Calculator: Java and Spring

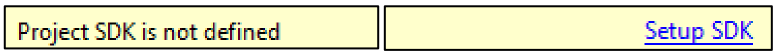
This document defines a complete walkthrough of creating a **Calculator** application with the Spring Framework, from setting up the framework to implementing the fully functional application.

I. Setting Up JDK and IntelliJ Idea Configuration

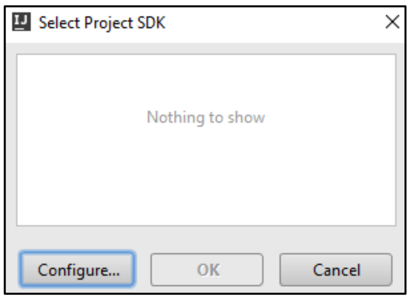
Before we start, you need to download the Java Development Kit, also known as **JDK**. Download the "**Java SE Development Kit 8u141**". After downloading it, install it **without changing the installation directory**. The default configuration will install it in the "**Program Files\Java**" folder if you're running **Windows**.

1. Set Up JDK

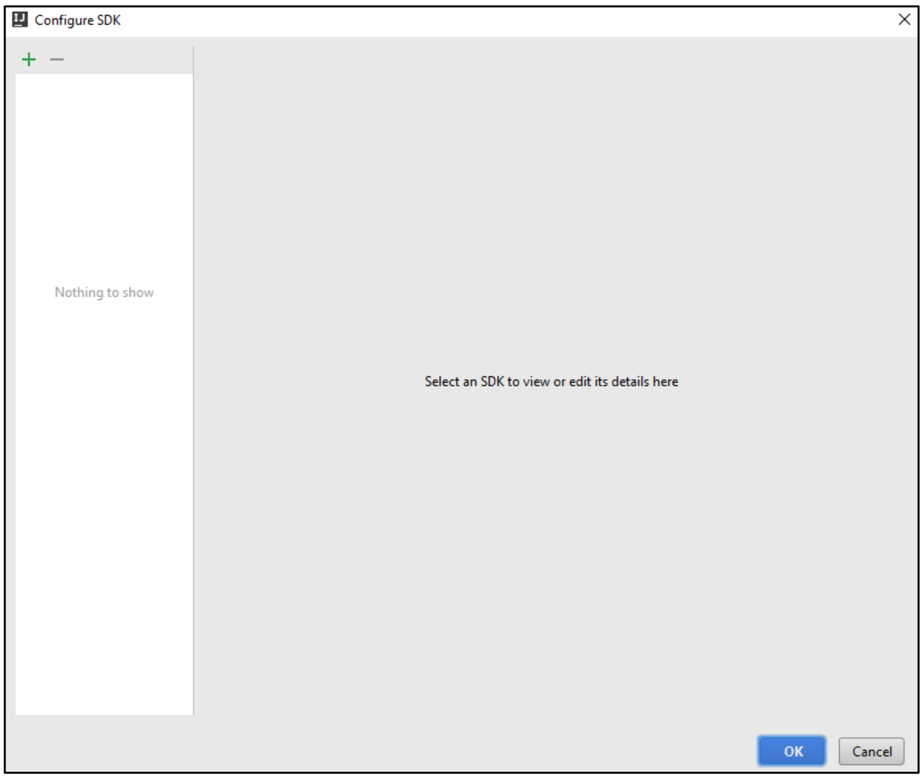
If you are using the skeleton and see something like this:



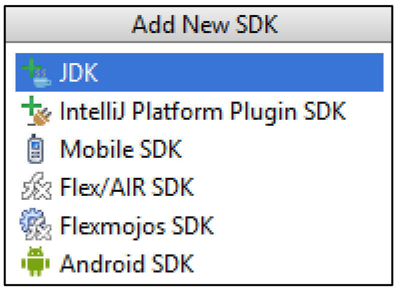
You should set-up the SDK. Click on "**Setup SDK**". You should see this screen:



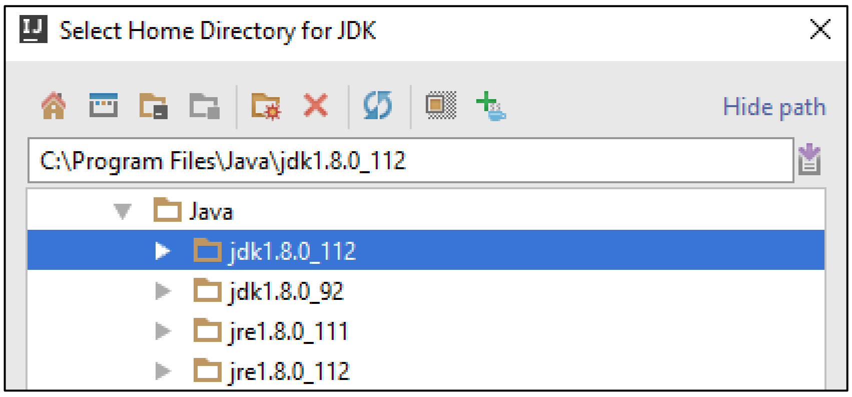
Click on "**Configure**" and see if you receive this screen:



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

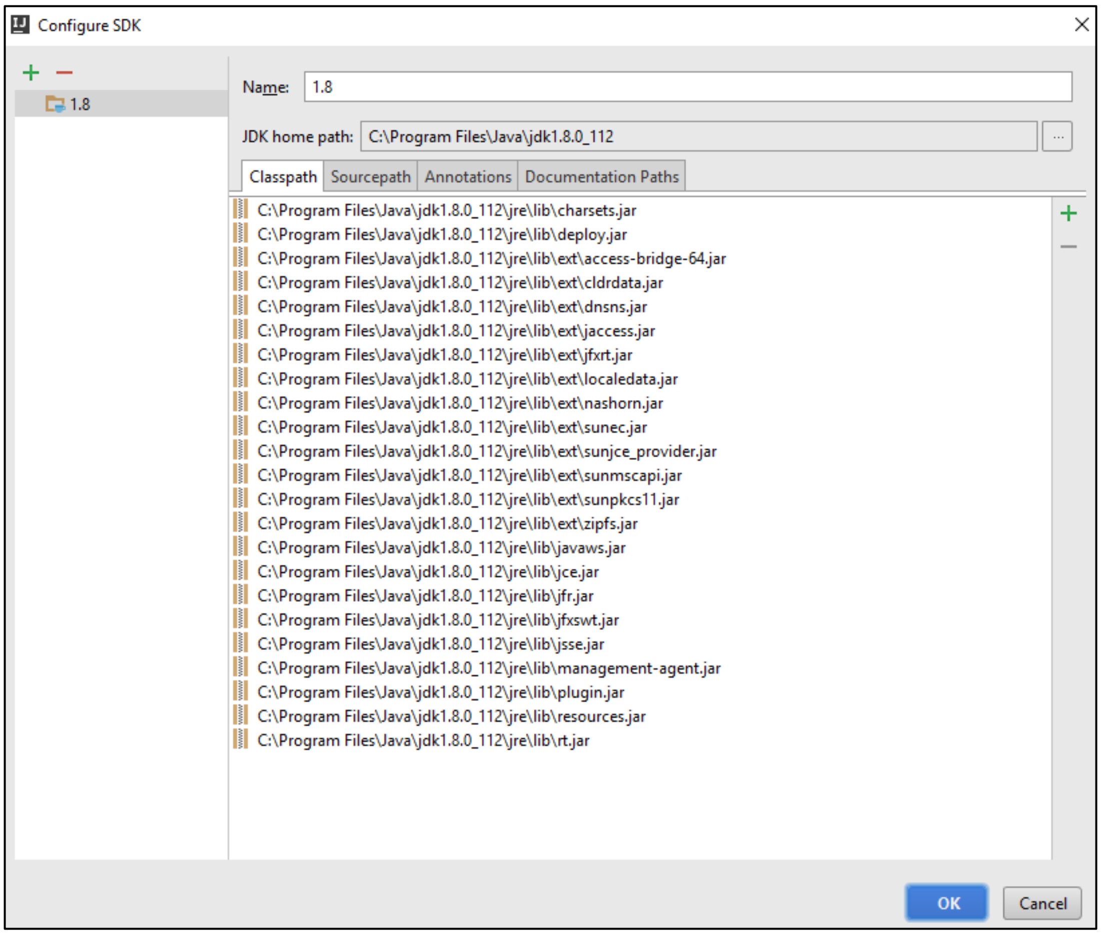


Then **locate your JDK**, it should be in the "**Program Files**" **folder** if you're using **windows**:



C:\Program Files\Java\jdk1.8.0_112

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



Name: 1.8

JDK home path: C:\Program Files\Java\jdk1.8.0_112

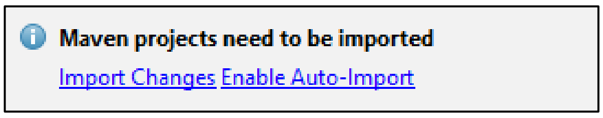
```
C:\Program Files\Java\jdk1.8.0_112\jre\lib\charsets.jar
C:\Program Files\Java\jdk1.8.0_112\jre\lib\deploy.jar
C:\Program Files\Java\jdk1.8.0_112\jre\lib\ext\access-bridge-64.jar
C:\Program Files\Java\jdk1.8.0_112\jre\lib\ext\cldrdata.jar
C:\Program Files\Java\jdk1.8.0_112\jre\lib\ext\dnsns.jar
C:\Program Files\Java\jdk1.8.0_112\jre\lib\ext\jaccess.jar
C:\Program Files\Java\jdk1.8.0_112\jre\lib\ext\jfxrt.jar
C:\Program Files\Java\jdk1.8.0_112\jre\lib\ext\localedata.jar
C:\Program Files\Java\jdk1.8.0_112\jre\lib\ext\nashorn.jar
C:\Program Files\Java\jdk1.8.0_112\jre\lib\ext\sunec.jar
C:\Program Files\Java\jdk1.8.0_112\jre\lib\ext\sunjce_provider.jar
C:\Program Files\Java\jdk1.8.0_112\jre\lib\ext\sunmscapi.jar
C:\Program Files\Java\jdk1.8.0_112\jre\lib\ext\sunpkcs11.jar
C:\Program Files\Java\jdk1.8.0_112\jre\lib\ext\zipfs.jar
C:\Program Files\Java\jdk1.8.0_112\jre\lib\javaws.jar
C:\Program Files\Java\jdk1.8.0_112\jre\lib\jce.jar
C:\Program Files\Java\jdk1.8.0_112\jre\lib\jfr.jar
C:\Program Files\Java\jdk1.8.0_112\jre\lib\jfxswt.jar
C:\Program Files\Java\jdk1.8.0_112\jre\lib\jsse.jar
C:\Program Files\Java\jdk1.8.0_112\jre\lib\management-agent.jar
C:\Program Files\Java\jdk1.8.0_112\jre\lib\plugin.jar
C:\Program Files\Java\jdk1.8.0_112\jre\lib\resources.jar
C:\Program Files\Java\jdk1.8.0_112\jre\lib\rt.jar
```

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

II. Base Project Overview

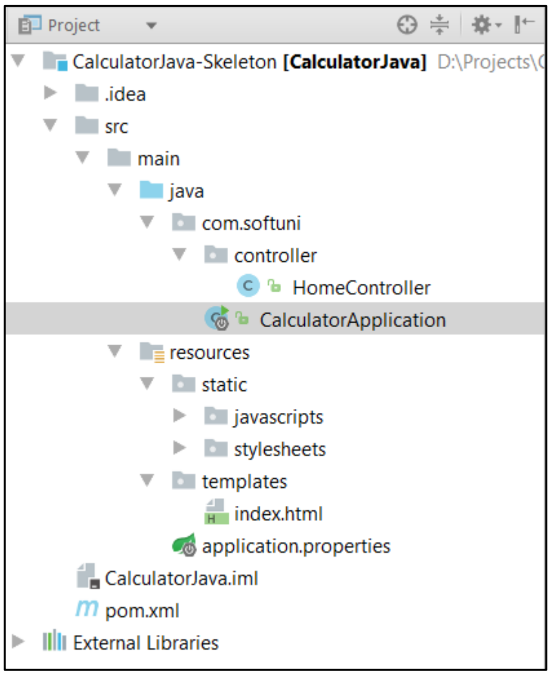
2. Open the Project

When you open the project you might see the following window in the lower right corner of Intllij:


Maven projects need to be imported
Import Changes Enable Auto-Import

Click on "**Enable Auto-Import**". It is **really important** and if you miss this step, the **project might not work** as **you would expect**.

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

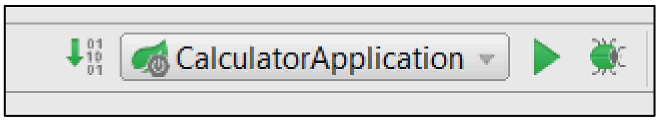


We can see several folders here. Let look at them one by one and see what they are for:

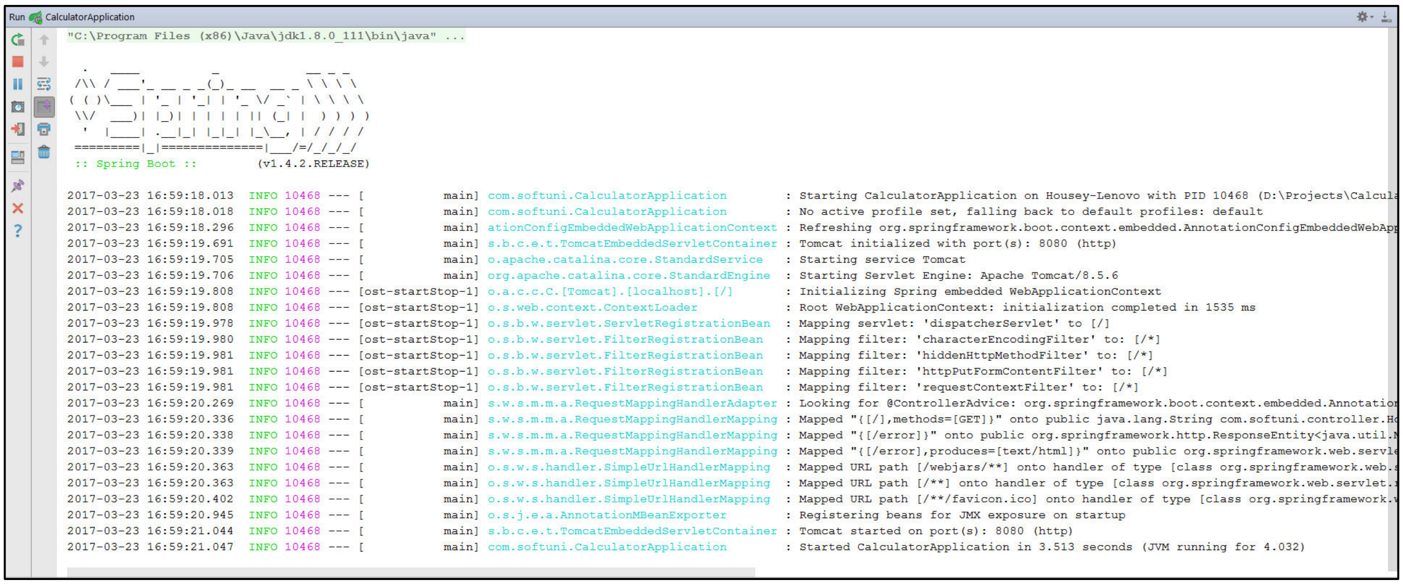
1. `src` – Contains all the **source** files of our applications, including **templates**, **models** (entities) and **controllers**.
2. `src/main/resources/static` – Everything that's in our **static folder** (files, images, stylesheets, JavaScript scripts, etc.) will be **accessible** by **every user**.
3. `src/main/java/{packageName}/…` **–** we'll store all 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**.

3. 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** corner 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):

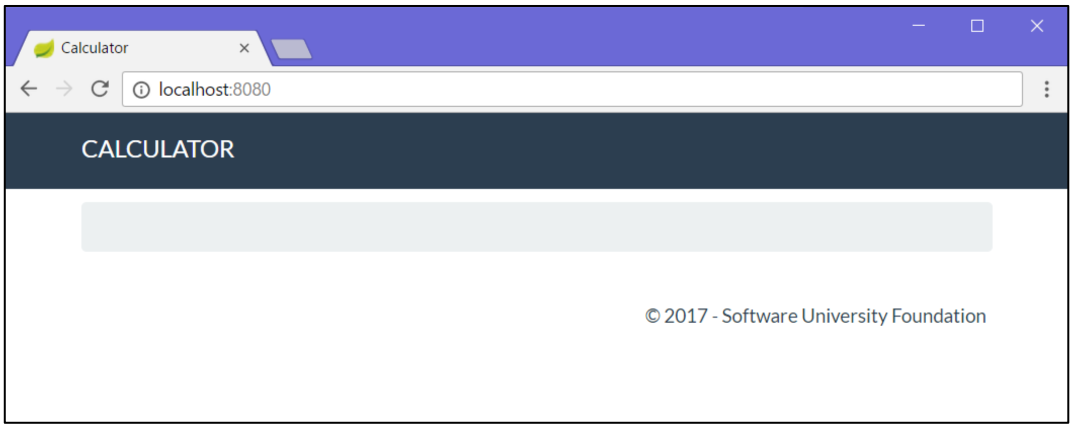


That's how we'll run our Spring app. Go ahead and click the **play button**. If everything goes according to plan, we should see this message in the console:



```
"C:\Program Files (x86)\Java\jdk1.8.0_111\bin\java" ...

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v1.4.2.RELEASE)

2017-03-23 16:59:18.013  INFO 10468 --- [           main] com.softuni.CalculatorApplication        : Starting CalculatorApplication on Housey-Lenovo with PID 10468 (D:\Projects\Calcula
2017-03-23 16:59:18.018  INFO 10468 --- [           main] com.softuni.CalculatorApplication        : No active profile set, falling back to default profiles: default
2017-03-23 16:59:18.296  INFO 10468 --- [           main] ationConfigEmbeddedWebApplicationContext : Refreshing org.springframework.boot.context.embedded.AnnotationConfigEmbeddedWebApp
2017-03-23 16:59:19.691  INFO 10468 --- [           main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat initialized with port(s): 8080 (http)
2017-03-23 16:59:19.705  INFO 10468 --- [           main] o.apache.catalina.core.StandardService   : Starting service Tomcat
2017-03-23 16:59:19.706  INFO 10468 --- [           main] org.apache.catalina.core.StandardEngine  : Starting Servlet Engine: Apache Tomcat/8.5.6
2017-03-23 16:59:19.808  INFO 10468 --- [ost-startStop-1] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebApplicationContext
2017-03-23 16:59:19.808  INFO 10468 --- [ost-startStop-1] o.s.web.context.ContextLoader            : Root WebApplicationContext: initialization completed in 1535 ms
2017-03-23 16:59:19.978  INFO 10468 --- [ost-startStop-1] o.s.b.w.servlet.ServletRegistrationBean  : Mapping servlet: 'dispatcherServlet' to [/]
2017-03-23 16:59:19.980  INFO 10468 --- [ost-startStop-1] o.s.b.w.servlet.FilterRegistrationBean   : Mapping filter: 'characterEncodingFilter' to: [/*]
2017-03-23 16:59:19.981  INFO 10468 --- [ost-startStop-1] o.s.b.w.servlet.FilterRegistrationBean   : Mapping filter: 'hiddenHttpMethodFilter' to: [/*]
2017-03-23 16:59:19.981  INFO 10468 --- [ost-startStop-1] o.s.b.w.servlet.FilterRegistrationBean   : Mapping filter: 'httpPutFormContentFilter' to: [/*]
2017-03-23 16:59:19.981  INFO 10468 --- [ost-startStop-1] o.s.b.w.servlet.FilterRegistrationBean   : Mapping filter: 'requestContextFilter' to: [/*]
2017-03-23 16:59:20.269  INFO 10468 --- [           main] s.w.s.m.m.a.RequestMappingHandlerAdapter : Looking for @ControllerAdvice: org.springframework.boot.context.embedded.Annotatio
2017-03-23 16:59:20.336  INFO 10468 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "{[/],methods=[GET]}" onto public java.lang.String com.softuni.controller.Ho
2017-03-23 16:59:20.338  INFO 10468 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "{[/error]}" onto public org.springframework.http.ResponseEntity<java.util.M
2017-03-23 16:59:20.339  INFO 10468 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "{[/error],produces=[text/html]}" onto public org.springframework.web.servle
2017-03-23 16:59:20.363  INFO 10468 --- [           main] o.s.w.s.handler.SimpleUrlHandlerMapping  : Mapped URL path [/webjars/**] onto handler of type [class org.springframework.web.s
2017-03-23 16:59:20.363  INFO 10468 --- [           main] o.s.w.s.handler.SimpleUrlHandlerMapping  : Mapped URL path [/**] onto handler of type [class org.springframework.web.servlet.r
2017-03-23 16:59:20.402  INFO 10468 --- [           main] o.s.w.s.handler.SimpleUrlHandlerMapping  : Mapped URL path [/**/favicon.ico] onto handler of type [class org.springframework.w
2017-03-23 16:59:20.945  INFO 10468 --- [           main] o.s.j.e.a.AnnotationMBeanExporter        : Registering beans for JMX exposure on startup
2017-03-23 16:59:21.044  INFO 10468 --- [           main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8080 (http)
2017-03-23 16:59:21.047  INFO 10468 --- [           main] com.softuni.CalculatorApplication        : Started CalculatorApplication in 3.513 seconds (JVM running for 4.032)
```

Now we can open the page (at **localhost:8080**) and see what we have:



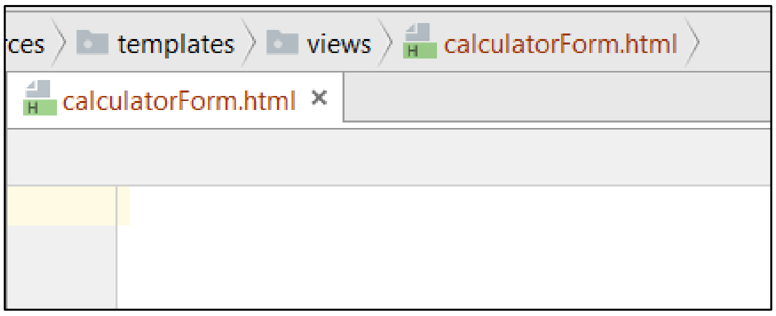
It doesn't look like much, but at least we have the basic layout down! Let's get to work on implementing some functionality!

III. Implementing Functionality

4. Create Calculator View

Before we start adding 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**:

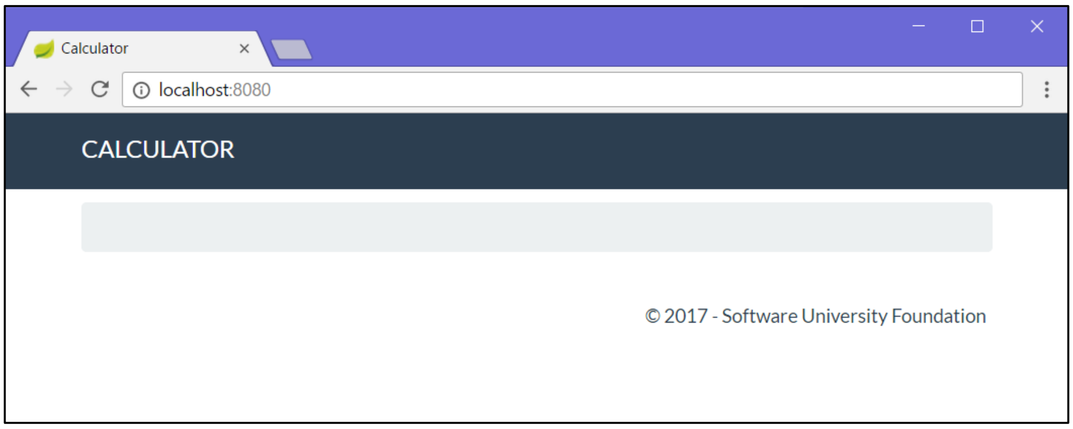
Go into the **`src/main/resources/templates/views/`** folder and open the **`calculatorForm.html`** file:



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 **automatically 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:



© 2017 - Software University Foundation

Still nothing?! The reason our form doesn't display is because we're not sending the **form view** to the user. 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`**:

```
<div class="container body-content span=8 offset=2">
    <div class="well">
        <main th:include="${view}"></main>
    </div>
```

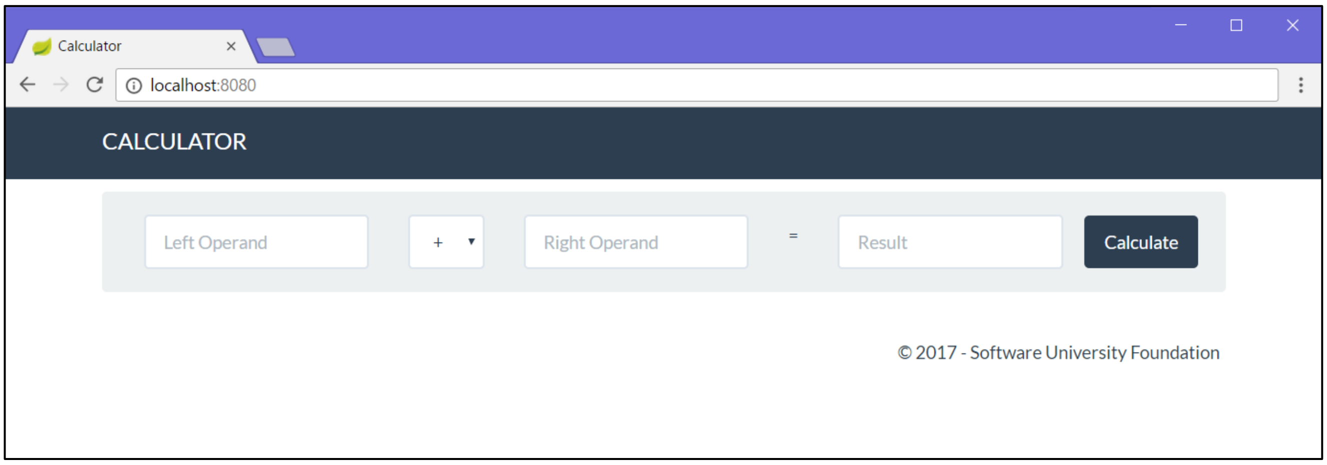
This line of code expects to be fed a **template** to display around the header and footer. So, we're going to do just that. Go into the **`HomeController.java`** file and check out what the **index** action does:

```
package com.softuni.controller;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;

@Controller
public class HomeController {
    @GetMapping("/")
    public String index(Model model) {
        model.addAttribute("operator", "+");
        return "base-layout";
    }
}
```

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 the calculator's operands, operator, 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:

```
public String index(Model model) {
    model.addAttribute("operator", "+");
    model.addAttribute("view", "views/calculatorForm");
    return "base-layout";
}
```

We've given it the view, 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**.

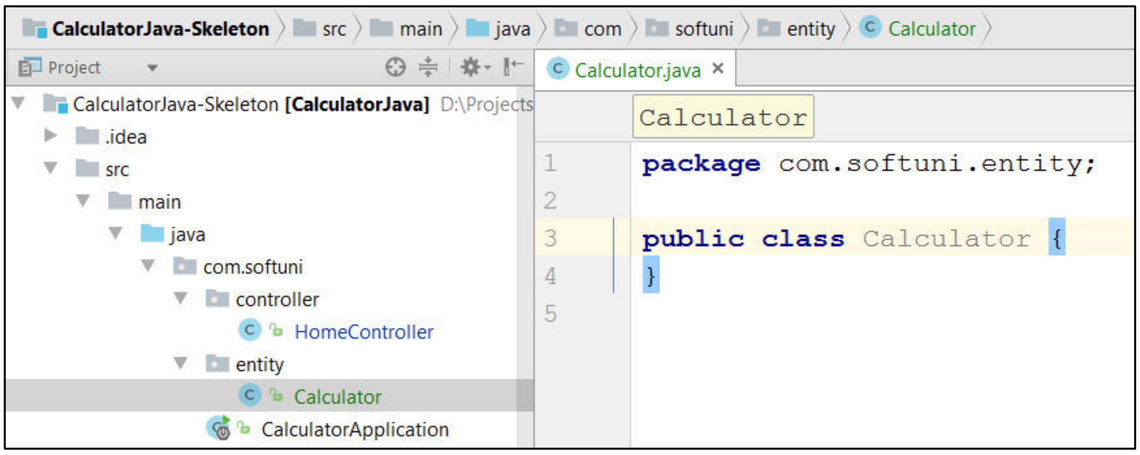
5. Create Calculator Model

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**.

Go into the **`entity`** folder and create a new Java class, called "**Calculator.java**":


CalculatorJava-Skeleton > src > main > java > com > softuni > entity > Calculator
Project
CalculatorJava-Skeleton [CalculatorJava] D:\Projects
.idea
src
main
java
com.softuni
controller
HomeController
entity
Calculator
CalculatorApplication
Calculator.java

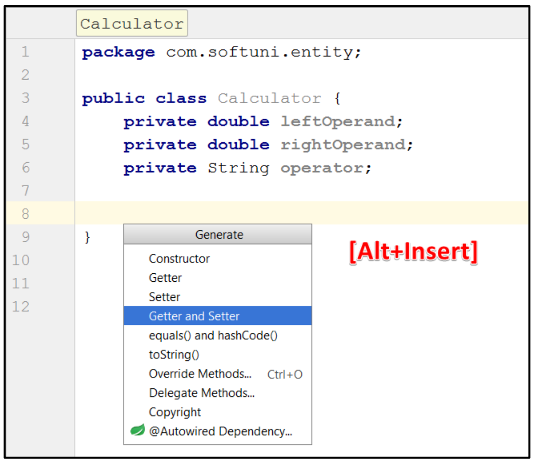
```java
package com.softuni.entity;

public class Calculator {
}
```

Now, let's define what our class will have:

1. Create **fields**, which will be used internally in the class:

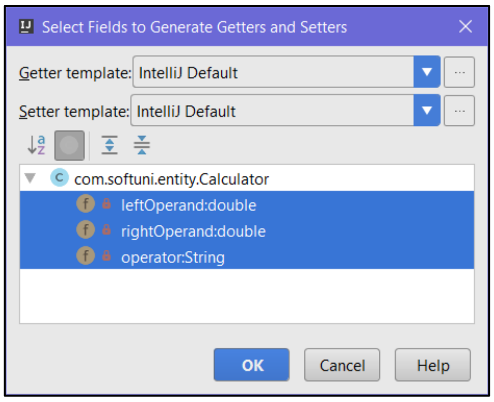
```java
package com.softuni.entity;

public class Calculator {
    private double leftOperand;
    private double rightOperand;
    private String operator;
}

```

2. **Define** the calculator **properties**:



```
package com.softuni.entity;

public class Calculator {
    private double leftOperand;
    private double rightOperand;
    private String operator;

}
```

→



Select Fields to Generate Getters and Setters

Getter template: IntelliJ Default

Setter template: IntelliJ Default

com.softuni.entity.Calculator
- leftOperand:double
- rightOperand:double
- operator:String

3. Create a **constructor** for **instantiating** the calculator:

```
private double leftOperand;
private double rightOperand;
private String operator;

public Calculator(double leftOperand, double rightOperand, String operator) {
    this.leftOperand = leftOperand;
    this.rightOperand = rightOperand;
    this.operator = operator;
}
```

4. Create **getters** and **setters** for the Calculator's **fields** (you can make them with **[Alt + Insert]**)
5. Create a **method** for **calculating** the result from the **properties**:

```
public double calculateResult() {

}
```

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.

6. Write the calculation logic:

```
public double calculateResult() {

    double result;

    switch (this.operator) {
        case "+":
            result  = this.getLeftOperand() + this.getRightOperand();
            break;
        [redacted]
        default:
            result = 0;
            break;
    }
    return result;
}
```

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.

6. 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:

```
@Controller
public class HomeController {
    @GetMapping("/")
    public String index(Model model) {
        model.addAttribute("operator", "+");
        model.addAttribute("view", "views/calculatorForm");
        return "base-layout";
    }
}
```

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**, 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("/")`**:

```
@PostMapping("/")
```

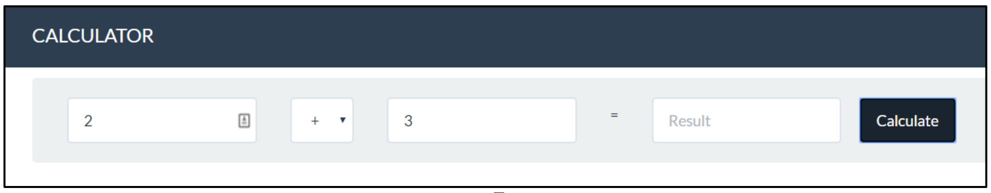
After that, let's declare our method. We'll use a couple of new types here – **`RequestParam`**. That's just a fancy way of getting the form data:

```
@PostMapping("/")
public String calculate(@RequestParam String leftOperand,
                        @RequestParam String operator,
                        @RequestParam String rightOperand,
                        Model model) {
```

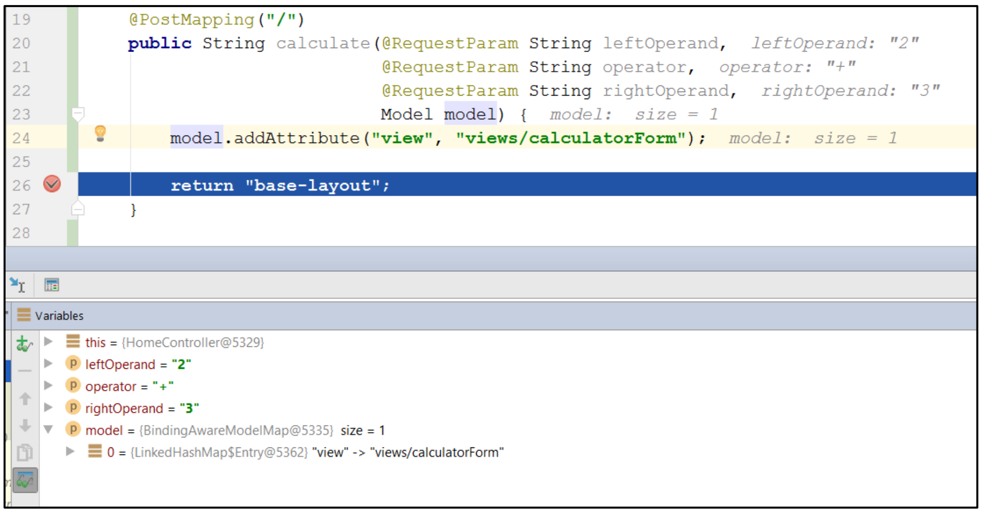
All this method should do at this point is just return the **`base-layout`** template with the **form attribute** inside of it:

```
@PostMapping("/")
public String calculate(@RequestParam String leftOperand,
                        @RequestParam String operator,
                        @RequestParam String rightOperand,
                        Model model) {
    model.addAttribute("view", "views/calculatorForm");

    return "base-layout";
}
```

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



⬇



```
@PostMapping("/")
public String calculate(@RequestParam String leftOperand,  leftOperand: "2"
                        @RequestParam String operator,  operator: "+"
                        @RequestParam String rightOperand,  rightOperand: "3"
                        Model model) {  model:  size = 1
    model.addAttribute("view", "views/calculatorForm");  model:  size = 1

    return "base-layout";
}
```

```
Variables
this = {HomeController@5329}
leftOperand = "2"
operator = "+"
rightOperand = "3"
model = {BindingAwareModelMap@5335}  size = 1
    0 = {LinkedHashMap$Entry@5362} "view" -> "views/calculatorForm"
```

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:

```
@PostMapping("/")
public String calculate(@RequestParam String leftOperand,
                        @RequestParam String operator,
                        @RequestParam String rightOperand,
                        Model model) {

    double num1;
    double num2;

    try {
        num1 = Double.parseDouble(leftOperand);
    } catch (NumberFormatException ex) {
        num1 = 0;
    }

    try {
        num2 = Double.parseDouble(rightOperand);
    } catch (NumberFormatException ex) {
        num2 = 0;
    }

    model.addAttribute("view", "views/calculatorForm");

    return "base-layout";
}
```

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:

```
@PostMapping("/")
public String calculate(@RequestParam String leftOperand,
                        @RequestParam String operator,
                        @RequestParam String rightOperand,
                        Model model) {
    double num1;
    double num2;

    try {
        num1 = Double.parseDouble(leftOperand);
    } catch (NumberFormatException ex) {
        num1 = 0;
    }

    try {
        num2 = Double.parseDouble(rightOperand);
    } catch (NumberFormatException ex) {
        num2 = 0;
    }

    Calculator calculator = new Calculator(num1, num2, operator);

    model.addAttribute("view", "views/calculatorForm");

    return "base-layout";
}
```

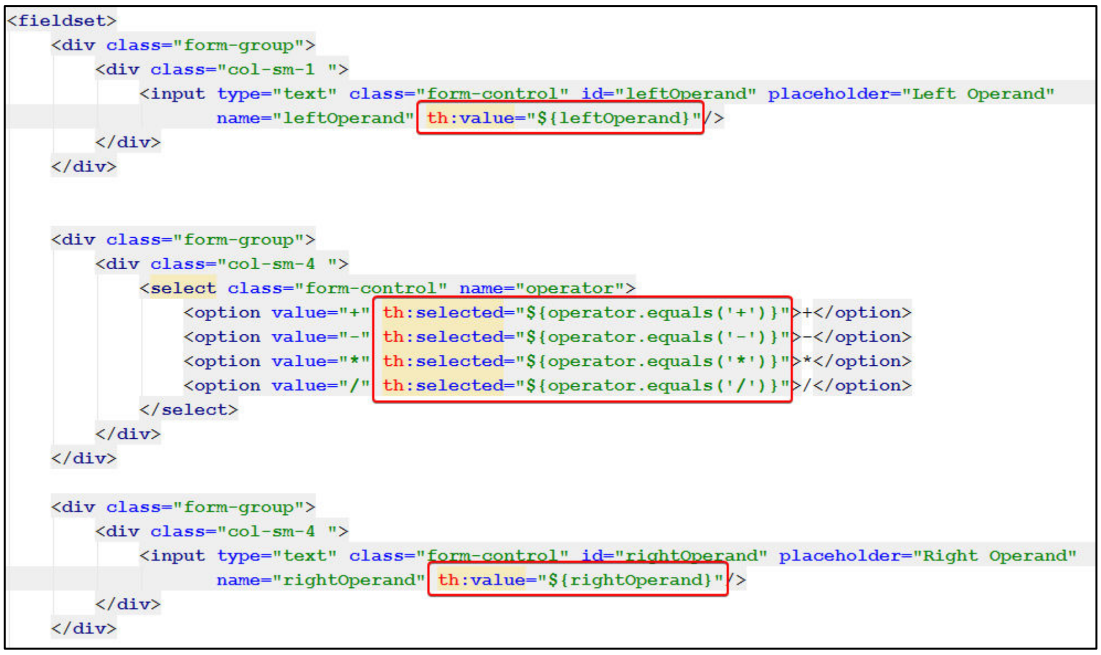
We can set its data, using its constructor, which we defined in step 4. Now that we've gotten the data, it's time to calculate the result from what we currently have. Remember that `calculateResult()` function we wrote a while ago? Now is the time to use it:

```
Calculator calculator = new Calculator(num1, num2, operator);

double result = calculator.calculateResult();

model.addAttribute("view", "views/calculatorForm");

return "base-layout";
```

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:

```
Calculator calculator = new Calculator(num1, num2, operator);

double result = calculator.calculateResult();

model.addAttribute("leftOperand", calculator.getLeftOperand());
model.addAttribute("operator", calculator.getOperator());
model.addAttribute("rightOperand", calculator.getRightOperand());

model.addAttribute("result", result);

model.addAttribute("view", "views/calculatorForm");

return "base-layout";
```

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:

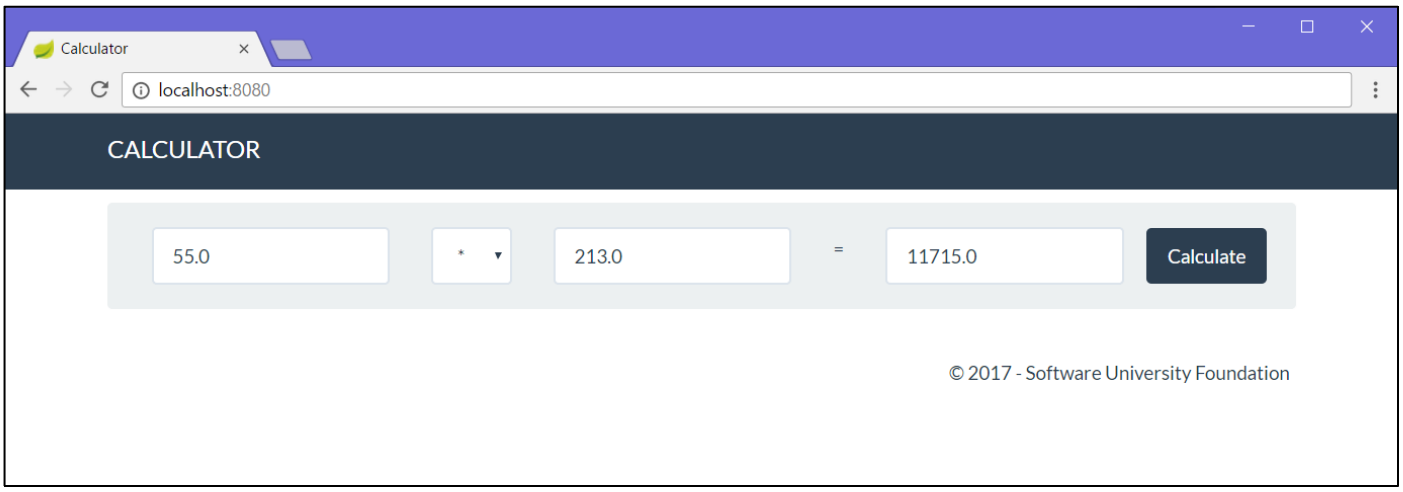


```
<fieldset>
    <div class="form-group">
        <div class="col-sm-1 ">
            <input type="text" class="form-control" id="leftOperand" placeholder="Left Operand"
                   name="leftOperand" th:value="${leftOperand}"/>
        </div>
    </div>


    <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>
```

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**.

IV. 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!



V. * 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! ☺