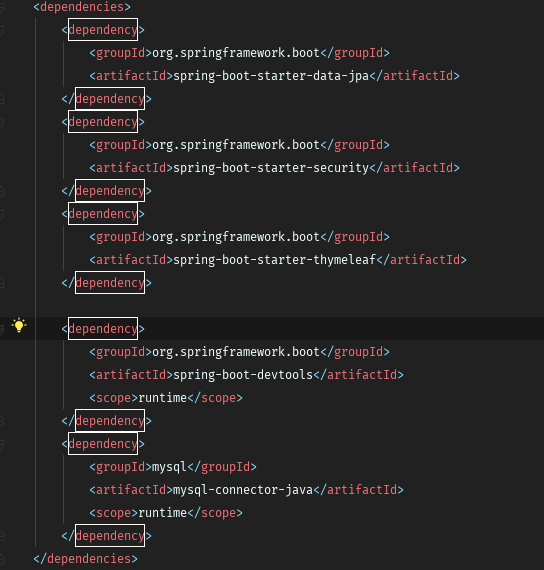


Click "**Next**" and on the final page click "**Finish**".

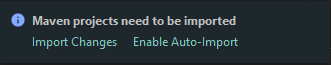
After few seconds, you should have project structure like this one:



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



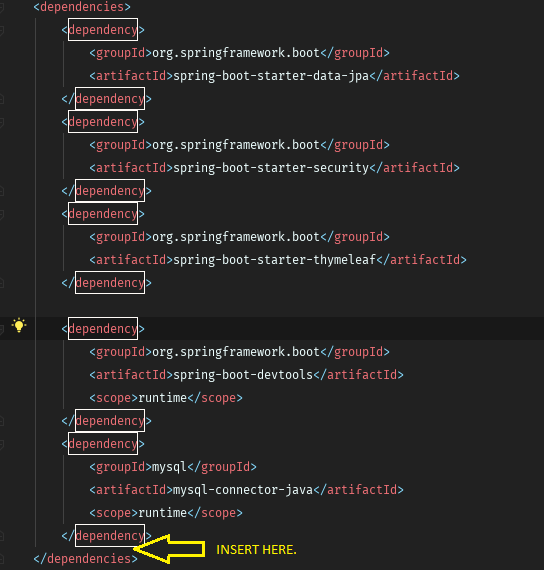
We want to **include** **additional dependency**, that will help us later. Before we continue, if you see the **following window**:



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

|  |
| --- |
| <dependency>  <groupId>org.thymeleaf.extras</groupId>  <artifactId>thymeleaf-extras-springsecurity5</artifactId>  </dependency> |

**Insert** this at the **bottom of the dependencies section,** orin other words:



We need this for security, but more for that later.

The next thing we are going to do is create the DB connection. For database, we are going to use **MySQL**. That means that you will need to have **XAMPP** [installed](https://softuni.bg/downloads/svn/soft-tech/May-2016/Software-Technologies-June-2016/06.%20Software-Technologies-XAMPP/06.%20Software-Technologies-XAMPP-Exercises.docx). Now you need to start the **MySQL module** in XAMPP and open [HeidiSQL](http://www.heidisql.com/download.php?download=installer).

Now we need to create the connection with our project. Find the file "**application.properties**":

|  |
| --- |
| # Database connection with the given database name  spring.datasource.url = jdbc:mysql://localhost:3306/e\_bank\_db?createDatabaseIfNotExist=true&useSSL=false  # Username and password  spring.datasource.username = root  spring.datasource.password =  # 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  spring.jpa.show-sql = true  # Allows Hibernate to generate SQL optimized for a particular DBMS  spring.jpa.properties.dialect = org.hibernate.dialect.MySQL5InnoDBDialect  spring.jpa.properties.hibernate.format\_sql=TRUE  #Turn off Thymeleaf cache  spring.thymeleaf.cache = false |

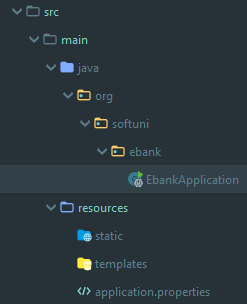
Our connection is done. We will test it later.

Our **project is ready** now.

### Project Structure

Now, let’s take a look at our 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:

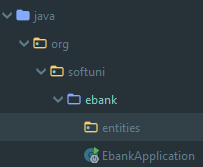


It contains main folder, which is then **separated into 2 different** folders. The first one is the "**java**" folder. This folder contains our **application package**. **Inside** that **package,** we are going to **create** our **entities**, **controllers**, **configurations**, etc. The other folder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

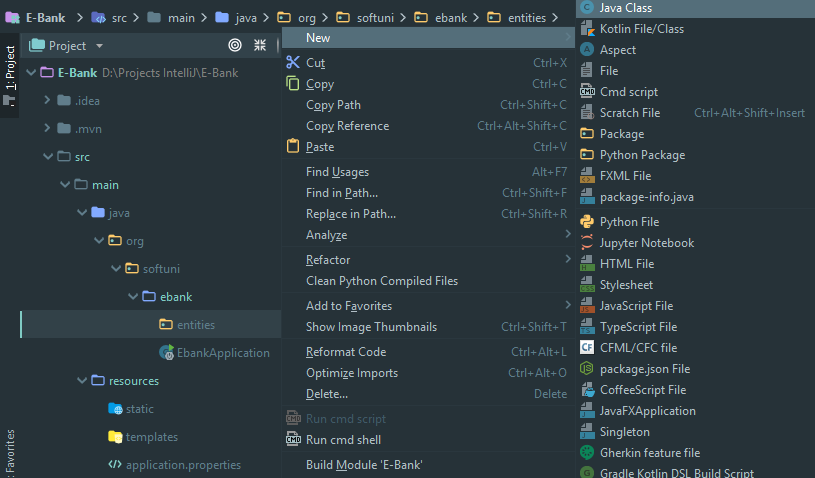
### Entities

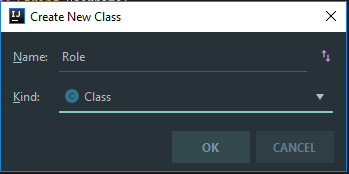
At the moment, you **cannot use your project**. Why? Because we've **imported the dependency for Spring**, that gives us the **authentication module**, but we haven’t configured it, yet. To do that we will create User **entity** using Hibernate. Then we are going to tell **Spring Security** what to use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

First, we will start with the Role entity. In **java/org/softuni/ebank** create new package called “**entities**”



This package will contain all of our entities – **users**, **roles**, **bank accounts**, **transactions**. We will start by **creating new Java** **class** called "Role":



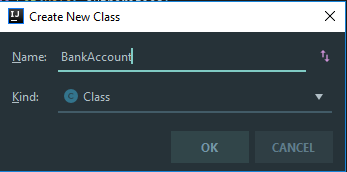


Write the following in the **Role** class:



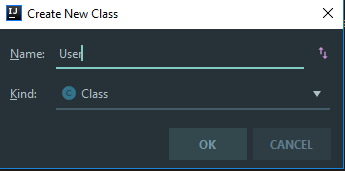
* **@Entity -** means that the Role class will become [entity](http://stackoverflow.com/questions/2550197/whats-the-difference-between-entity-and-class) that will get **saved into our database**.
* **@Table -** **defines the table name** in our **database.**
* **@Id -** tells [Hibernate](http://hibernate.org/orm/) that **this field will be the primary key** for our **database**. The second annotation make the **field generated automatically**, without us doing anything.
* **@Column –** **defines the column name** in our **table**.

Then, we need to create a **BankAccount** class in the “**entites**” package:

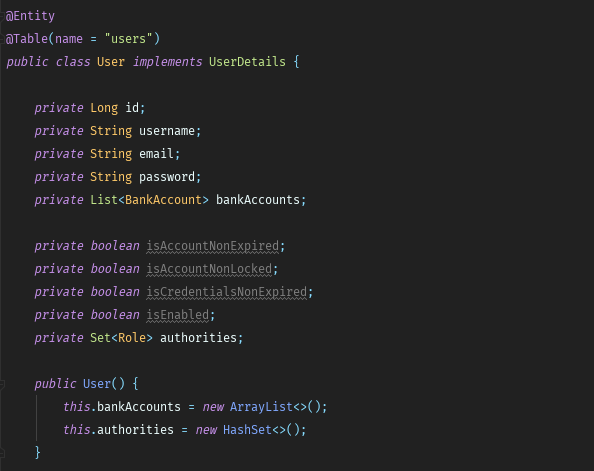


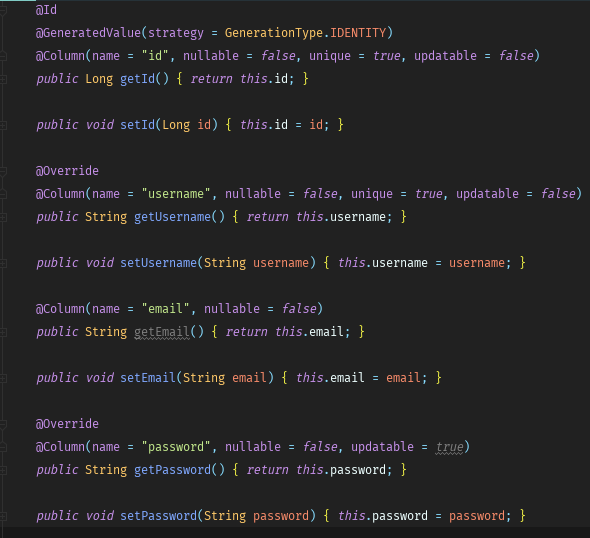
For now, we are leaving it, later we will get back to that class.

Next, we will create a **User** class in the “**entites**” package:



Write the following in it:



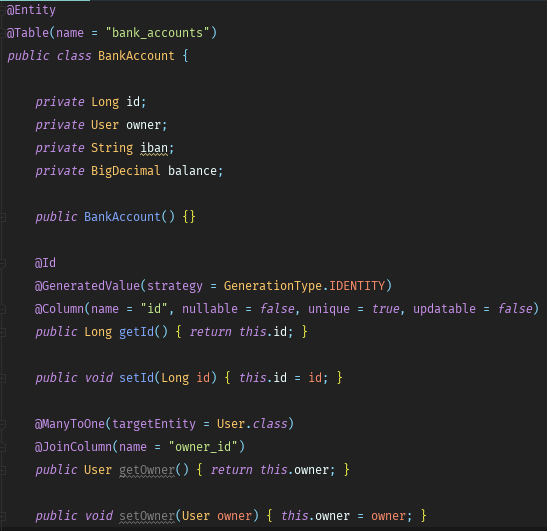


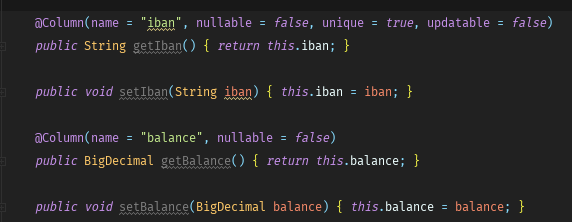




* **@OneToMany -** represents [OneToMany](https://en.wikipedia.org/wiki/One-to-many_(data_model)) relationship from the side of the “one”. Because we are working with the User entity, we are telling **Hibernate** that **one of our users** will correspond **to many bank accounts**.
* **@ManyToMany -** annotation we are telling that our "fetch" will be of type "EAGER". It basically means that we want the **roles to be loaded** **together** with the **user**. **Usually that will happen** when we want to **use the roles**, but that's an [advanced topic](https://howtoprogramwithjava.com/hibernate-eager-vs-lazy-fetch-type/). The other annotation will create the **joining table** for our relation and will **name it** "users\_roles".

Now let’s head back to our **BankAccount** entity. Write some code, so it look like this.



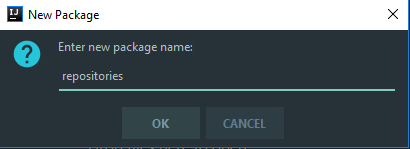


Ok. Those are our entities for now. Let’s see what we have next.

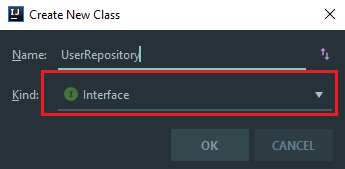
### Repositories

Our next step is to get our entities from the database. We will do this by creating a some repositories.

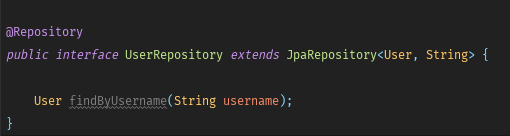
In In **java/org/softuni/ebank** create new package called “**repositories**”:



In it create **UserRepository**, the important thing here is that it won’t be a class, it will be an [interface](http://tutorials.jenkov.com/java/interfaces.html). The interface is a special type, which **can't contain functional methods**. It can **only** **declare them**.

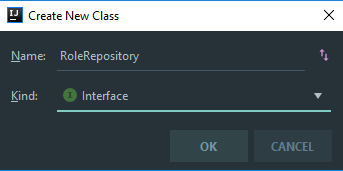


In it, write the following:

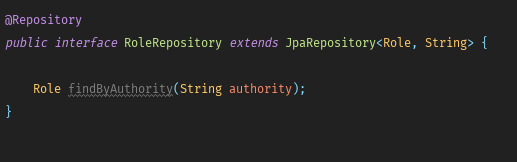


As you can see, this **method is different**. It doesn't have body. Using magic (and [reflection](http://www.javatpoint.com/java-reflection)) **Spring** will find a **user** **by** **his username**. It will use **reflection** to get the **type** of the **repository**, which is our entity "User", then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

Ok, it is time to create the **RoleRepository** interface:

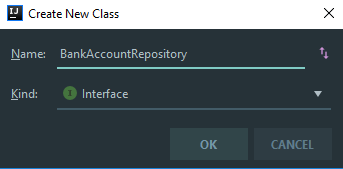


After that, write the following code in it:

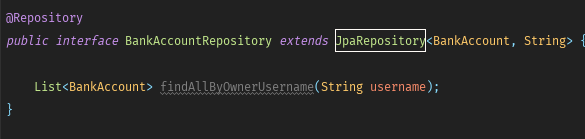


The method will get us the **role** with **given name**. It is almost the same as the method in the UserRepository, but the **criteria** and **return types** are **different**.

Here comes the **BankAccount** repository.



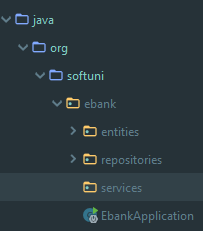
In it, we write this.



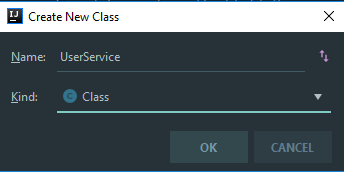
We are ready with our repositories for now.

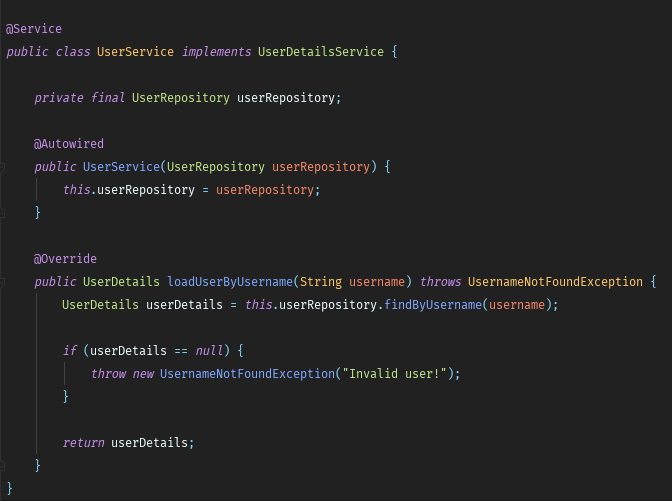
### Services

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 "services" in **java/org/softuni/ebank**.



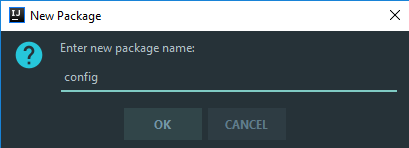
In the **services** package create class **UserService** with the following content:





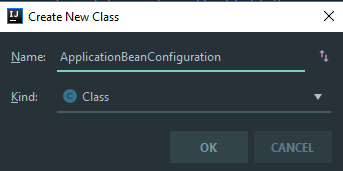
### S-Configurations

We've got to the point, where we need to configure our **Security** module. We should start by creating a new "config" package in **java/org/softuni/ebank.**

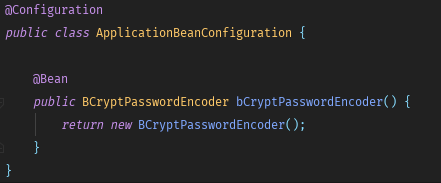


In our **config** package we will create two classes.

The first one is called **ApplicationBeanConfiguration:**

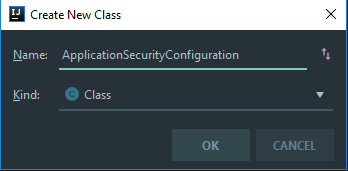


We will be using this class for configuring our **BCryptPasswordEncoder** used for encoding users’ passwords.



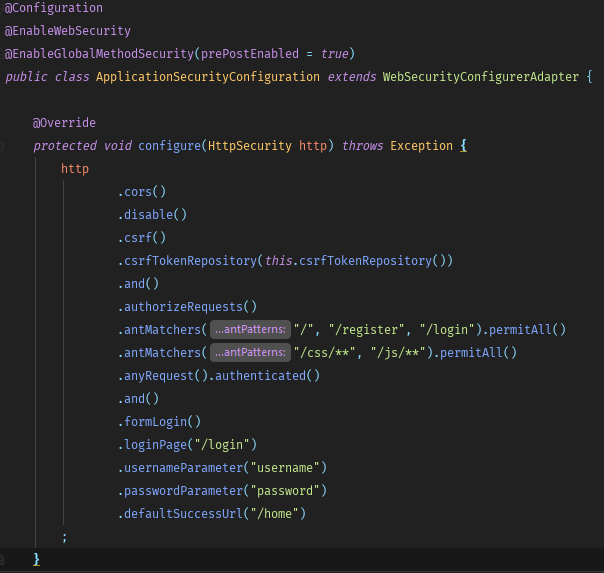
* **@Configuration -** indicates that a class declares one or more **@Bean** methods and may be processed by Spring, but more for that in the Java MVC Frameworks – Spring @ SoftUni.

The second class that we must implement in the **config** package is **ApplicationSecurityConfiguration**:



Here is the part, where things can get very messy. You must **be very careful** when creating your Security Configuration, check two or three times that you re-wrote the configuration correct. In our configuration class we will have two methods.

The first one is called **configure** and it is used to configure our permited urls and login parameters:



The second one generates a token repository for our logged in user.

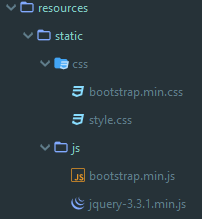


We are ready with our security.

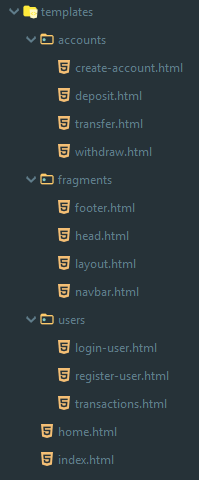
### S-Views

The next thing we have to do is create our views, but first let’s take a look at our **resources folder**.

We have **static** resources like stylesheets and scripts(those will be given to you).

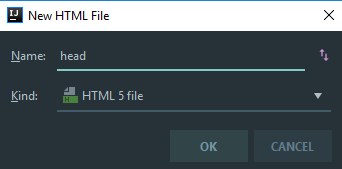


And we have a **templates** folder. Our views stay in templates folder.



Every time we want to create a view the html code will be given. For now we have to create the following things.

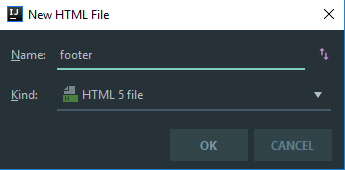
In **fragments** folder(you will have to create it in **resources/templates** if you don’t have it ) create new **html file** called **head**:



**Delete** everything from it and **paste** the following html code.

|  |
| --- |
| <th:block th:fragment>  <meta charset="UTF-8">  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>E-Bank</title>  <link rel="stylesheet" th:href="@{/css/bootstrap.min.css}">  <link rel="stylesheet" th:href="@{/css/style.css}">  <script src="/js/jquery-3.3.1.min.js"></script>  <script src="/js/bootstrap.min.js"></script>  </th:block> |

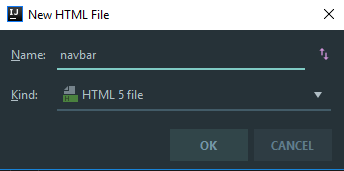
Next fragment we are going to create is called **footer**(it stays at the bottom of our pages).



Again **delete** everything and paste this:

|  |
| --- |
| <th:block th:fragment>  <footer class="footer py-3 fixed-bottom">  <div class="container-fluid">  <div class="container-fluid text-center">© CopyRight Abazov Web Design Studios 2018. All rights reserved.</div>  </div>  </footer>  </th:block> |

The third fragment in our **resources/templates/fragments** will be the **navbar** fragment.

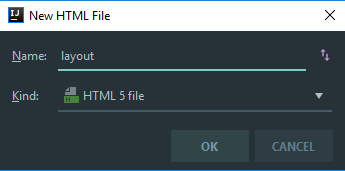


Again we **delete** everything from it and **paste**:

|  |
| --- |
| <th:block th:fragment>  <nav class="navbar navbar-expand-lg navbar-dark bg-color-dark">  <a class="navbar-brand h4" th:href="@{/}">E-Bank™</a>  <button class="navbar-toggler navbar-toggler-right" type="button" data-toggle="collapse" data-target="#navbarSupportedContent" aria-controls="navbarSupportedContent" aria-expanded="false" aria-label="Toggle navigation">  <span class="navbar-toggler-icon"></span>  </button>  <div class="collapse navbar-collapse justify-content-end row" id="navbarSupportedContent">  <ul class="navbar-nav ml-auto">  <li class="nav-item active col-md-4">  <a class="nav-link h5" th:href="@{/home}">Home</a>  </li>  <th:block sec:authorize="isAnonymous()">  <li class="nav-item active col-md-4">  <a class="nav-link h5" th:href="@{/login}">Login</a>  </li>  <li class="nav-item active col-md-4">  <a class="nav-link h5" th:href="@{/register}">Register</a>  </li>  </th:block>  <th:block sec:authorize="isAuthenticated()">  <li class="nav-item active col-md-4">  <a class="nav-link h5" th:href="@{/transactions}">Transactions</a>  </li>  <li class="nav-item active col-md-4">  <form th:action="@{/logout}" th:method="post">  <button class="nav-link h5 border-0 text-white bg-transparent" th:href="@{/logout}">Logout</button>  </form>  </li>  </th:block>  </ul>  </div>  </nav>  </th:block> |

What buttons will be shown in the navbar depends on that are we **logged in** or **not**. That’s come from **Thymeleaf Security** dependency we added earlier.

Our final fragment will be the **layout fragment.**



You guesed right. We **delete** everything and **paste:**

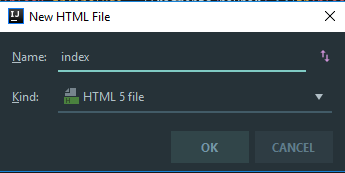
|  |
| --- |
| <!DOCTYPE html>  <html lang="en" xmlns:th="http://www.thymeleaf.org" xmlns:sec="http://www.thymeleaf.org/extras/spring-security">  <head>  <th:block th:include="~{fragments/head}"></th:block>  </head>  <body>  <header>  <th:block th:include="~{fragments/navbar}"></th:block>  </header>  <main class="bg">  <div class="jumbotron col-md-11 mx-5">  <th:block th:include="${view}"></th:block>  </div>  </main>  <footer>  <th:block th:include="~{fragments/footer}"></th:block>  </footer>  </body>  </html> |

In our layout we have our **head**, **navbar** and **footer** included via **Thymeleaf** and every time we want to open a given page, Thymeleaf will include only the needed view, so we won’t be needing to insert the head, the navbar and the footer.

### S-Home Page

Now we will need some index and home views.

Inside **resources/templates** create new html file named **index.**



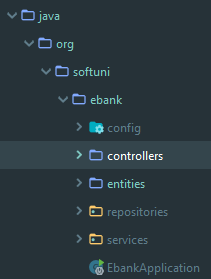
And **paste** this after you **delete** the given text

|  |
| --- |
| <th:block>  <h1 class="text-center">Welcome to E-Bank</h1>  <p class="text-center">Online Banking System</p>  <hr/>  <p sec:authorize="isAnonymous()" class="text-center">If you have an account <a th:href="@{/login}">Login</a>, if you don't - <a th:href="@{/register}">Register</a></p>  </th:block> |

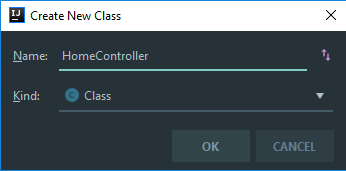
Again inside **resources/templates** create another html file named **home** and inside of it **paste**:

|  |
| --- |
| <th:block>  <h1 class="text-center" th:text="|Hello, ${username}|"></h1>  <p class="text-center">Your bank accounts are listed below.</p>  <hr />  <p class="text-center">If you want to create an account, click <a th:href="@{/accounts/create}">Here</a></p>  <div class="d-flex justify-content-center align-items-center container">  <table class="table table-striped">  <thead>  <tr>  <th scope="col">#</th>  <th scope="col">IBan</th>  <th scope="col">Balance</th>  <th scope="col">Actions</th>  </tr>  </thead>  <tbody>  <th:block th:each="account, iter : ${accounts}">  <tr>  <td scope="row" th:text="${iter.index + 1}"></td>  <td scope="row" th:text="${account.iban}"></td>  <td scope="row" th:text="${account.balance}"></td>  <td scope="row">  <a th:href="@{/accounts/deposit/{id}(id=${account.id})}" class="btn btn-success">Deposit</a>  <a th:href="@{/accounts/withdraw/{id}(id=${account.id})}" class="btn btn-danger">Withdraw</a>  <a th:href="@{/accounts/transfer/{id}(id=${account.id})}" class="btn btn-info">Transfer</a>  </td>  </tr>  </th:block>  </tbody>  </table>  </div>  </th:block> |

Next step is to create a package **controllers** inside **java/org/softuni/ebank.**



The first controller we are going to create is **HomeController** and it stays in the newly created “**controllers**” package.

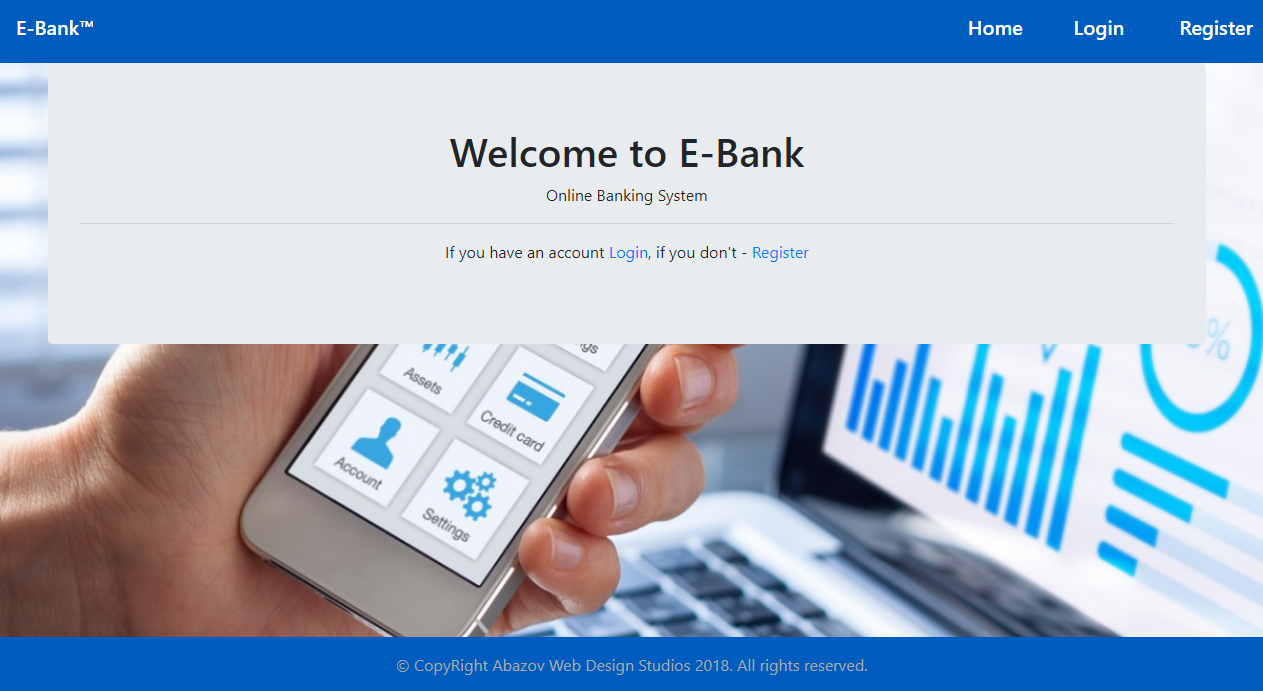
****

**HomeController** must look like this:



* **@PreAuthorize(“isAuthenticated()”) –** this means that the **home** action can be accessed only when we have **logged in** user.

Now if we run the application(and you have done everything described above correctly) we should be able to see our index page.



If you've downloaded the **skeleton**, this is your **starting point**. After importing it and run the application you should see the index page. So we have our index page and spring create a database for us but we have no functionality.

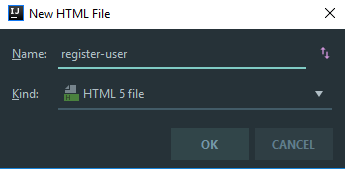
## S-User Registration and Login

If we click on register or login, nothing happens. That’s because we don’t have views and controllers to register and login.

Great, first we create some views.

Inside **resources/templates** we create a folder “**users**”**.** In it we will put our views, that are related to the user.

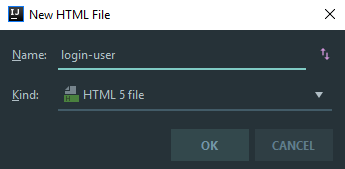
First, we create the **register** view.



Like before, we **delete** everything and we **put** this in it.

|  |
| --- |
| <th:block>  <div class="text-center">  <h1>Register</h1>  <hr />  <div class="d-flex justify-content-center align-items-center container ">  <form class="col-md-4" th:action="@{/register}" th:method="post" th:object="${userBindingModel}">  <div class="form-group">  <label for="username">Username</label>  <input type="text" class="form-control" id="username" placeholder="Enter username" th:field="\*{username}" required>  </div>  <div class="form-group">  <label for="email">Email address</label>  <input type="email" class="form-control" id="email" aria-describedby="emailHelp" placeholder="Enter email" th:field="\*{email}" required>  </div>  <div class="form-group">  <label for="password">Password</label>  <input type="password" class="form-control" id="password" placeholder="Password" th:field="\*{password}" required>  </div>  <div class="form-group">  <label for="password">Confirm Password</label>  <input type="password" class="form-control" id="confirmPassword" placeholder="Confirm Password" th:field="\*{confirmPassword}" required>  </div>  <button type="submit" class="btn btn-primary">Register</button>  </form>  </div>  </div>  </th:block> |

Next, it is time for the **login** view.

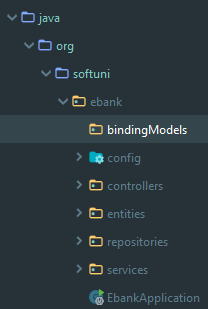


This should be pasted in this view.

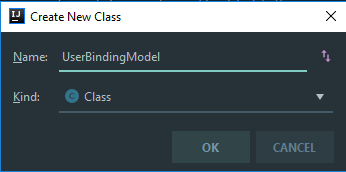
|  |
| --- |
| <th:block>  <div class="text-center">  <h1>Login</h1>  <hr />  <div class="d-flex justify-content-center align-items-center container ">  <form class="col-md-4" th:action="@{/login}" th:method="post" th:object="${userBindingModel}">  <div class="form-group">  <label for="username">Username</label>  <input type="text" class="form-control" id="username" placeholder="Enter username" th:field="\*{username}" required>  </div>  <div class="form-group">  <label for="password">Password</label>  <input type="password" class="form-control" id="password" placeholder="Password" th:field="\*{password}" required>  </div>  <input type="hidden" th:name="${\_csrf.parameterName}" th:value="${\_csrf.token}">  <button type="submit" class="btn btn-primary">Login</button>  </form>  </div>  </div>  </th:block> |

Enough for the views, next thing we have to do is the so-called **BindingModel.**

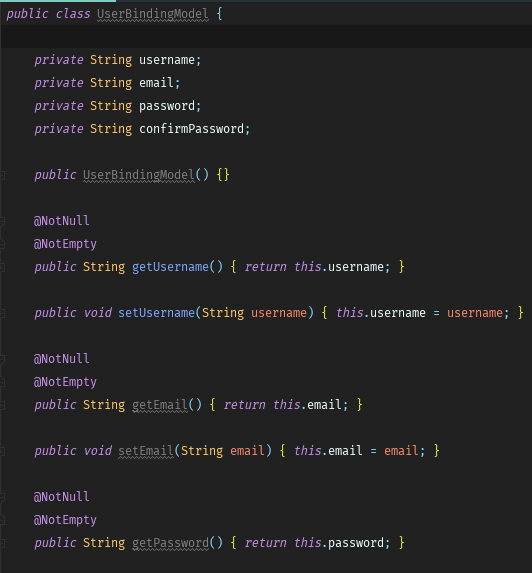
Inside **java/org/softuni/ebank** create new package “**bindingModels**”.

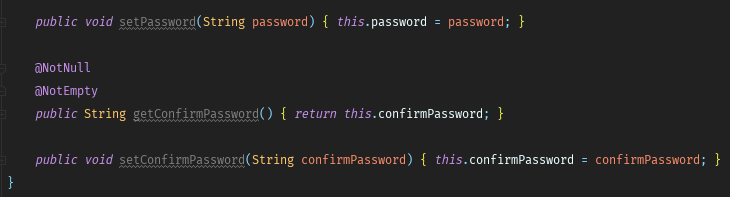


In the “**bindingModels**” package, we must create a new class called **UserBindingModel**.

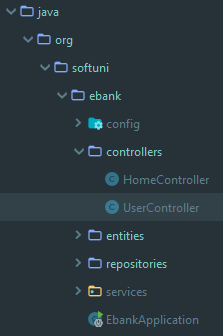


After that, we have to create some fields and getters and setters in it, so it look like this.





Great, now it is time to create a **UserController** class inside our “**controllers**” package.



Let’s start implementing the user controller. The first thing is to create **fields** and **constructor**.



Ok, next thing we have to do is to create 2 private methods, luckly you can **copy-paste** them some where in the **UserController.**

|  |
| --- |
| private void seedRolesInDb() {  if (this.roleRepository.count() == 0) {  Role role = new Role();  role.setAuthority("USER");  this.roleRepository.save(role);  }  } |

This method is used for **seeding roles** in database, whenever we register our first user.

|  |
| --- |
| private boolean registerUser(UserBindingModel userBindingModel) {  User user = this.userRepository.findByUsername(userBindingModel.getUsername());  if (user != null) {  return false;  } else if (!userBindingModel.getPassword().equals(userBindingModel.getConfirmPassword())) {  return false;  }  this.seedRolesInDb();  Role role = this.roleRepository.findByAuthority("USER");  if (role == null) {  return false;  }  user = new User();  user.setUsername(userBindingModel.getUsername());  user.setEmail(userBindingModel.getEmail());  user.setPassword(this.bCryptPasswordEncoder.encode(userBindingModel.getPassword()));  user.getAuthorities().add(role);  this.userRepository.save(user);  return true;  } |

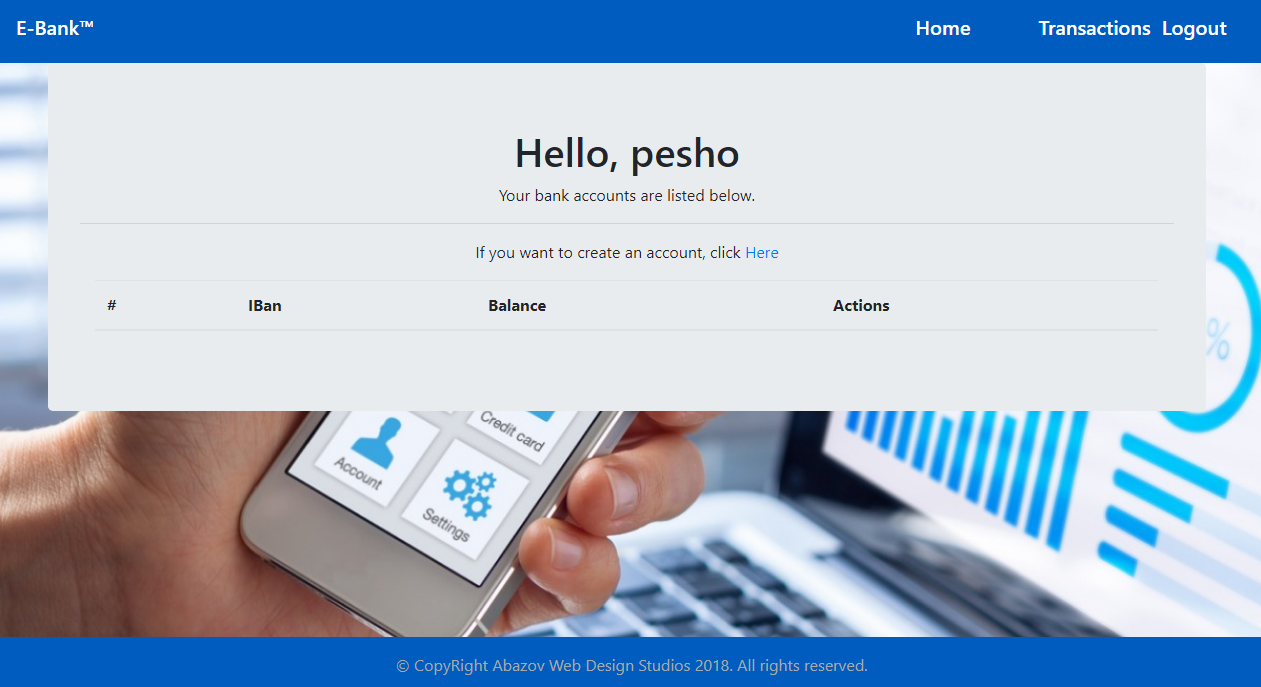
The second one is the so-called bussines logic about **registering** the user and seeding it into the database.

Ok. Enough with these methods, now finish your **UserController** to look like this.





We are done with our register and login actions and if we worked correctly, when we start our application, register user and login, we should see this.



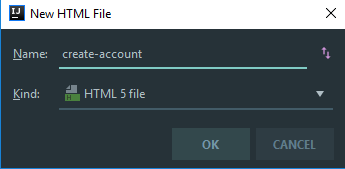
## Bank Accounts

Ok, so far, so good. Now we have to implement the logic about creating a bank account.

Again, we start with the view.

Inside **resources/templates** we create a folder “**accounts**”**.** In it we will put our views, that are related to the bank accounts.

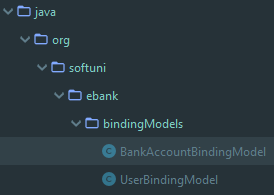
Let’s create a new html file called “**create-account**”.



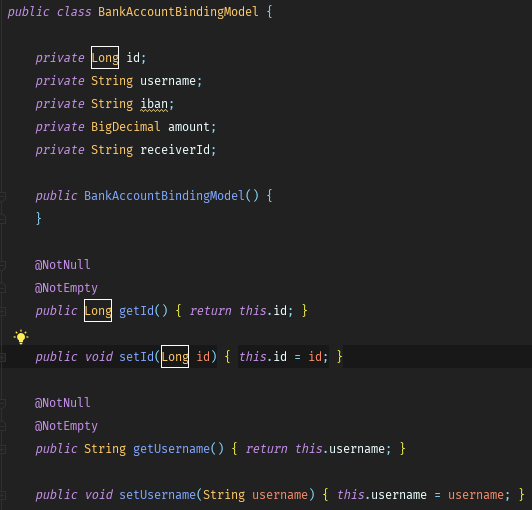
As you already know, we delete everything in that file and we paste this.

|  |
| --- |
| <th:block>  <div class="text-center">  <h1>Create Account</h1>  <hr />  <div class="d-flex justify-content-center align-items-center container ">  <form class="col-md-4" th:action="@{/accounts/create}" th:method="post" th:object="${bankAccountBindingModel}">  <div class="form-group">  <label for="username">Username</label>  <input type="text" class="form-control" id="username" placeholder="Enter username" th:field="\*{username}" readonly>  </div>  <div class="form-group">  <label for="iban">IBan</label>  <input type="text" class="form-control" id="iban" placeholder="IBan" th:field="\*{iban}" required>  </div>  <button type="submit" class="btn btn-primary">Create</button>  </form>  </div>  </div>  </th:block> |

Next thing we do, inside the “**bindingModels**” package create **BankAccountBindingModel.**



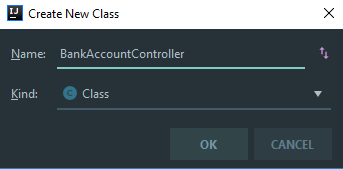
Your binding model must look like this.



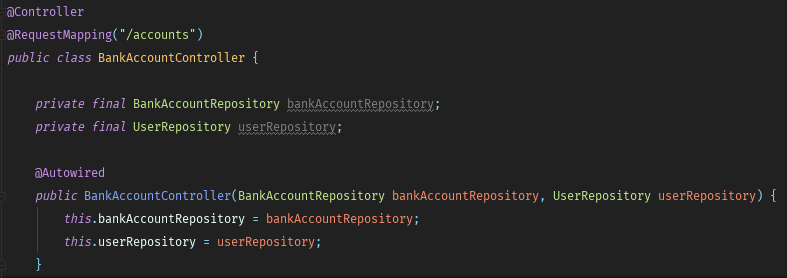


Let’s implement our controller now.

Inside the “**controllers**” package we create **BankAccountController** class



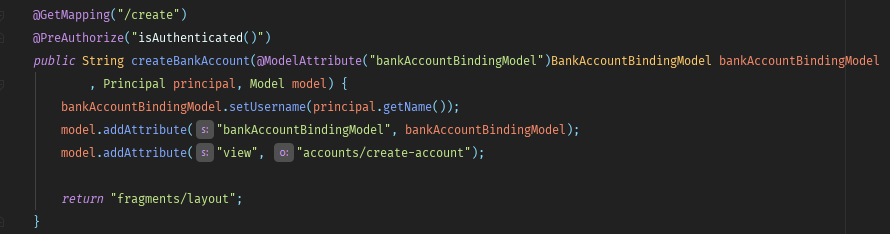
Again, we have to create **fields** and **constructor.**



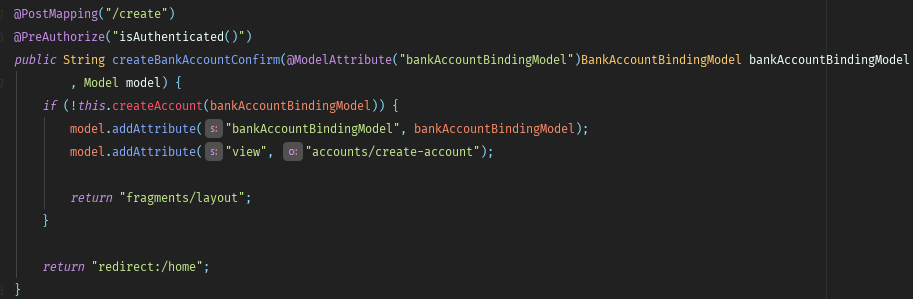
Now we will implement our logic for creating a bank account.

Again we have **two methods**. **Get** method and **Post** method.

We start with the **get** method



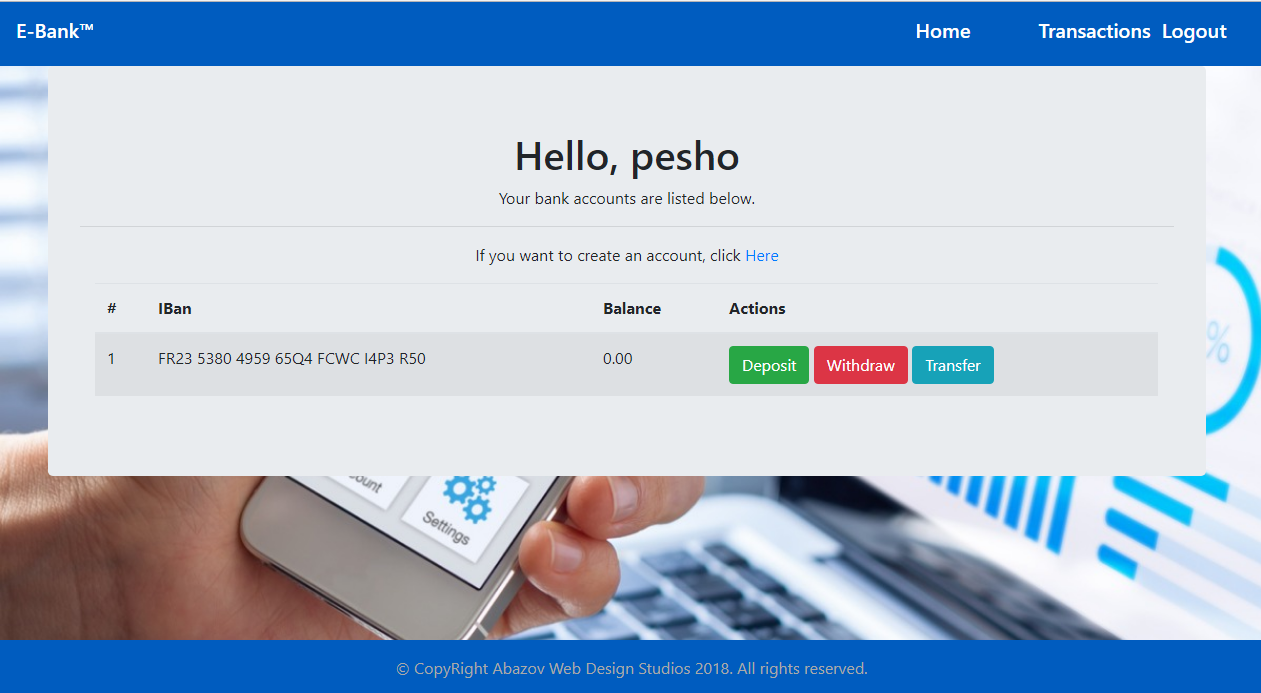
Now, the **post** method.



Don’t worry “**createAccount**” goes red. Copy the given method below.

|  |
| --- |
| private boolean createAccount(BankAccountBindingModel bankAccountBindingModel) {  if (bankAccountBindingModel.getIban() == null || bankAccountBindingModel.getIban().equals("")) {  return false;  }  User user = this.userRepository.findByUsername(bankAccountBindingModel.getUsername());  if (user == null) {  return false;  }  BankAccount bankAccount = new BankAccount();  bankAccount.setOwner(user);  bankAccount.setIban(bankAccountBindingModel.getIban());  bankAccount.setBalance(BigDecimal.ZERO);  this.bankAccountRepository.save(bankAccount);  return true;  } |

That’s all for creating an account. If you worked correctly your home page should look like this after creating an account.

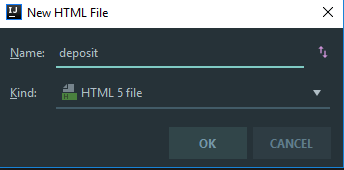


## Deposit and Withdraw

It is time to create deposit and withdraw actions.

Again we start by creating our views. Inside “**resources/templates/accounts**” we create two views.

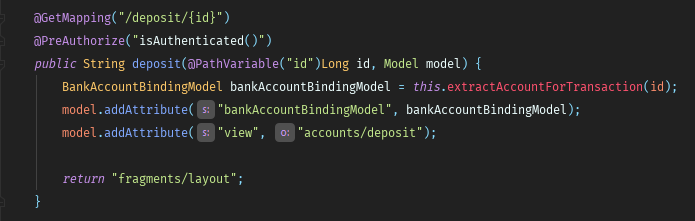
The first one is the **deposit** view.



Put this inside of it.

|  |
| --- |
| <th:block>  <div class="text-center">  <h1>Deposit</h1>  <hr />  <div class="d-flex justify-content-center align-items-center container ">  <form class="col-md-4" th:action="@{/accounts/deposit/{id}(id=\*{id})}" th:method="post" th:object="${bankAccountBindingModel}">  <div class="form-group">  <label for="username">Username</label>  <input type="text" class="form-control" id="username" placeholder="Enter username" th:field="\*{username}" readonly>  </div>  <div class="form-group">  <label for="iban">IBan</label>  <input type="text" class="form-control" id="iban" placeholder="IBan" th:field="\*{iban}" readonly>  </div>  <div class="form-group">  <label for="amount">Amount</label>  <input type="number" step="any" class="form-control" id="amount" placeholder="Amount" th:field="\*{amount}" required>  </div>  <button type="submit" class="btn btn-success">Deposit</button>  </form>  </div>  </div>  </th:block> |

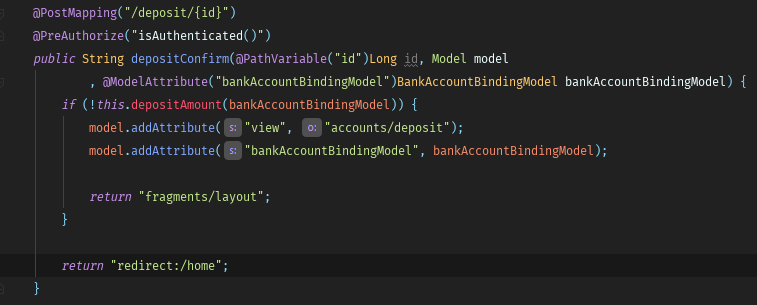
Great. Now let’s head to the **BankAccountController** and implement **get** and **post** methods for deposit. This is the **Get** method.



And here it is the “**extractAccountForTransaction(id)**” method.

|  |
| --- |
| private BankAccountBindingModel extractAccountForTransaction(Long id) {  BankAccount bankAccount = this.bankAccountRepository.findById(id).orElse(null);  if (bankAccount == null) {  throw new IllegalArgumentException("Invalid Bank Account!");  }  BankAccountBindingModel bankAccountBindingModel = new BankAccountBindingModel();  bankAccountBindingModel.setId(id);  bankAccountBindingModel.setUsername(bankAccount.getOwner().getUsername());  bankAccountBindingModel.setIban(bankAccount.getIban());  return bankAccountBindingModel;  } |

And here is the **Post** method.



“**depositAmount(bankAccountBindingModel)**”:

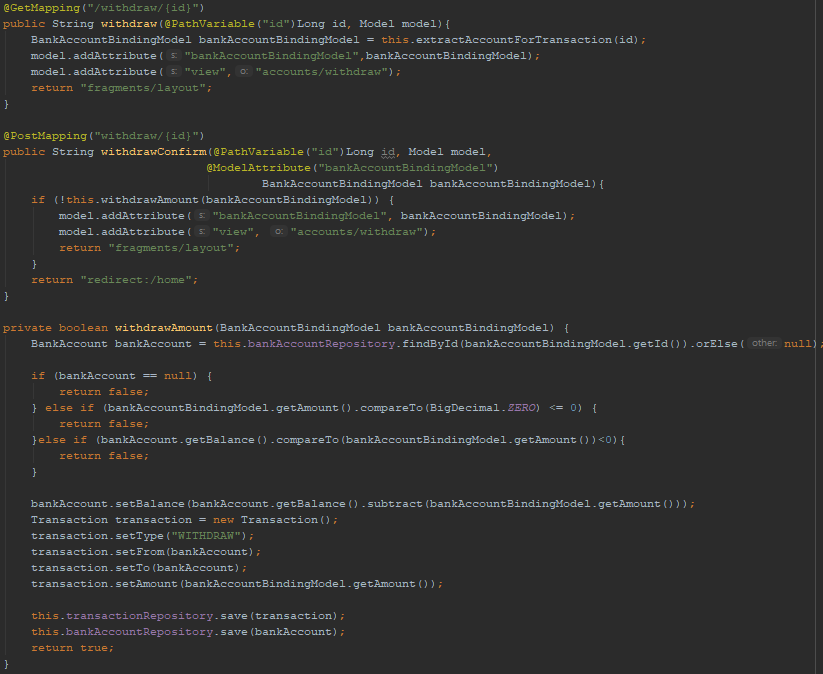
|  |
| --- |
| private boolean depositAmount(BankAccountBindingModel bankAccountBindingModel) {  BankAccount bankAccount = this.bankAccountRepository.findById(bankAccountBindingModel.getId()).orElse(null);  if (bankAccount == null) {  return false;  } else if (bankAccountBindingModel.getAmount().compareTo(BigDecimal.ZERO) <= 0) {  return false;  }  bankAccount.setBalance(bankAccount.getBalance().add(bankAccountBindingModel.getAmount()));  this.bankAccountRepository.save(bankAccount);  return true;  } |

Ok, that’s all for depositing amount into bank account.

It is up to you to implement the **withdraw** action. It is pretty much the same. Here is the view.

|  |
| --- |
| <th:block>  <div class="text-center">  <h1>Withdraw</h1>  <hr />  <div class="d-flex justify-content-center align-items-center container ">  <form class="col-md-4" th:action="@{/accounts/withdraw/{id}(id=\*{id})}" th:method="post" th:object="${bankAccountBindingModel}">  <div class="form-group">  <label for="username">Username</label>  <input type="text" class="form-control" id="username" placeholder="Enter username" th:field="\*{username}" readonly>  </div>  <div class="form-group">  <label for="iban">IBan</label>  <input type="text" class="form-control" id="iban" placeholder="IBan" th:field="\*{iban}" readonly>  </div>  <div class="form-group">  <label for="amount">Amount</label>  <input type="number" step="any" class="form-control" id="amount" placeholder="Amount" th:field="\*{amount}" required>  </div>  <button type="submit" class="btn btn-danger">Withdraw</button>  </form>  </div>  </div>  </th:block> |

Again, you must have a **get** method, a **post** method and method that is allmost the same as “**depositAmount(bankAccountBindingModel)**”



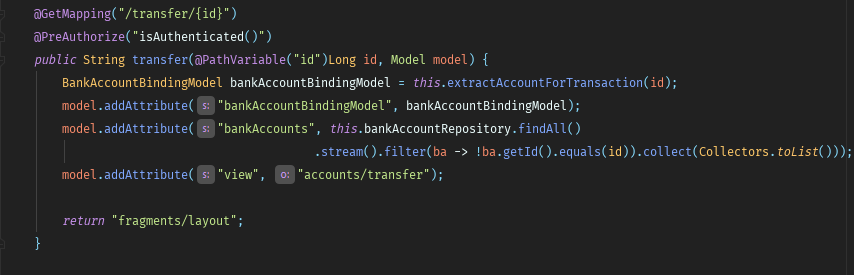
## Transfer

Again, we start with the view. Inside “**accounts**” folder create new html file called “transfer”.

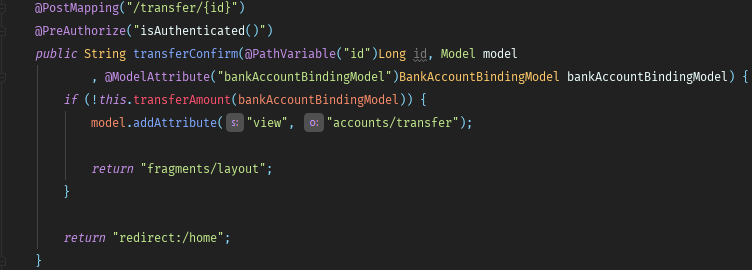
Delete everything and copy this inside of it.

|  |
| --- |
| <th:block>  <div class="text-center">  <h1>Deposit</h1>  <hr />  <div class="d-flex justify-content-center align-items-center container ">  <form class="col-md-4" th:action="@{/accounts/transfer/{id}(id=\*{id})}" th:method="post" th:object="${bankAccountBindingModel}">  <div class="form-group">  <label for="username">Username</label>  <input type="text" class="form-control" id="username" placeholder="Enter username" th:field="\*{username}" readonly>  </div>  <div class="form-group">  <label for="iban">IBan</label>  <input type="text" class="form-control" id="iban" placeholder="IBan" th:field="\*{iban}" readonly>  </div>  <div class="form-group">  <label for="receiver">Receiver IBan</label>  <select class="form-control" id="receiver" th:field="\*{receiverId}">  <th:block th:each="bankAccount : ${bankAccounts}">  <option th:value="${bankAccount.id}" th:text="|${bankAccount.owner.username} ${bankAccount.iban}|"></option>  </th:block>  </select>  </div>  <div class="form-group">  <label for="amount">Amount</label>  <input type="number" step="any" class="form-control" id="amount" placeholder="Amount" th:field="\*{amount}" required>  </div>  <button type="submit" class="btn btn-info">Transfer</button>  </form>  </div>  </div>  </th:block> |

This is the **get** method.



Now, **post**.



“**transferAmount(bankAccountBindingModel)**” method will be given to you, again paste it in this controller.

|  |
| --- |
| private boolean transferAmount(BankAccountBindingModel bankAccountBindingModel) {  BankAccount from = this.bankAccountRepository.findById(bankAccountBindingModel.getId()).orElse(null);  BankAccount to = this.bankAccountRepository.findById(bankAccountBindingModel.getReceiverId()).orElse(null);  if (from == null || to == null) {  return false;  }else if (bankAccountBindingModel.getAmount().compareTo(BigDecimal.ZERO) <= 0) {  return false;  }  from.setBalance(from.getBalance().subtract(bankAccountBindingModel.getAmount()));  if (from.getBalance().compareTo(BigDecimal.ZERO) <= 0) {  return false;  }  to.setBalance(to.getBalance().add(bankAccountBindingModel.getAmount()));  this.bankAccountRepository.save(from);  this.bankAccountRepository.save(to);  return true;  } |

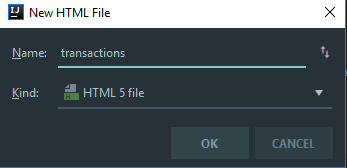
Note that you will need at least **2 accounts** to test the transfer functionality.

## Transactions

In this part we are going to implement the logic for showing a log of all transactions.

Again, we start with the view.

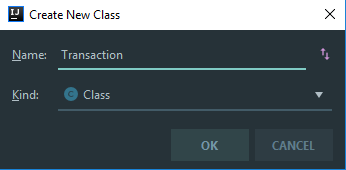
Inside **“resources/templates/users”** create view named “**transactions**”



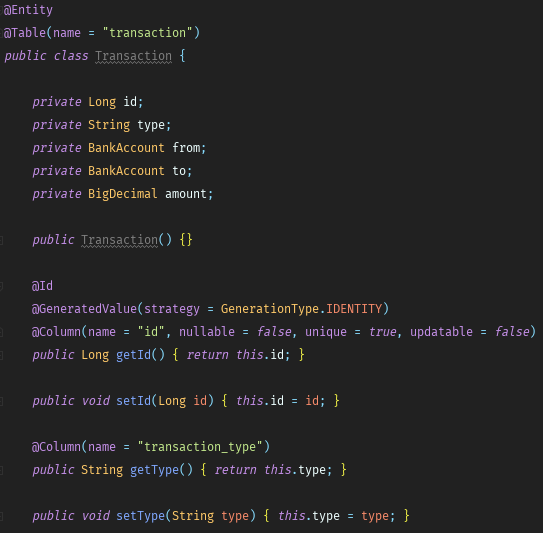
Delete the generated text and paste the following.

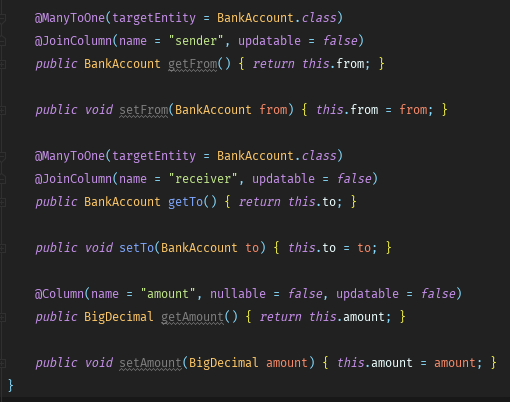
|  |
| --- |
| <th:block>  <h1 class="text-center" th:text="|Hello, ${username}|"></h1>  <p class="text-center">Your transactions are listed below.</p>  <hr />  <div class="d-flex justify-content-center align-items-center container">  <table class="table table-striped">  <thead>  <tr>  <th scope="col">#</th>  <th scope="col">Type</th>  <th scope="col">From</th>  <th scope="col">To</th>  <th scope="col">Amount</th>  </tr>  </thead>  <tbody>  <th:block th:each="transaction, iter : ${transactions}">  <tr>  <td scope="row" th:text="${iter.index + 1}"></td>  <td scope="row" th:text="${transaction.type}"></td>  <td scope="row" th:text="${transaction.from.getIban()}"></td>  <td scope="row" th:text="${transaction.to.getIban()}"></td>  <td scope="row" th:text="${transaction.amount}"></td>  </tr>  </th:block>  </tbody>  </table>  </div>  </th:block> |

Our next move will be creating a **Transaction** entity in the entities package.



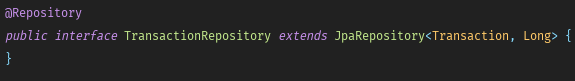
It must look like this.





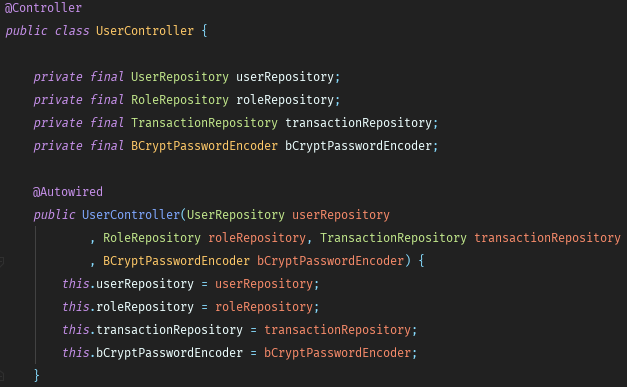
Next we need a repository for our transactions.

Inside “**repositories**” package, we create new interface “**TransactionRepository**”



Now lets head to **UserController** and create a method that gets all transactions and send them to our view.

First we need to import our newly created repository by creating a private field and adding it into the constructor. Look at the picture below.



Now the method that will **get** our transactions.



We are almost ready. Now let’s head to our **BankAccount** controller, create transaction repository private field and add it into the constructor.



Next, find our method “**depositAmount()**”. Inside of it, right before saving our entity in the database paste the following.

|  |
| --- |
| Transaction transaction = **new** Transaction();  transaction.setType("DEPOSIT");  transaction.setFrom(bankAccount);  transaction.setTo(bankAccount);  transaction.setAmount(bankAccountBindingModel.getAmount());  **this**.transactionRepository.save(transaction); |

It should look like this.

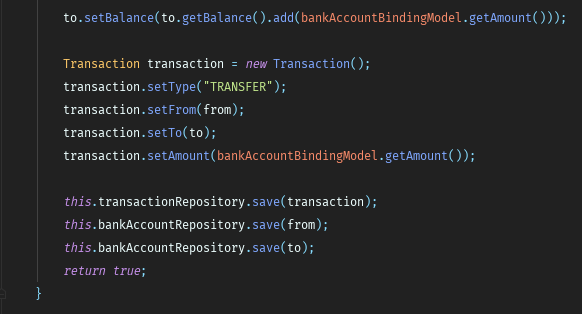


Try implementing the withdraw logic. It is the same as deposit.

Now, the last thing we need to do is going to the “**transferAmount()”** method and right before saving our entities, paste this.

|  |
| --- |
| Transaction transaction = **new** Transaction();  transaction.setType("TRANSFER");  transaction.setFrom(from);  transaction.setTo(to);  transaction.setAmount(bankAccountBindingModel.getAmount());  **this**.transactionRepository.save(transaction); |

So our method must look like this.



With that we finished our E-Bank application. Have fun. ☺