Lab: Calculator using Javascript and ExpressJS

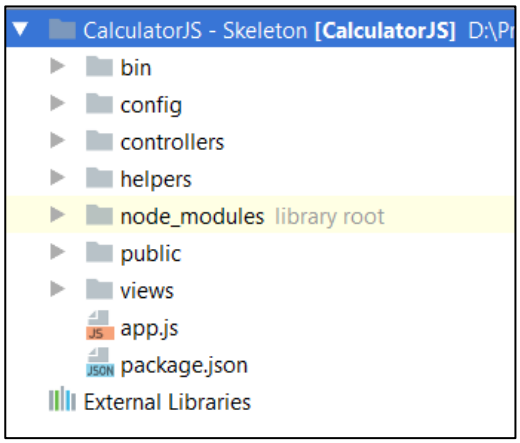
This document defines a complete walkthrough of creating a **Calculator** application with the Express.js Framework, from setting up the framework, to implementing the models, views and controllers necessary for our application to function.

Make sure you have already gone through the Getting Started: JavaScript guide. In this guide we will be using: WebStorm. You can download the **calculator's skeleton** from here. The rest of the needed non-optional software is described in the guide above.

I. Base Project Overview

Node.js is a **platform** written in **JavaScript** and provides **back-end** functionality. Express is a **module** (for now we can associate module as a **class** which provides some functionality), which wraps Node.js in way that makes coding faster and easier and it is suitable for the **MVC** architecture.

Initially the project comes with the following structure:

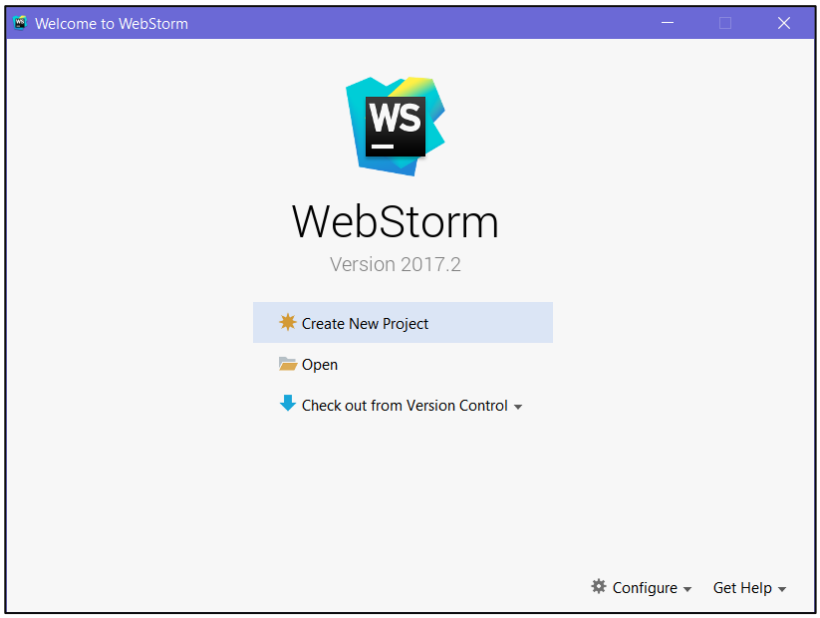


We can see several folders here. Let look at them one by one and see what are they for:

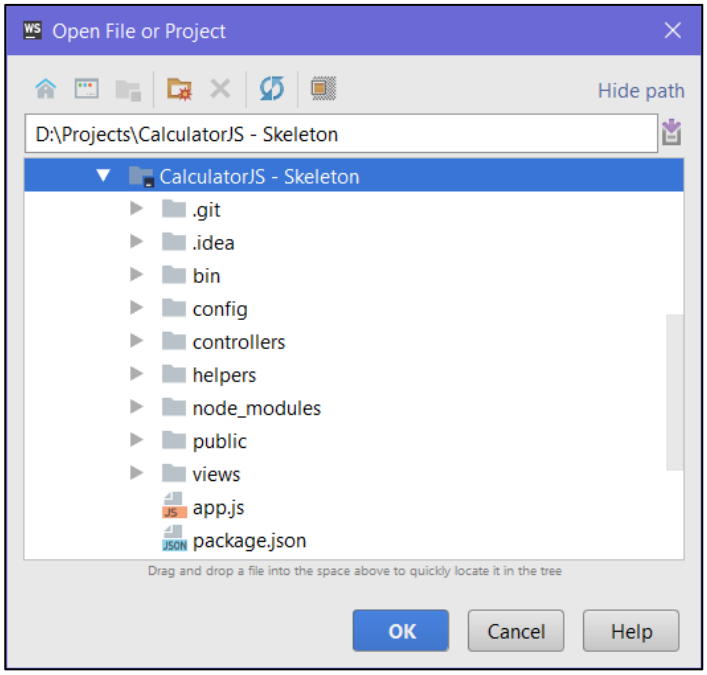
1. **`bin`** – Contains a single file named **www**, which is the **starting point** of our program. The file itself contains some **configuration** logic needed to run the server **locally**.
2. **`node_modules`** (library root) – As far as the name tells us, in this folder we put every library (**module**) that our project depends on.
3. **`public`** – Everything in our **public** folder (**files**, **images**, etc.) will be **accessible** by **every user**. We'll cover on that later.
4. **`routes`** – A folder in which we will put our **route** configurations. We'll find out what a **route** is in a bit.
5. **`views`** – Like in the previous calculator (in PHP), we again have a folder named **views**. There, we will store the views for our model. Again, we will use templates with the help of the **Handlebars** view engine (last time we used **Twig**).
6. **`app.js`** – The script containing the logic needed to **start the server**.
7. **`package.json`** – a file containing project information (like the project's **name**, **license** etc.). The most important thing is that there is a "**dependencies**" part, where all the names and versions of every module that our projects uses will reside.

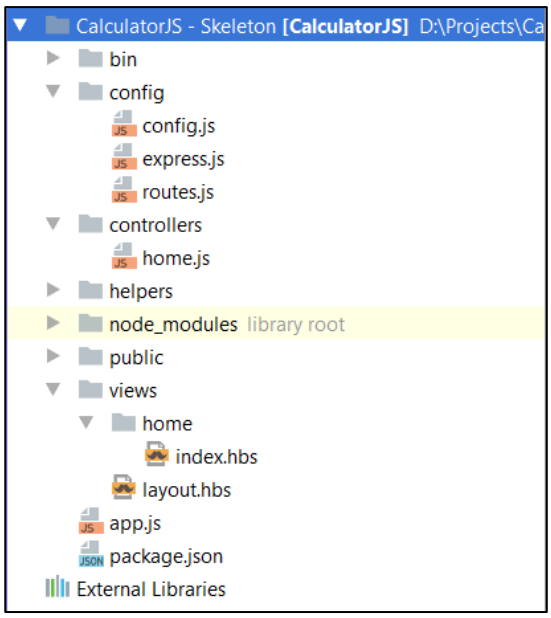
Let's see what we have in the skeleton before we start working on it:

1. Open the Project



Let's go ahead and load the skeleton. Click "**Open**", and find the downloaded and unzipped skeleton project:





This is our Node.js project. In the previous steps, we described on how we got here. Now let's talk about **Node.JS**:

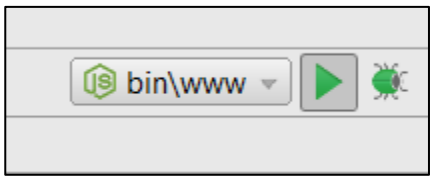
As we know, it's a **platform** written in **Javascript**, providing **back-end** functionality. This gives us a lot of flexibility, because our **front-end** usually uses **JavaScript** as well. This makes mutual **communication** easier.

Furthermore, Node.JS is fast because it uses C++ behind the scenes and also because it's capable of making asynchronous calls. It uses the event loop system.

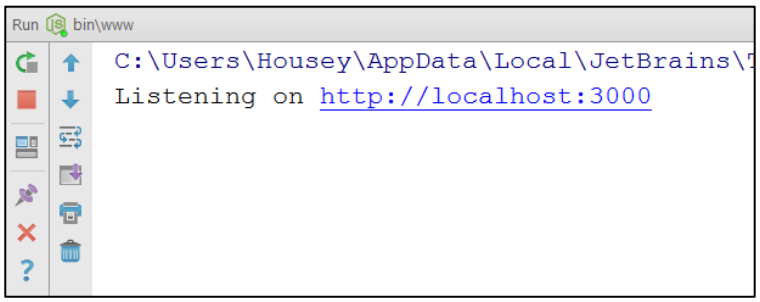
Summary: we have downloaded the project and we are ready to start writing code!

2. 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 WebStorm, where you'll find a Run button, which looks like a green play button:



That's how we'll start our Express app. Go ahead and click the button. If everything goes according to plan, we should see this message on the console:



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

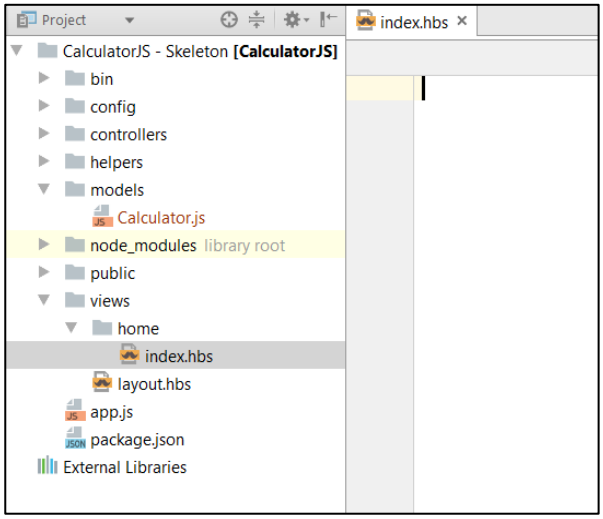
It doesn't look like much, but at least we have the basic layout down! Let's get to work on implementing some functionality!

II. Implementing Functionality

1. 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 **views/home** folder and open the **`index.hbs`** file:

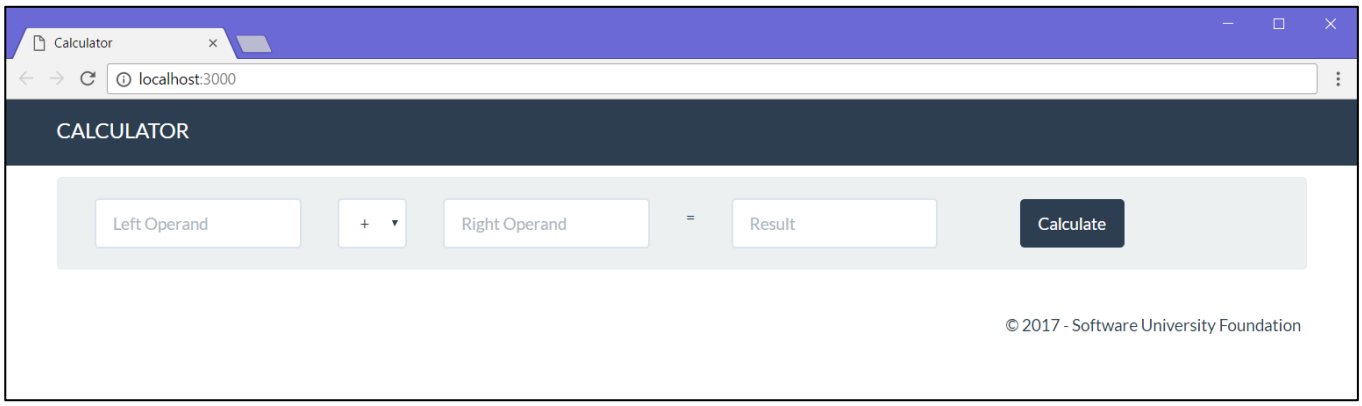


It's empty?! How does the header and footer seen above get displayed then? The answer is, we use a **global layout file**, so we don't have to copy-paste our page layout into every single view in our project (which could have **tens** or **hundreds** of views). All the **actual design HTML** is inside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

```
<div class="container body-content span=8 offset=2">
  <div class="well">
    <form class="form-inline" 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="calculator[leftOperand]" value="{{calculator.leftOperand}}">
          </div>
        </div>

        <div class="form-group">
          <div class="col-sm-4 ">
            <select class="form-control" name="calculator[operator]">
              <option value="+"{{selectif calculator.operator '+'}}>+</option>
              <option value="-"{{selectif calculator.operator '-'}}>-</option>
              <option value="*"{{selectif calculator.operator '*'}}>*</option>
```

```
            <option value="/"{{selectif calculator.operator '/'}}>/</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="calculator[rightOperand]" value="{{calculator.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" value="{{result}}">
        </div>
      </div>

      <div class="form-group">
        <div class="col-sm-4 col-sm-offset-4">
          <button type="submit" class="btn btn-primary">Calculate</button>
        </div>
      </div>
    </fieldset>
  </form>
 </div>
</div>
```

Just like with the PHP blog, we will **save the state** of the operands and operator for ease of use, so the **handlebars syntax** you see here does just that. The **`{{selectif}}`** helper 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:3000 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**.

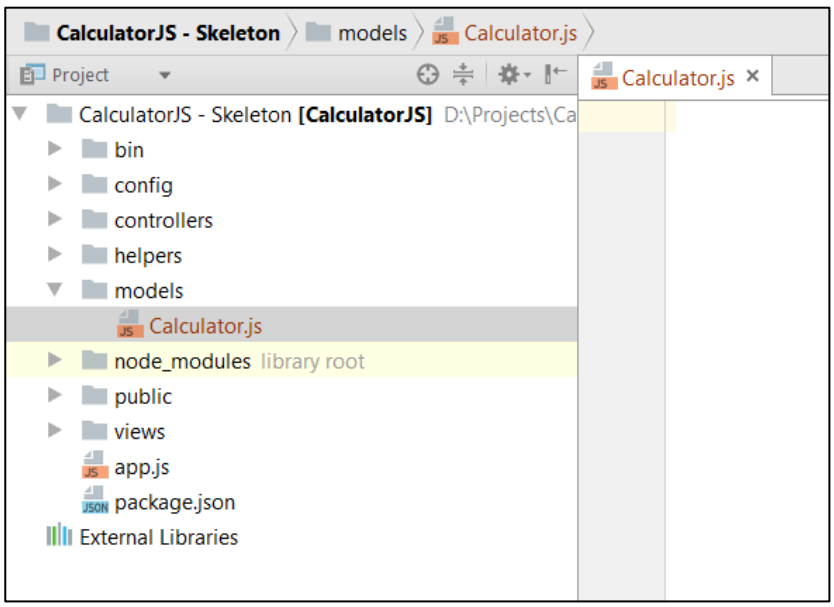
2. Create Calculator Model

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

- **`leftOperand`**
- **`operator`**
- **`rightOperand`**

Let's create our model in the **JavaScript** way. Since we're **not** using a database in this exercise, we're just going to define the calculator as a **simple JavaScript class**.

Create a folder, called "**models**" and inside it, create a **JavaScript** file, named "**Calculator.js**":



Now, let's write the code, which will define our **Calculator** model:

1. Create a function, which will be called upon creating a new calculator instance:

```
function Calculator (leftOperand, operator, rightOperand) {

}
```

2. Inside it, **define** the calculator **properties**:

```
function Calculator (leftOperand, operator, rightOperand) {
    this.leftOperand = leftOperand;
    this.operator = operator;
    this.rightOperand = rightOperand;
}
```

3. Create a **function** for **calculating** the result from the **properties**:

```
function Calculator (leftOperand, operator, rightOperand) {
    this.leftOperand = leftOperand;
    this.operator = operator;
    this.rightOperand = rightOperand;

    this.calculateResult = function() {

    }
}
```

Inside this function, 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:

4. Write the calculation logic:

```
this.calculateResult = function() {
    let result = 0;

    switch (this.operator) {
        case "+":
            result = this.leftOperand + this.rightOperand;
            break;
        case "-":
            result = this.leftOperand - this.rightOperand;
            break;
        case "*":
            result = this.leftOperand * this.rightOperand;
            break;
        case "/":
            result = this.leftOperand / this.rightOperand;
            break;
    }

    return result;
}
```

5. Almost done: **export** our model so it can be **visible** for the **outer world** (outside of the **Calculator.js** file):

```
function Calculator (leftOperand, operator, rightOperand) {
    this.leftOperand = leftOperand;
    this.operator = operator;
    this.rightOperand = rightOperand;

    this.calculateResult = function() {...}
}

module.exports = Calculator;
```

6.

Summary: We now know how to create a simple model and make it visible to the outside world.

3. Add a Route, which Calls the Controller Action

As for the routing – with **ExpressJS**, all our routing logic is usually located within a file, called **routes.js**. Here's what that currently looks like:

```
const homeController = require('./../controllers/home');

module.exports = (app) => {
    app.get('/', homeController.indexGet);
};
```

Let's break this code down into understandable chunks:

- **`const homeController = require('./../controllers/home')`**

  This bit of code **imports** our controller's logic. Before we can **call** any methods, we need to know those methods **exist**, right?

- **`module.exports`**

  This is the piece of code which takes the code inside it and **exposes** it to the outside world. We're putting our code here, because Node.JS needs to have **access** to it, so it can **execute the action** when our user **calls** the specified **route** (examples: **`/calculate`**, **`/edit/2`**, **`/login`**).

- **`app.get('/', homeController.indexGet)`**

This piece of code tells Node.JS to **listen** for any **GET requests** on the "**/**" route, and when it finds one – to execute the **`homeController.indexGet`** method (this is why we needed to **import** the **home controller** up there using **`require()`**.

Now that we've understood how the **routing system** in **Express.JS** works, it's time to **add** our **own route**, which calls our own **controller action**, which gets the data from the user and uses it to calculate the result.

Since we're going to be sending out **form data** to the server, we're going to be using a **HTTP POST request** to do it. Hence, we need to tell **routes.js** to expect **POST requests** and **execute** the appropriate controller action:

```
module.exports = (app) => {
    app.get('/', homeController.indexGet);
    app.post('/', homeController.indexPost);
};
```

We just made a **route**, which listens on "**/**", and upon matched, executes the **`indexPost`** action inside the **home controller**. That's alright, but the problem is we **don't have** an action with that name yet. So, guess what happens if we try to send a form to the server:

Not so great. 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.

4. 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 create the driving force behind the whole app – **the controller**.

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 "**controllers**" folder. Let's see what it looks like:

home.js

```
module.exports = {
    indexGet: (req, res) => {
        res.render('home/index');
    }
};
```

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

- **`module.exports`** ➔ the piece of code, which takes the code inside it and **exposes** it to the outside world. We're putting our code here, because the **router** needs to have **access** to it, so it can **execute the action** when our user **calls** the specified **route** (right now, that's "**/**").
- **`indexGet: (req, res)`** ➔ This is the actual **controller action**. It's a function, which **holds the logic**, which will be **executed**, when it's **called**. It's **no different** than a **regular method**.
  It has 2 parameters: **`req`** and **`res`**. They hold data about the **HTTP request** and **HTTP response** respectively. They'll be used for **getting data** from the user and also doing things such as **rendering views** in the

response. Remember – all we're doing here is returning different **HTML code**, based on the logic we've implemented in our app.

- **`res.render('home/index')`** ➔ This function **renders** a **view** in the **response** (in essence, takes whatever's inside of "**`home/index.hbs`**", runs it through the **Handlebars** templating engine, and returns it back to the user.

So, using this newfound knowledge, let's try to create our own **action**.

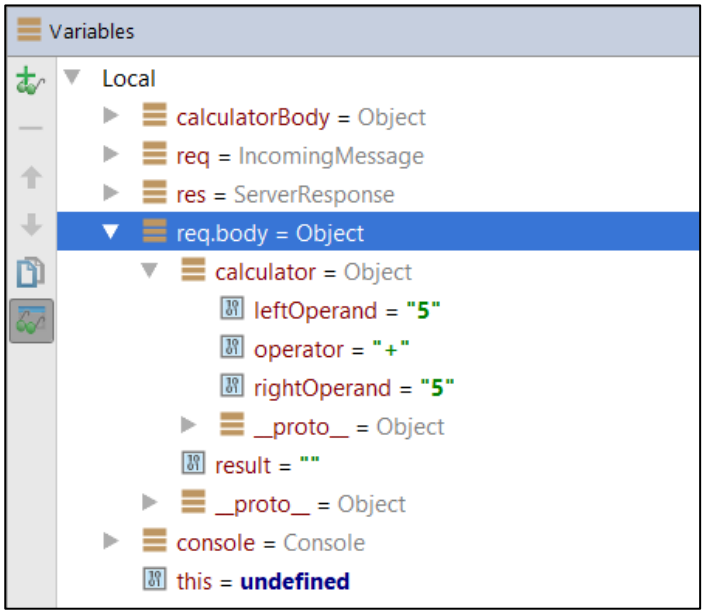
First, we'll start off by declaring the action:

```
module.exports = {
    indexGet: (req, res) => {
        res.render('home/index');
    },
    indexPost: (req, res) => {

    }
};
```

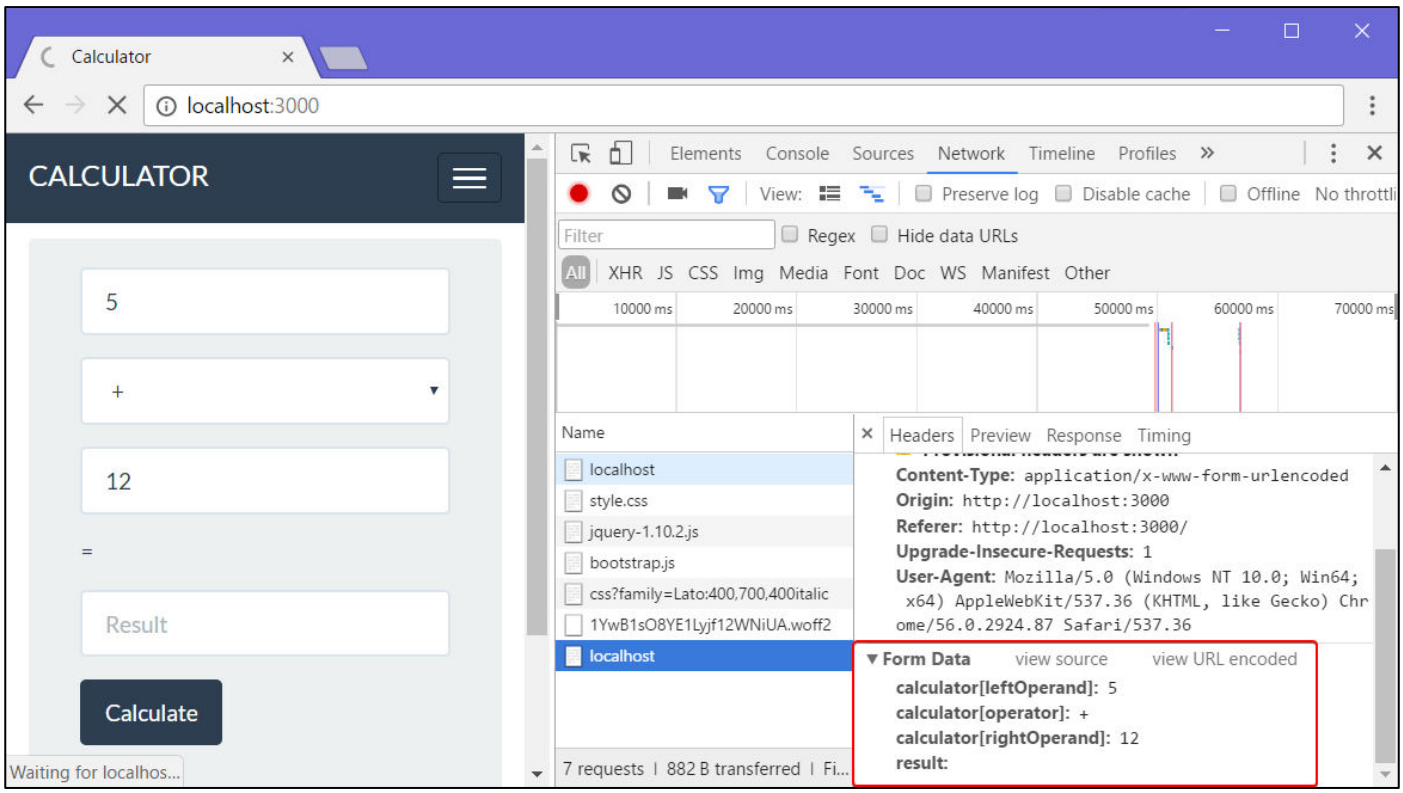
As we know from above, we have **`req`** and **`res`** parameters. We'll use the **`req`** parameter to get the data from the **request**. That's the data the user **sent us** through the **form**. We can get a hold of that data by accessing **`req.body`**:

```
indexPost: (req, res) => {
    let calculatorBody = req.body;
},
```

Before we continue, let's see what that property would hold if we looked at it during a **debug session**:



The data comes through the request as a **calculator** variable. Why does that happen? Well, let's look at what data we're sending the server with a tool like **Chrome Developer Tools (F12)**:



The form has a **calculator variable** with **3 values**, sent as **strings**. And as such, that comprises the **request body** – something we can access by using the **`req.body`** property. Let's go back into our controller action and **process** the data.

We can access the properties of the calculator by accessing the **request body** and getting the **calculator** as a key:

```
indexPost: (req, res) => {
    let calculatorBody = req.body;

    let calculatorParams = calculatorBody['calculator'];
```

Accessing **`calculatoryBody`**'s values by the "**calculator**" key is the same as just writing "**`req.body['calculator']`**", but we're extracting every step into variables for clarity.

Before we can use our **calculator model**, we need to tell the controller that it exists. We'll do that by **importing** it, using **`require()`** at the **top** of the file:

```
const Calculator = require('../models/Calculator');
```

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

```
indexPost: (req, res) => {
    let calculatorBody = req.body;

    let calculatorParams = calculatorBody['calculator'];

    let calculator = new Calculator();
    calculator.leftOperand = Number(calculatorParams.leftOperand);
    calculator.operator = calculatorParams.operator;
    calculator.rightOperand = Number(calculatorParams.rightOperand);
}
```

We use the **`Number()`** function to convert the operands from **strings** to **numbers**. 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's the time to use it:

```
let calculator = new Calculator();
calculator.leftOperand = Number(calculatorParams.leftOperand);
calculator.operator = calculatorParams.operator;
calculator.rightOperand = Number(calculatorParams.rightOperand);

let result = calculator.calculateResult();
```

After that, all we have left is to **render the view**. We can do that by using the **`render`** function inside the **`res`** (response) method parameter:

```
let result = calculator.calculateResult();

res.render('home/index', {'calculator': calculator, 'result': result});
```

Let's break down what **`res.render()`** does:

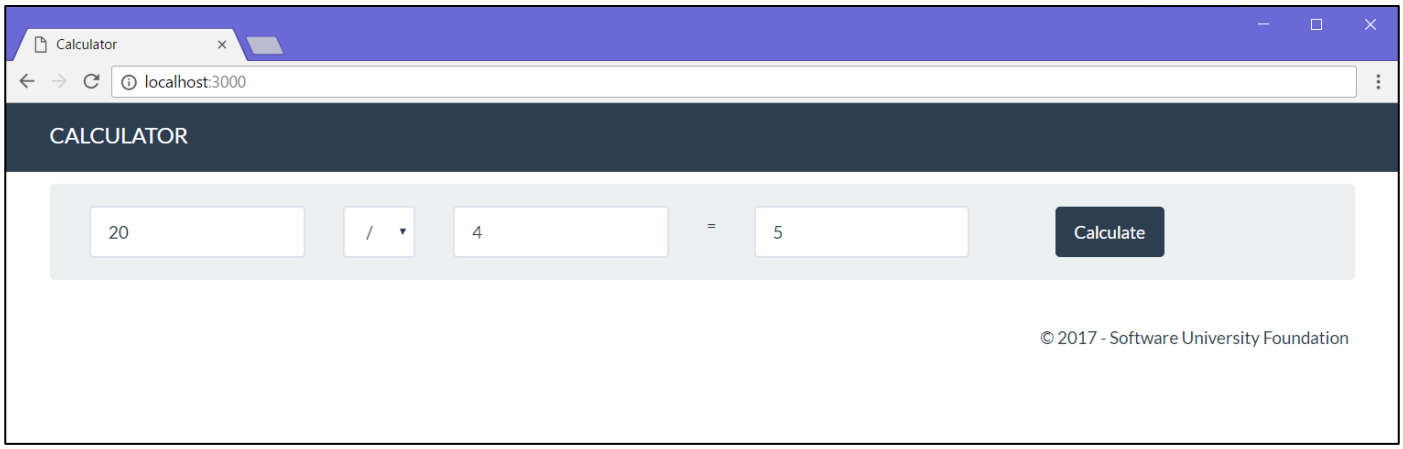
- **`'home/index'`**
  This parameter specifies which view to return.
- **`{'calculator': calculator, 'result': result}`**
  This parameter is a **JavaScript Object**, which specifies what we're going to **send** to **the view** (in our well-known **`key -> value`** pairs).
  So, when we send over the **calculator object** and the **result value** to the view, we can **fill** the **form fields** with our data. This happens here:

```
<div class="container body-content span=8 offset=2">
    <div class="well">
        <form class="form-inline" 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="calculator[leftOperand]" value="{{calculator.leftOperand}}">
                    </div>
                </div>

                <div class="form-group">
                    <div class="col-sm-4 ">
                        <select class="form-control" name="calculator[operator]">
                            <option value="+"{{selectif calculator.operator '+'}}>+</option>
                            <option value="-"{{selectif calculator.operator '-'}}>-</option>
                            <option value="*"{{selectif calculator.operator '*'}}>*</option>
                            <option value="/"{{selectif calculator.operator '/'}}>/</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="calculator[rightOperand]" value="{{calculator.rightOperand}}">
                    </div>
                </div>
```

We use the data from the controller in the **`home/index.hbs`** 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**.

III. Test the Application

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



IV. * 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 the one 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 JS! ☺