Calculator: Java and Spring

This document defines a complete walkthrough of creating a **Calculator** application with the [Spring](http://spring.io/) Framework, from setting up the framework to implementing the fully functional application.

# Setting Up JDK and IntelliJ Idea Configuration

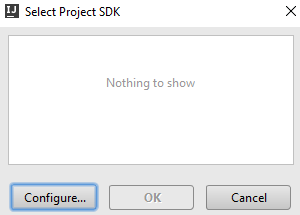
Before we start, you need to download the [Java Development Kit](http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html), also known as **JDK**. Download the "**Java SE Development Kit 8u112**". After downloading it, install it **without changing the installation directory**.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

## Set Up JDK

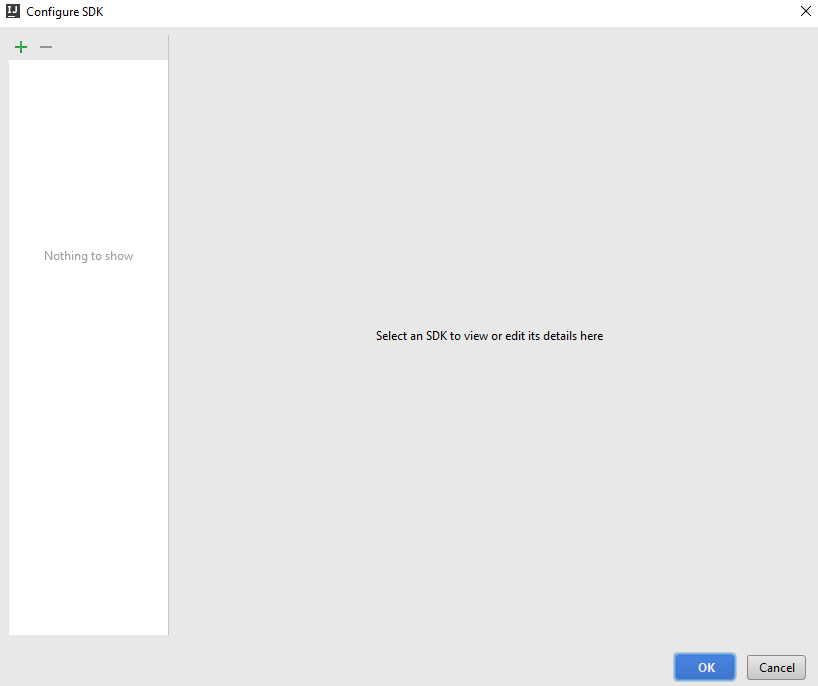
If you are using the skeleton and see something like this:

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

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



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



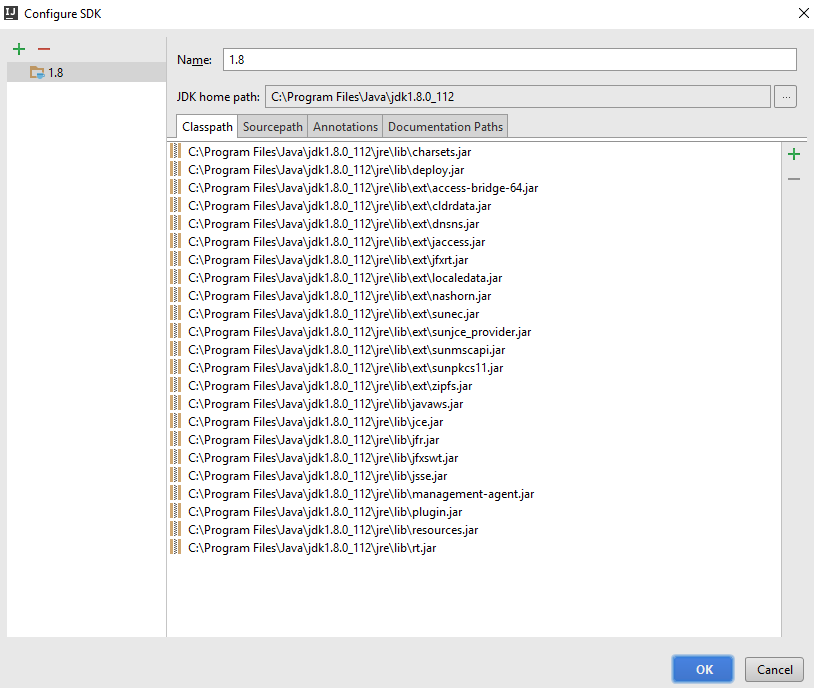
Click on the **green plus sign** in the top left corner of the window and choose **JDK**:



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



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

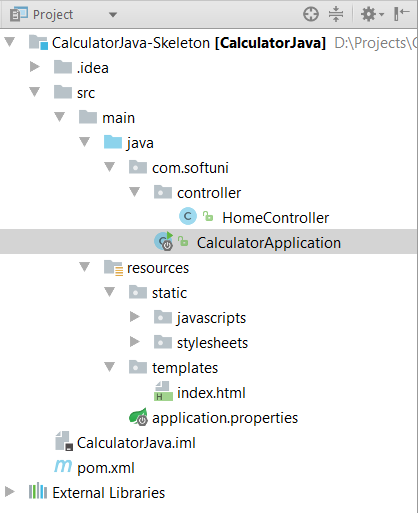


That is everything, your **JDK is now configured**.

# Base Project Overview

## Open the Project

In our project, there is only one folder we're interested in. That would be the "src" folder. That folder will **contain all of the files** we are **going to create**. Let's take a look:



We can see several folders here. Let look at them one by one and see what are they for:

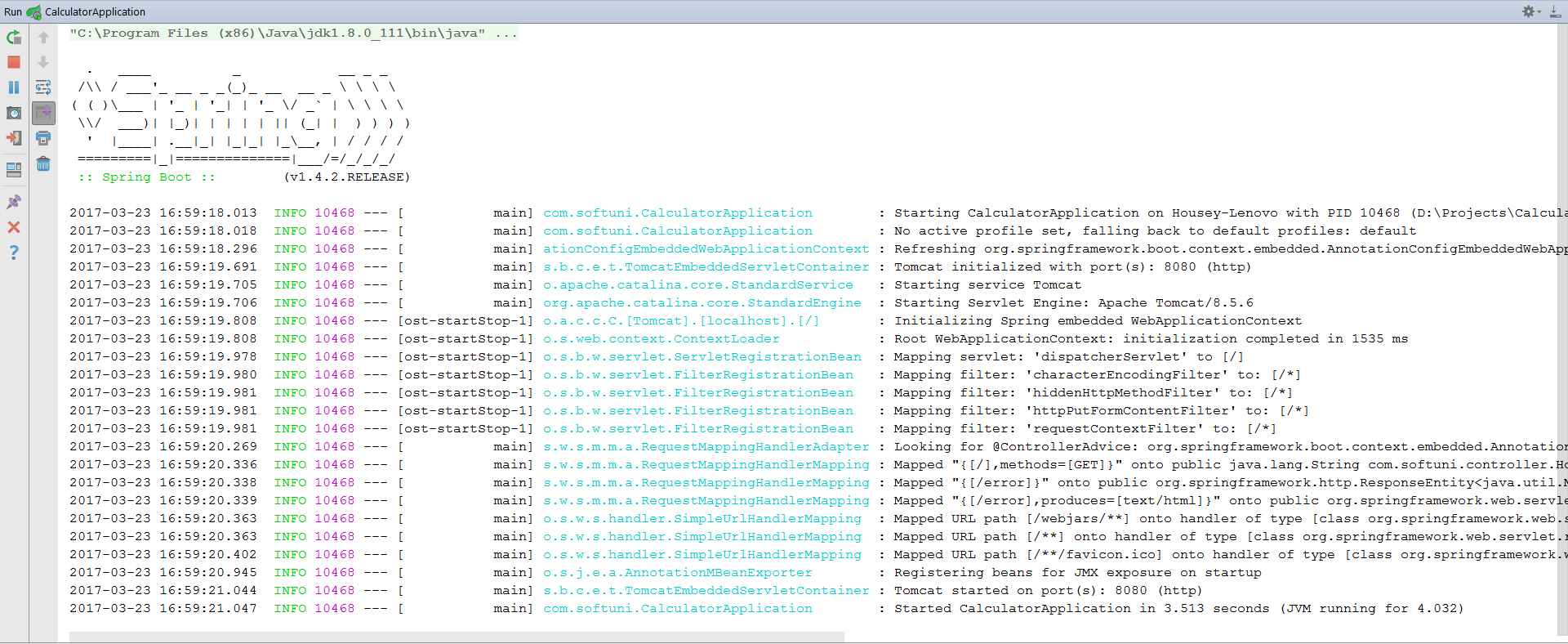
1. **src –** contains all the source files of our applications, including templates, models (entities) and controllers.
2. **src/main/resources/static** – everything that is in our static folder (files, images, stylesheets, JavaScript scripts, etc.) will be accessible by every user.
3. **src/main/java/{packageName}/… –** we’ll store all of our back-end logic (controllers, entities, etc.) here (in separate folders, of course).
4. **src/main/resources/templates** – we’ll store our **view templates** here. We’ll be using the template engine **Thymeleaf**.

## Run the Project

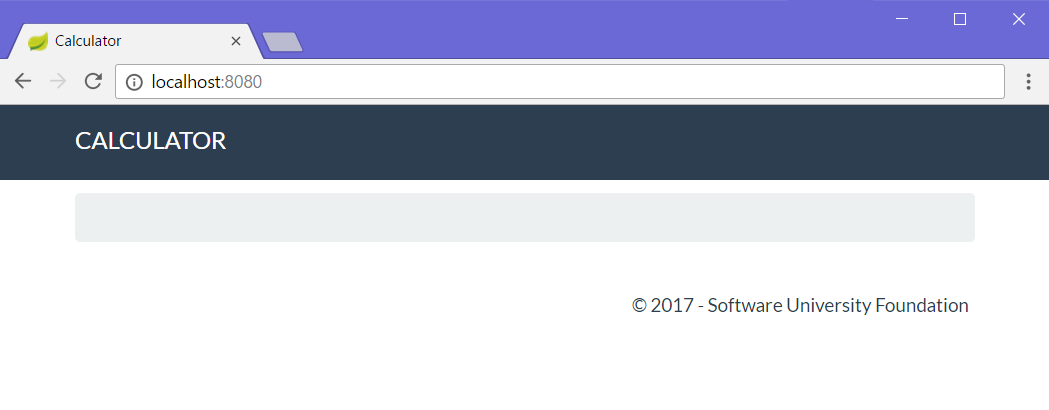
Now that we’ve opened the project, let’s try running it, so we can see what we’re working with. Go to the top right of IntelliJ Idea, where you’ll find a Run button, which looks like a green play button (be patient – the first build may take a while due to resolving dependencies:



That’s how we’ll run our Spring app. Go ahead and click it. If everything goes according to plan, we should see this message in the console:



Now we can open the page and see what we have:

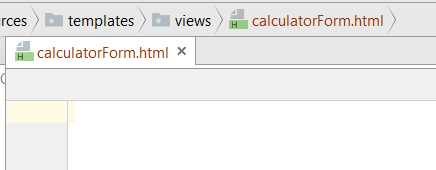
  
It doesn’t look like much, but at least we have the basic layout down! Let’s get to work on implementing some functionality!

# Implementing Functionality

## Create Calculator View

Before we can have any functionality, it would be nice to have an idea of what we’re working against, so let’s go ahead and **create** a **form**, which the **user** will use for **calculations**:

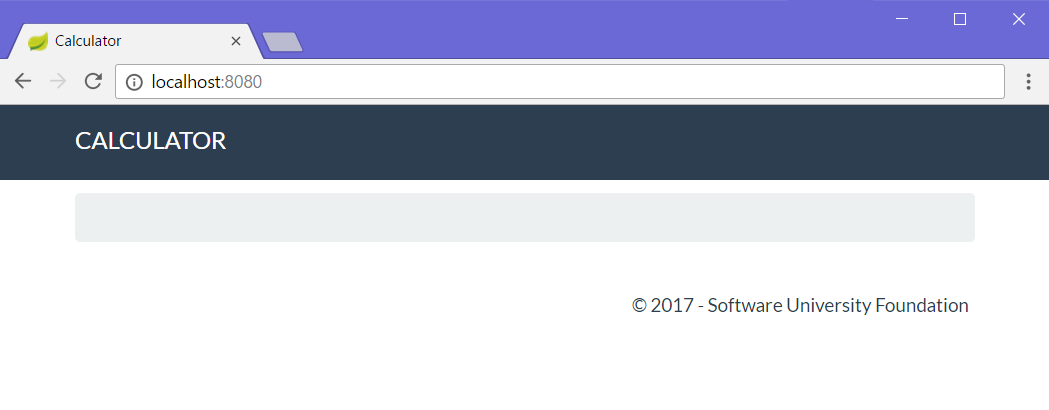
Go into the src/main/resources/templates/views/ folder and open the calculatorForm.html file:



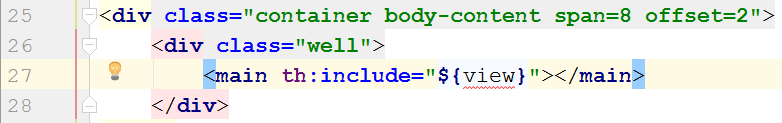
It’s empty?! How does the header and footer seen above get displayed then? The answer is, we use a global **layout** file (base-layout.html), so we don’t have to copy-paste our page layout into every single view in our project (which could have tens or hundreds of views). All the **actual base design HTML** is inside base-layout.html. We won’t be touching that, so let’s go to the calculatorForm.html file and add our form:

|  |
| --- |
| <**form class="form-inline" th:action="@{/}" method="POST"**>  <**fieldset**>  <**div class="form-group"**>  <**div class="col-sm-1 "**>  <**input type="text" class="form-control" id="leftOperand" placeholder="Left Operand"  name="leftOperand" th:value="${leftOperand}"**/>  </**div**>  </**div**>    <**div class="form-group"**>  <**div class="col-sm-4 "**>  <**select class="form-control" name="operator"**>  <**option value="+" th:selected="${operator.equals('+')}"**>+</**option**>  <**option value="-" th:selected="${operator.equals('-')}"**>-</**option**>  <**option value="\*" th:selected="${operator.equals('\*')}"**>\*</**option**>  <**option value="/" th:selected="${operator.equals('/')}"**>/</**option**>  </**select**>  </**div**>  </**div**>   <**div class="form-group"**>  <**div class="col-sm-4 "**>  <**input type="text" class="form-control" id="rightOperand" placeholder="Right Operand"  name="rightOperand" th:value="${rightOperand}"**/>  </**div**>  </**div**>   <**div class="form-group"**>  <**div class="col-sm-2 "**>  <**p**>=</**p**>  </**div**>  </**div**>   <**div class="form-group"**>  <**div class="col-sm-4 "**>  <**input type="text" class="form-control" id="result" placeholder="Result"  name="result" th:value="${result}"**/>  </**div**>  </**div**>   <**div class="form-group"**>  <**button type="submit" class="btn btn-primary"**>Calculate</**button**>  </**div**>  </**fieldset**> </**form**> |

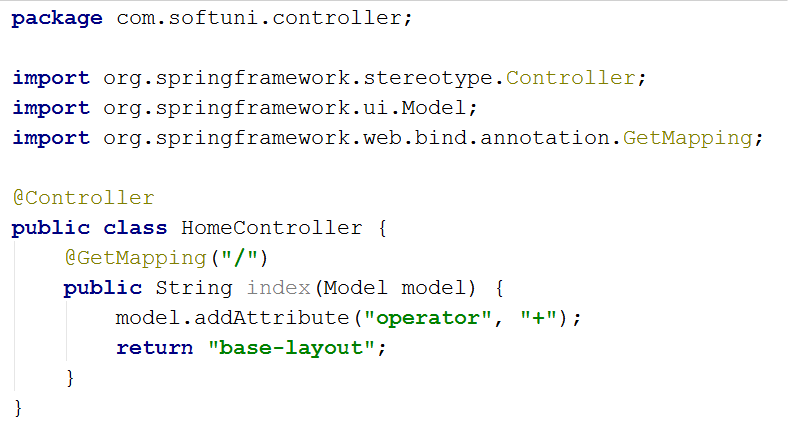
Just like with the JavaScript blog, we will **save the state** of the operands and operator for ease of use, so the **thymeleaf syntax** you see here does just that. The th:selected="" template is a bit more special: it selects the operator from the dropdown list, **based on** the last used operator. We’ll see how that’s implemented a bit later. For now, let’s navigate to our web app at <http://localhost:8080> and see how we’re doing:



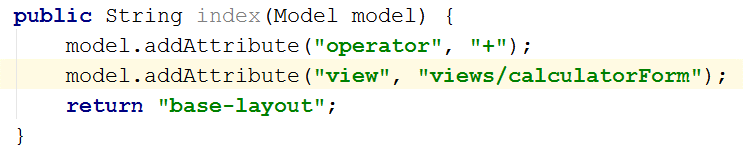
Still nothing?! The reason our form doesn’t display is because we don’t have the form as-is. In order to add the form to our project, we need to do two things: The first thing is to add this line of code in base-layout.html:



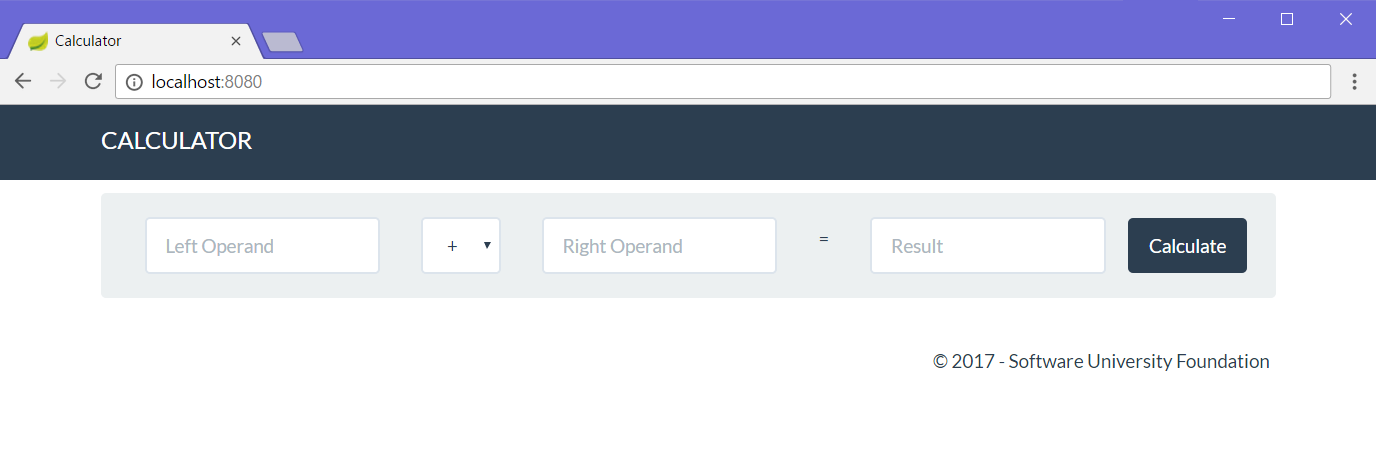
This line of code expects to be fed a **template** to display around the header and footer. So, we’re going to do just that. Go into the **HomeController.java** file and check out what the **index** action does:



All this action does is listen on the “**/**” (site root) route, and display the **base-layout** view. But before it displays it, it **adds an attribute** to the model. The model is the data that’s fed to the view, so it can perform various functions (such as displaying an article’s title, content, etc.). We’re going to give the **view** one more **attribute**. Remember that ${view} template the **base-layout** expects? Let’s **add** it here **before** returning the base layout view:



We’ve given the view over, so let’s get **restart** our spring application and see how we’re doing:



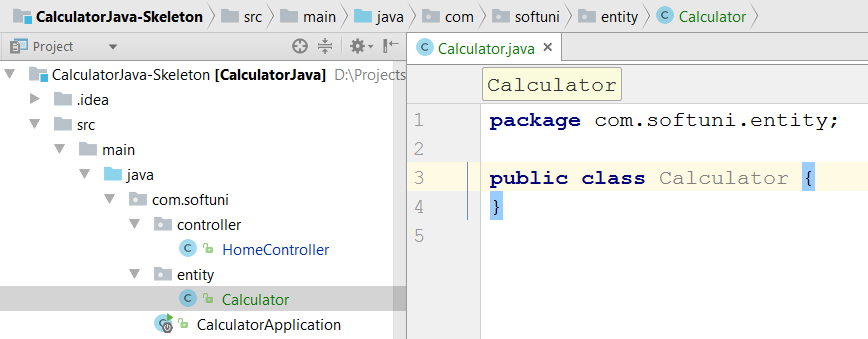
Looking good! Except it doesn’t do anything. First, let’s get down to making the thing, which will hold our data: the **model**.

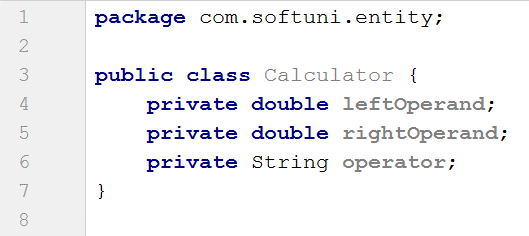
## Create Calculator Model

It’s time to design our main model – the **Calculator**. It will contain the following properties:

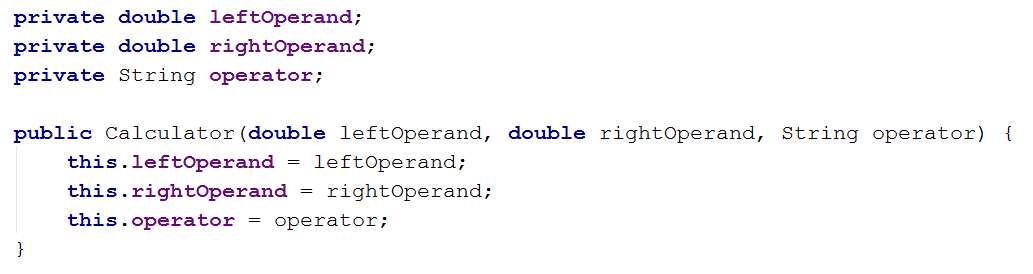
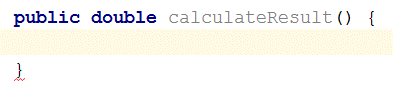
* leftOperand
* operator
* rightOperand

Let’s create our mode. Since we’re **not** using a database in this exercise, we’re just going to define the calculator as a **simple Java class** (the only difference between Java classes and Hibernate models is that Hibernate models have annotations, which help it name database columns and set restrictions). Go into the **entity** folder and create a new Java class, called “**Calculator.java**”:



1. Create **fields**, which will be used internally in the class:  
   
2. **Define** the calculator **properties**:

|  |  |  |
| --- | --- | --- |
|  | 🡪 |  |

1. Create a **constructor** for **instantiating** the calculator:  
   
2. Create a **method** for **calculating** the result from the **properties**:  
     
   Inside this method, we’ll write the logic, which is needed for calculating the result from the operands and operator. Let’s create the logic, needed for that.
3. Write the calculation logic:  
   

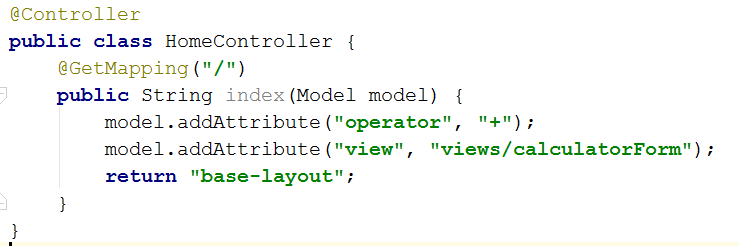
Our calculator logic is neatly nested inside the Calculator class. Now all that’s left is to connect it to the rest of our little web application.

For our final trick, we’ll create our own controller action, which will **process** what the user sent us and **return** a **view** with the **result** from the calculation.

## Implement the Controller Action

Now that we’ve created the **view**, which will **hold our data** and allow the **user** to **interact** with our web application, it’s time to implement the driving force behind the whole app – **the controller action**.

As it turns out, we already have a **home controller** set up, and an action, set up on the “**/**” route, otherwise we wouldn’t even be able to see our calculator. You can find the **home controller** in the “**controller**„ folder. Let’s see what it looks like:



Not much going on here… Let’s break it down:

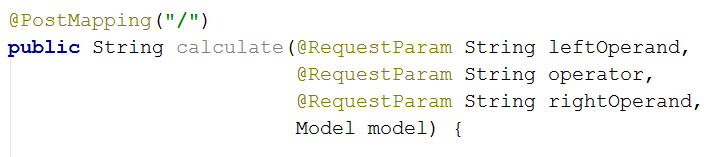
* @GetMapping("/") 🡪 the piece of code, which binds a URL Route to a method, so it can **execute the action** when our user **calls** the specified **route** (right now, that’s “**/**”).
* public String index(Model model) 🡪 This is the actual **controller action**. It’s a method, which **holds the** **logic**, which will be **executed**, when it’s **called**.   
  It has one parameter: model – it holds the data, which will be passed to the **view** for processing. Remember – all we’re doing here is returning different HTML code, based on the logic we’ve implemented in our app.
* model.addAttribute(String, String) 🡪 this piece of code takes **2 parameters** – the **first** states the **name** of the **attribute** we’re going to **send** to the **view**. The **second** one – the **actual data**.
* return "base-layout" 🡪 This function **renders** a **layout** in the **response** (in essence, takes whatever’s inside of “templates/base-layout.html”, runs it through the Thymeleaf templating engine, and returns it to the user.

So, using that newfound knowledge, let’s try to create our own **action**.

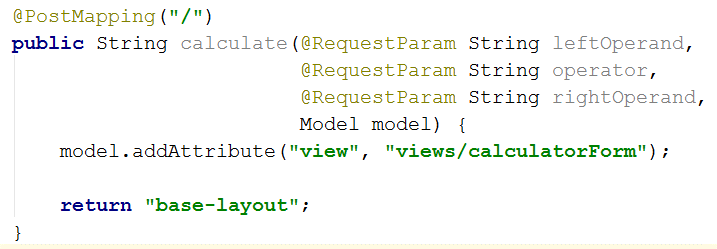
First, we’ll start off by declaring what kind of **HTTP method** this method will be handling (either GET or POST). In our case, since we’re processing **form data** sent to the “**/**” URL, we’ll set it to @PostMapping("/"):



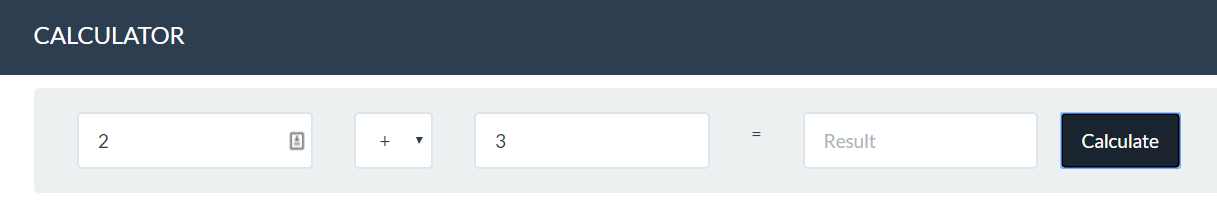
After that, let’s declare our method. We’ll use a couple of new types here – RequestParam. That’s just a fancy way of getting the form data:

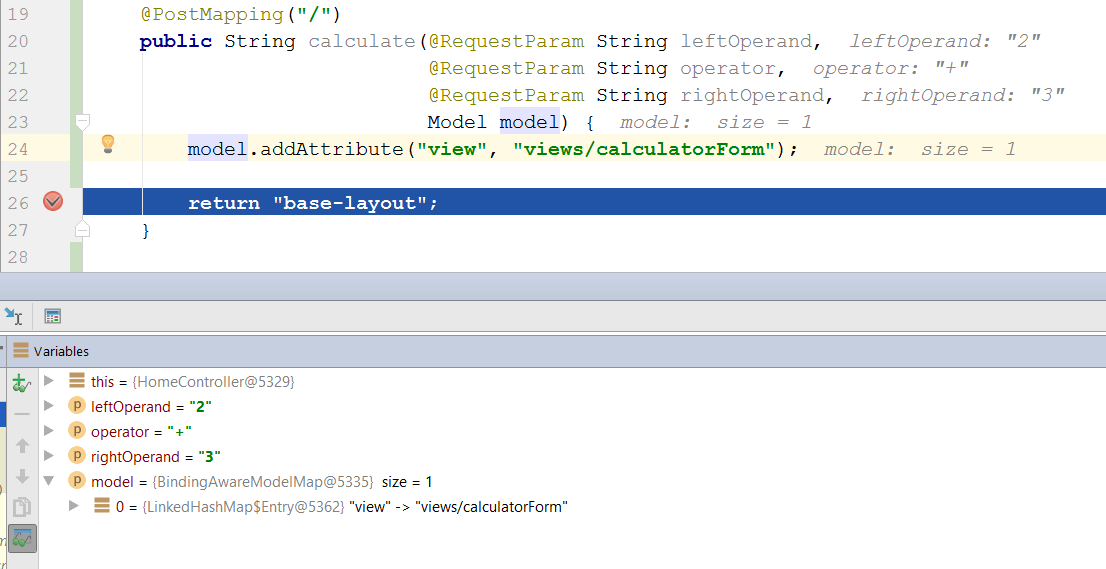


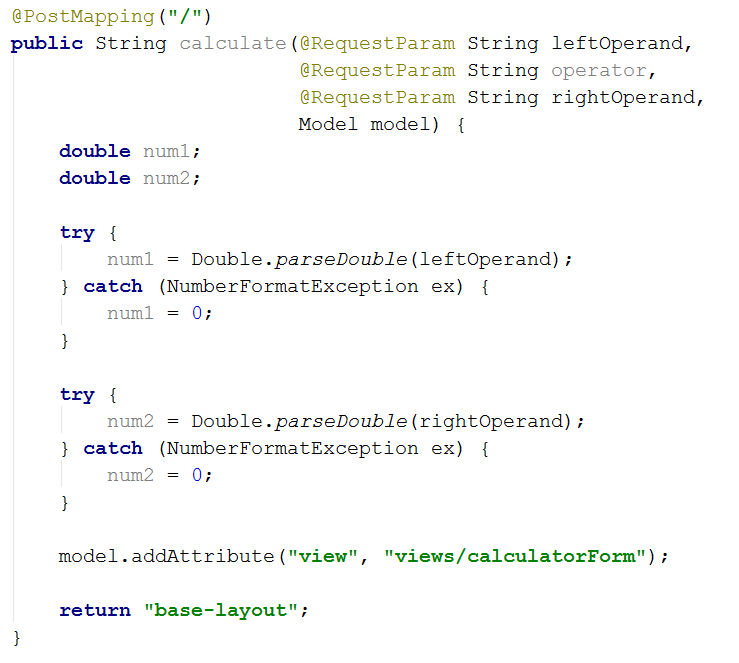
All this method should do at this point is just return the base-layout template with the **form attribute** inside of it:



Let’s see what a debug session would show us if we were to debug this method:

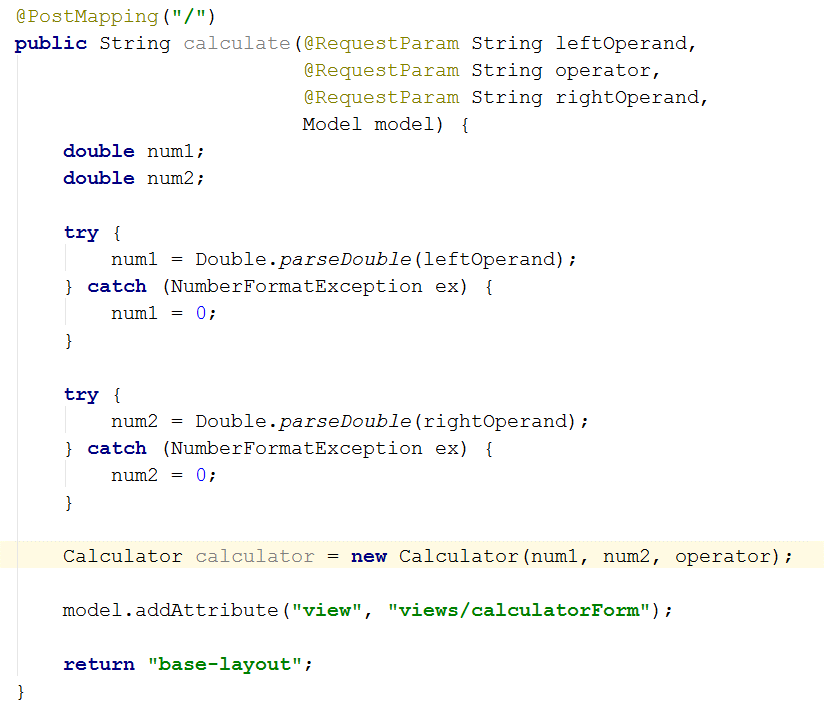


  
The leftOperand, operator, and rightOperand variables come in the form of **strings**. So, before we feed them to our **calculator** class, we need to **parse** them as numeric variables, using **try-catch** blocks:

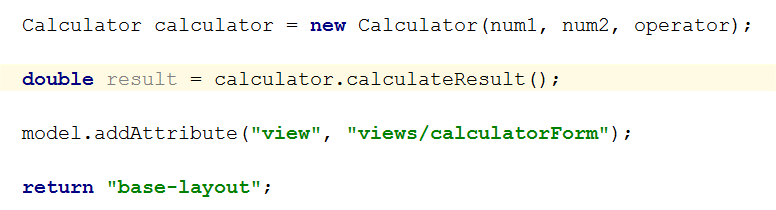


This will ensure, that if the user tries to send us **invalid data**, it’ll be set to **zero** instead.

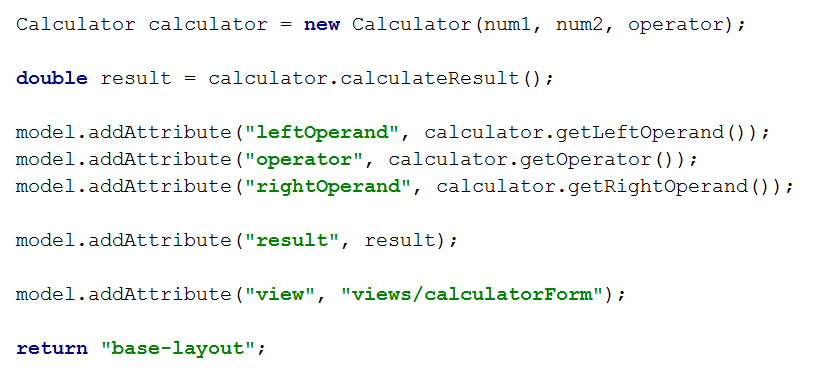
Next, we need to create an **instance** of our calculator model, which we’ll use for storing the data inside:



We can set its data, using its constructor, which we defined in step 4. Now that we’ve gotten the data, it’s time to calculate the result from what we currently have. Remember that calculateResult() function we wrote a while ago? Now is the time to use it:

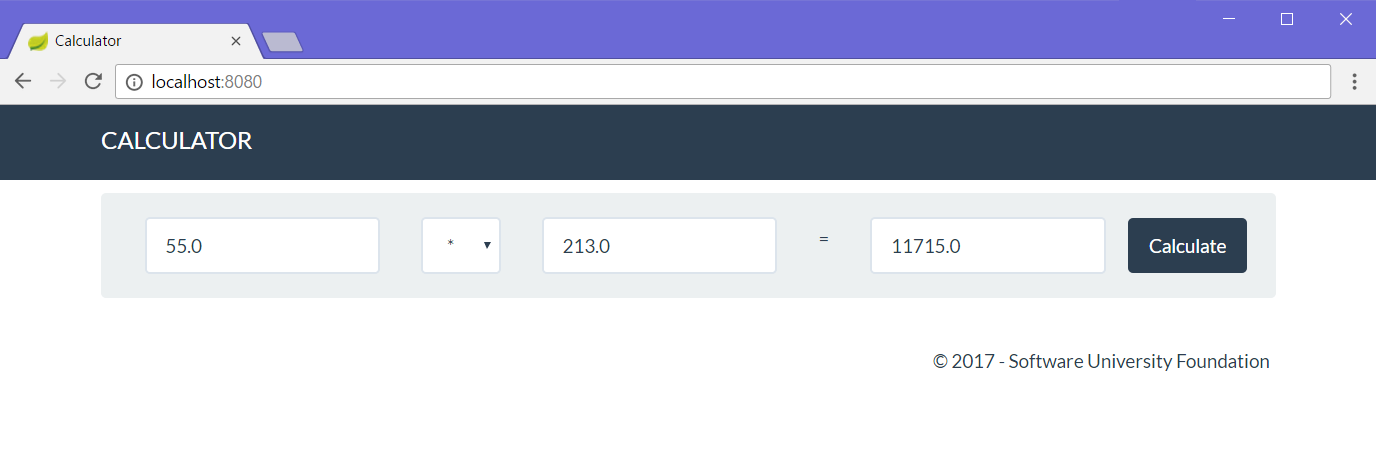


After that, all we have left is to **send the result to the** **view**. Apart from the result, we also need to send the leftOperand, rightOperand and operator, so we can **save the state** of our calculator through requests. We can do that, using the model.addAttribute function:

  
This way, we specify what we’re going to **send** to **the view**. So, when we send over the **result value** and the previous state of the calculator to the view, we can **fill** the **form fields** with our data. This happens here:  
  
We use the data from the controller in the views/calculatorForm.html view to set the **values** of the form inputs to whatever we want. In this case, we set the **operands**, and select the last used **operator**.

# Test the Application

All our hard work should finally pay off now, right? If you’ve followed all the steps properly, and have read all the explanatory text, hopefully we should have a functioning calculator!



# \* Implement Extra Functionality

Just like last time, you’re free to implement extra functionality like **extra operators**, **input validation**, and whatever else you can think of. Happy coding. ☺

Next time we’ll be using the same logic as in this lab to implement a **fully functioning blog system**, with a **database** behind it for storing everything and even **user authentication** and **security**. Have fun with Java! ☺