**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.Този документ дефинира пълен пътеводител за създаването на приложение за калкулатори с пролетната рамка, от създаването на рамката до внедряването на напълно функционалното приложение.

# I. Setting Up JDK and IntelliJ Idea Configuration

I. Създаване на конфигурация на идеи за JDK и IntelliJ

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 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**.

Преди да започнете, трябва да изтеглите комплекта за разработка на Java, известен също като JDK. Изтеглете "Java SE Development Kit 8u141". След като го изтеглите, инсталирайте го, без да променяте инсталационната директория. Конфигурацията по подразбиране ще я инсталира в папката "Program Files \ Java", ако работите с Windows.

1. Set Up JDK 1. Настройте JDK

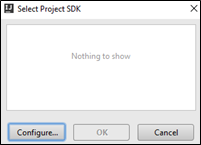
If you are using the skeleton and see something like this:

Ако използвате скелета и вижте нещо подобно:



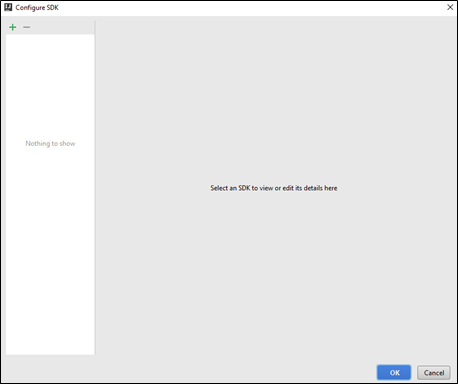
You should set-up the SDK. Click on "**Setup SDK**". You should see this screen:

Трябва да настроите SDK. Кликнете върху "Настройка SDK". Трябва да видите този екран:



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

Кликнете върху "Конфигуриране" и вижте дали ще получите този екран:

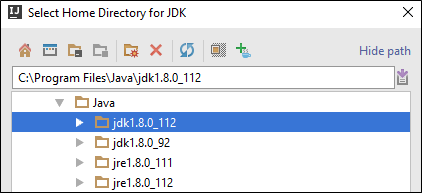


Click on the **green plus sign** in the top left corner of the window and choose **JDK**:

Кликнете върху зеления знак плюс в горния ляв ъгъл на прозореца и изберете JDK:

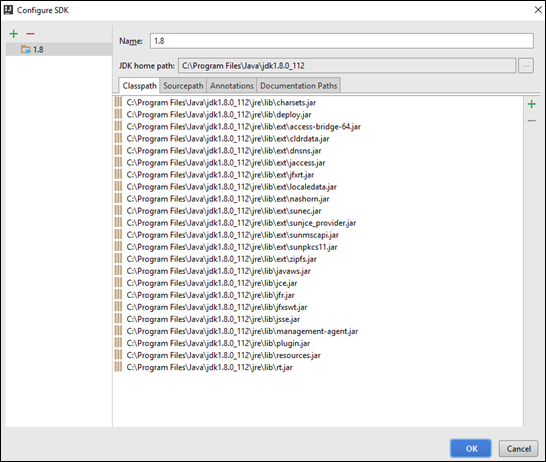
Then **locate your JDK**, it should be in the "**Program Files**" **folder** if you're using **windows**:

След това намерете JDK, той трябва да е в папката "Program Files", ако използвате Windows:



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

След като кликнете върху "OK", трябва да видите този екран:



That’s everything, your **JDK is now configured**. Това е всичко, вашият JDK вече е конфигуриран.

# I. Base Project Overview

## 1. Open the Project

When you open the [project](http://softuni.bg/downloads/svn/soft-tech/May-2017/Software-Technologies-July-2017/09.%20Software-Technologies-Java-Syntax-and-Basic-Web/09.%20Software-Technologies-Java-Syntax-Basic-Web-Calculator-Skeleton.zip) you might see the following window in the lower right corner of IntelliJ IDEA:

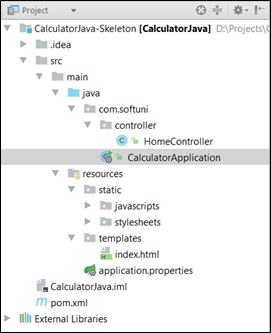
I. Основен преглед на проекта  
1. Отворете проекта  
Когато отворите проекта, може да видите следния прозорец в долния десен ъгъл на IntelliJ IDEA:



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](http://softuni.bg/downloads/svn/soft-tech/May-2017/Software-Technologies-July-2017/09.%20Software-Technologies-Java-Syntax-and-Basic-Web/09.%20Software-Technologies-Java-Syntax-Basic-Web-Calculator-Skeleton.zip), 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:

Кликнете върху "Активиране на автоматичното импортиране". Това е наистина важно и ако пропуснете тази стъпка, проектът може да не работи както бихте очаквали.  
В нашия проект има само една папка, от която се интересуваме. Това би било папката "src". Тази папка ще съдържа всички файлове, които ще създадем. Нека да разгледаме:



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

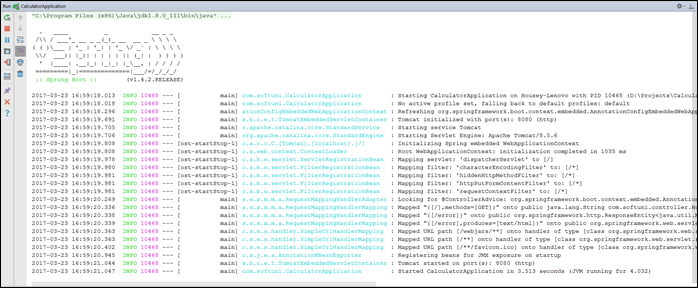
Now that we’ve opened the project, let’s try running it, so we can see what we’re working with. Go to the **top right** corner of **IntelliJ Idea**, where you’ll find a **Run** button, which looks like a **green play button** (be patient – the first build may take a while due to resolving dependencies):

Тук можем да видим няколко папки. Нека да ги разгледаме един по един и да видим за какво са:  
1. src - Съдържа всички изходни файлове на нашите приложения, включително шаблони, модели (обекти) и контролери.  
2. src / main / resources / static - всичко, което е в нашата статична папка (файлове, изображения, стилове, JavaScript скриптове и т.н.) ще бъде достъпно за всеки потребител.  
3. src / main / java / {packageName} / ... - ще съхраним цялата ни логика отзад (контролери, обекти и т.н.) тук (разбира се, в отделни папки).  
4. src / main / resources / templates - тук ще запазим шаблоните за преглед. Ще използваме шабловия двигател Thymeleaf.  
3. Изпълнете проекта  
След като отворихме проекта, нека се опитаме да го изпълним, за да можем да видим с какво работим. Отидете в горния десен ъгъл на IntelliJ Idea, където ще намерите бутон Run, който изглежда като зелен бутон за възпроизвеждане (бъдете търпеливи - първото изграждане може да отнеме известно време поради разрешаването на зависимостите):



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:

Ето как ще стартираме пролетното ни приложение. Отидете напред и кликнете върху бутона за възпроизвеждане. Ако всичко върви според плана, трябва да видим това съобщение в конзолата:

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

Сега можем да отворим страницата (в localhost: 8080) и да видим какво имаме:



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

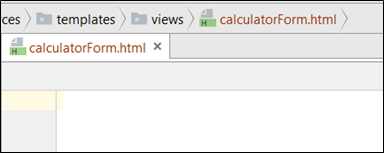
# III. Implementing Functionality

## 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:

Тя не изглежда много, но поне имаме основното оформление надолу! Нека да се заемем с изпълнението на някои функции!  
III. Внедряване на функционалност  
4. Създайте преглед на калкулатора  
Преди да започнем да добавяме функционалност, би било хубаво да имаме представа за това, срещу което работим, така че нека да продължим и да създадем формуляр, който потребителят ще използва за изчисления:  
Отидете в папката src / main / resources / templates / views / и отворете файла calculatorForm.html:



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:

Празно е?! Как се виждат горните и долните колони, показани по-горе? Отговорът е, че използваме глобален файл с оформление (base-layout.html), така че не е необходимо да копираме нашето разположение на страницата във всеки един изглед в нашия проект (който може да има десетки или стотици изгледи). Целият HTML дизайн на основния дизайн е в основата base-layout.html. Няма да се докосваме до това, така че нека да отидем до файла calculatorForm.html и да добавим формуляра ни:

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

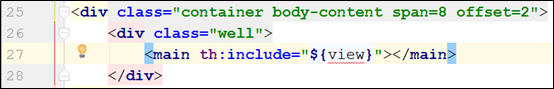
Just like with the JavaScript blog, we will **save the state** of the operands and operator for ease of use, so the **Thymeleaf syntax** you see here does just that. The th:selected="" template is a bit more special: it **automatically** **selects** the operator from the dropdown list, **based on** the 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](http://localhost:8080/) and see how we’re doing:

Точно както при JavaScript блог, ние ще запазим състоянието на операндите и оператора за лесна употреба, така че тилемеафският синтаксис, който виждате тук, прави точно това. Шаблонът th: selected = "" е малко по-специален: той автоматично избира оператора от падащия списък, въз основа на последния използван оператор. Ще видим как това се изпълнява малко по-късно. Засега нека да се придвижим до нашето уеб приложение на http: // localhost: 8080 и да видим как се справяме:



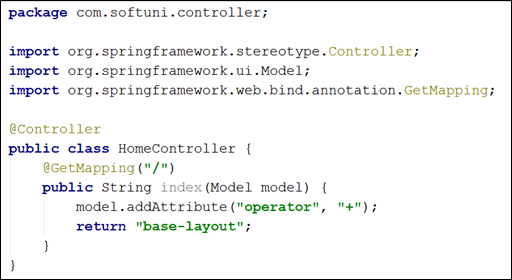
Still nothing?! The reason our form doesn’t display is because we’re not sending the **form view** to the user. To add the form to our project, we need to do two things: The first thing is to add this line of code in base-layout.html:

Все още нищо?! Причината, поради която нашата форма не се показва, е, че не изпращаме изгледа на формуляра на потребителя. За да добавим формуляра към нашия проект, трябва да направим две неща: Първото нещо е да добавите този ред код в base-layout.html:



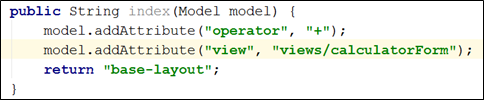
This line of code expects to be fed a **template** to display around the header and footer. So, we’re going to do just that. Go into the HomeController.java file and check out what the **index** action does:

Този ред код очаква да бъде нахранен шаблон за показване около заглавката и долния колонтитул. Така че, ние ще направим точно това. Отидете във файла HomeController.java и проверете какво прави индексното действие:



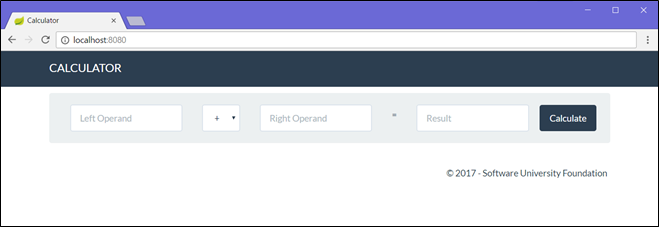
All this action does is listen on the “**/**” (site root) route, and display the **base-layout** view. But before it displays it, it **adds an attribute** to the model. The **model** is the **data** that’s fed to the **view**, so it can perform various functions (such as displaying the calculator’s operands, 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:

Цялото това действие е да слушате маршрута "/" (корен на сайта) и да показвате изгледа на основната оформление. Но преди да го покаже, той добавя атрибут към модела. Моделът е данните, които се подават към изгледа, така че той може да изпълнява различни функции (като показване на операндите на калкулатора, оператора и т.н.). Ще дадем още един атрибут. Не забравяйте, че шаблонът $ {view} очаква базовото оформление? Нека да я добавим тук, преди да върнем изгледа на основното оформление:



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

Дадохме го, така че нека да рестартираме пролетното ни приложение и да видим как се справяме:



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

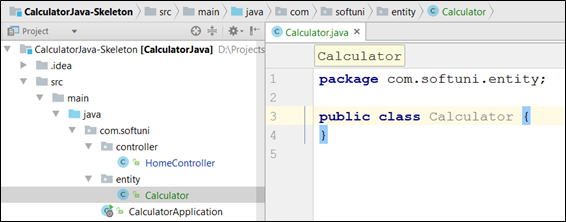
It’s time to design our main model – the **Calculator**. It will contain the following properties:

* + - * 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**”:

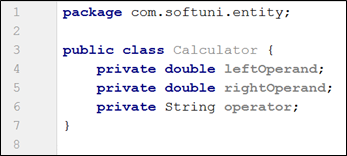
Изглежда добре! Освен че не прави нищо. Първо, да стигнем до създаването на нещо, което ще държи данните ни: модела.  
5. Създайте модел на калкулатор  
Време е да проектираме нашия основен модел - Калкулаторът. Той ще съдържа следните свойства:  
• leftOperand  
• оператор  
• rightOperand  
Нека да създадем нашия режим. Тъй като не използваме база данни в това упражнение, ние просто ще определим калкулатора като обикновен Java клас.  
Отидете в папката на обекта и създайте нов клас Java, наречен "Calculator.java":



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

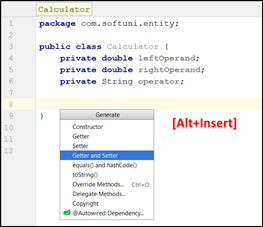
1.Create **fields**, which will be used internally in the class:

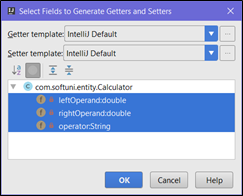
Сега, да определим какво ще има нашата класа:  
1. Създайте полета, които ще се използват вътрешно в класа:



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

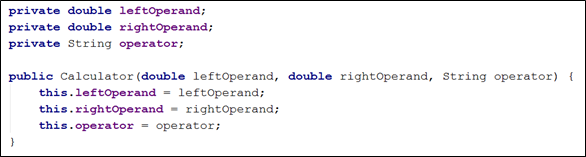
2. Определете свойствата на калкулатора:

 ->



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

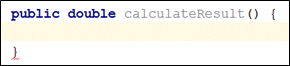
3. Създайте конструктор за инстанцииране на калкулатора:



4.Create **getters** and **setters** for theCalculator’s **fields** (you can make them with **[Alt + Insert]**)

5.Create a **method** for **calculating** the result from the **properties**:

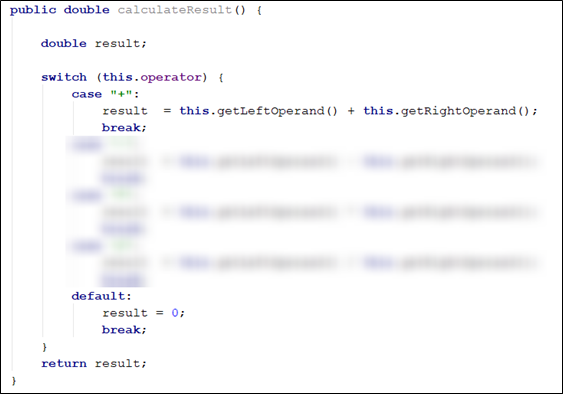
4. Създайте getters и setters за полетата на Calculator (можете да ги направите с [Alt + Insert])  
5. Създайте метод за изчисляване на резултата от свойствата:



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.

В този метод ще напишем логиката, която е необходима за изчисляване на резултата от операндите и оператора. Нека да създадем логиката, необходима за това.

6.Write the calculation logic: 6. Напишете логиката на изчислението:



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:

Нашата логика на калкулатора е добре вградена в клас Калкулатор. Сега остава само да го свържете с останалата част от нашето малко уеб приложение.  
За последния ни трик ще създадем свой собствен контролер, който ще обработва това, което потребителят ни е изпратил и ще върне изгледа с резултата от изчислението.  
6. Изпълнете действието на контролера  
Сега, след като създадохме изгледа, който ще държи данните ни и ще позволи на потребителя да взаимодейства с нашето уеб приложение, е време да приложим движещата сила зад цялото приложение - действието на контролера.  
Както се оказа, вече имаме инсталиран домашен контролер и действие, настроено на маршрута "/", в противен случай дори няма да можем да видим нашия калкулатор. Главният контролер можете да намерите в папката "контролер". Да видим как изглежда:



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

* @GetMapping("/") è 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) è 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) è 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" è 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**.

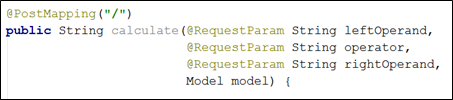
First, we’ll start off by declaring what kind of **HTTP method** this method will be handling (either GET or POST). In our case, since we’re processing **form data** sent to the “**/**” URL, we’ll set it to @PostMapping("/"):

Не се случва много тук ... Нека да го счупим:  
• @GetMapping ("/")  кода на кода, който свързва URL път към даден метод, така че той може да изпълни действието, когато потребителят звъни на определения маршрут (точно сега, това е "/").  
• публичен индекс на низове (модел модел)  Това е действителното действие на контролера. Това е метод, който държи логиката, която ще бъде изпълнена, когато се нарича.  
Той има един параметър: модел - той държи данните, които ще бъдат предадени на изгледа за обработка. Запомнете - всичко, което правим тук, е връщането на различен HTML, въз основа на логиката, която сме въвели в нашето приложение.  
• model.addAttribute (String, String)  тази част от кода има 2 параметъра - първата посочва името на атрибута, който ще изпратим към изгледа. Вторият - действителните данни.  
• Връщане на "базово оформление"  Тази функция оформя оформлението на отговора (по същество отнема каквото е вътре в "templates / base-layout.html", преминава през шаблона на Thymeleaf и го връща на потребителя.  
Така че, използвайки нововъведеното знание, нека се опитаме да създадем свои собствени действия.  
Първо, ще започнем, като обясним какъв тип HTTP метод ще обработва този метод (GET или POST). В нашия случай, тъй като обработваме данните от формуляра, изпратени до URL адреса "/", ще го настроим на @PostMapping ("/"):



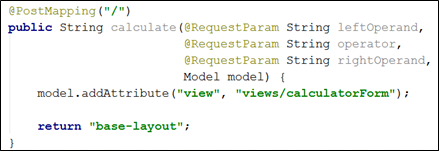
After that, let’s declare our method. We’ll use a couple of new types here – RequestParam. That’s just a fancy way of getting the form data:

След това нека декларираме нашия метод. Ще използваме няколко нови типа тук - RequestParam. Това е просто фантастичен начин за получаване на данните за формуляра:



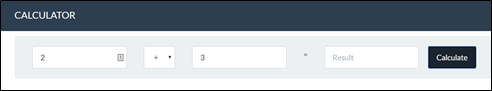
All this method should do at this point is just return the base-layout template with the **form attribute** inside of it:

Този метод, който трябва да направите на този етап, е просто да върнете шаблона за основно оформление с атрибута на формуляра вътре в него:

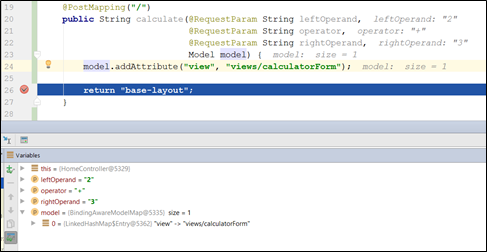


Let’s see what a debug session would show us if we were to debug this method:

Нека видим какво ще ни покаже сесия за отстраняване на грешки, ако трябва да отстраняваме този метод:

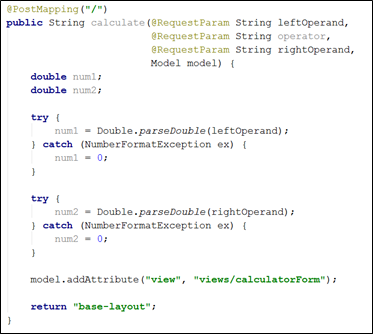






The leftOperand, operator, and rightOperand variables come in the form of **strings**. So, before we feed them to our **calculator** class, we need to **parse** them as numeric variables, using **try-catch** blocks:

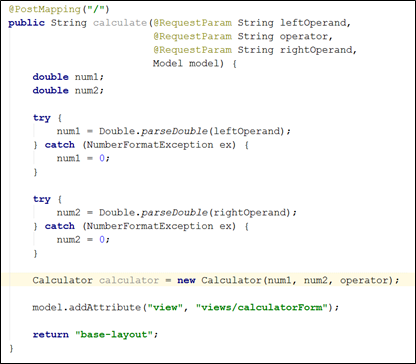
Променливите leftOperand, оператор и rightOperand идват под формата на струни. Така че, преди да ги нахраним в класа ни за калкулатори, трябва да ги анализираме като цифрови променливи, като използваме блокове за опита:



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

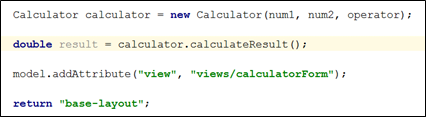
Next, we need to create an **instance** of our calculator model, which we’ll use for storing the data inside:

Това ще гарантира, че ако потребителят се опита да ни изпрати невалидни данни, вместо това ще бъде нулиран.  
След това трябва да създадем пример за нашия модел на калкулатора, който ще използваме за съхранение на данните в него:



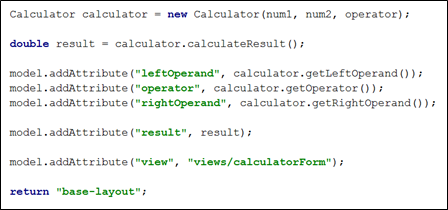
We can set its data, using its constructor, which we defined in step 4. Now that we’ve gotten the data, it’s time to calculate the result from what we currently have. Remember that calculateResult() function we wrote a while ago? Now is the time to use it:

Можем да зададем данните си, като използваме нейния конструктор, който определихме в стъпка 4. Сега, след като получихме данните, е време да изчислим резултата от това, което имаме в момента. Не забравяйте, че функцията calculateResult (), която написахме преди малко? Сега е времето да го използвате:



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:

След това остава само да изпратим резултата до изгледа. Освен резултата, ние също трябва да изпратим leftOperand, rightOperand и оператор, за да можем да запазим състоянието на нашия калкулатор чрез заявки. Можем да направим това, като използваме функцията model.addAttribute:



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:

По този начин посочваме какво ще изпратим до изгледа. Така че, когато изпратим резултантната стойност и предишното състояние на калкулатора към изгледа, можем да попълним полетата на формуляра с нашите данни. Това се случва тук:



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!

Използваме данните от контролера в изгледа на изгледите / calculatorForm.html, за да настроим стойностите на входовете на формуляри за това, което искаме. В този случай зададете операндите и изберете последния използван оператор.  
IV. Тествайте приложението  
Цялата ни упорита работа трябва най-накрая да се изплати, нали? Ако сте следвали всички стъпки правилно и сте прочели целия обяснителен текст, надяваме се да имаме функциониращ калкулатор!

