Calculator: PHP and Symfony

This document defines a complete walkthrough of creating a **Calculator** application with the Symfony Framework

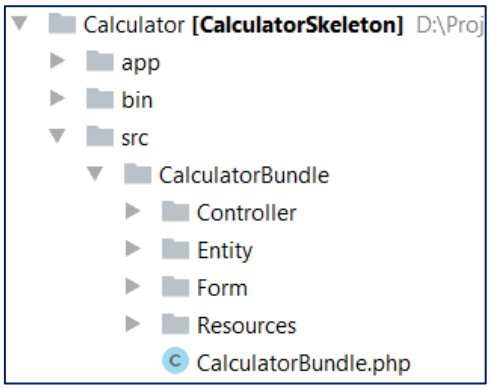
Make sure you have installed XAMPP, and added PHP root folder to the path environment variable.

You can download the **calculator skeleton** from here.

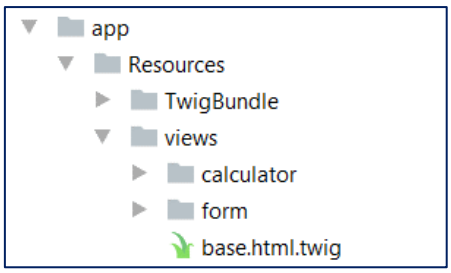
I. Symfony Base Project Overview

Symfony is a modular enterprise web-framework, which comes with a solid vendor support, **bundle** system, **enterprise** mechanisms and is most-suitable for **MVC** architecture.

Initially the project comes with a main bundle, which can be treated as a plugin later. A **bundle** often has **Controller**, **Entities** and related components (e.g. Repositories, Forms, Commands…)

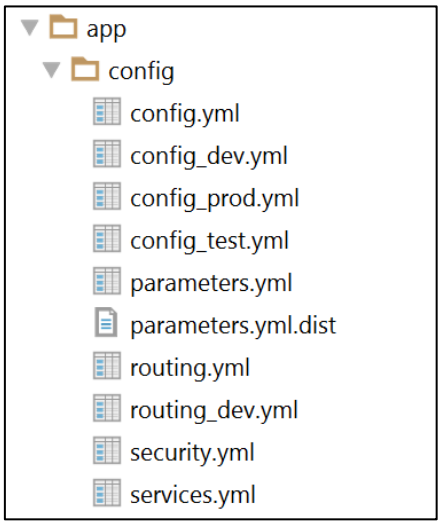


Standard templates (views) reside in the application folder (**app**) and are usually separated in folder named after the **controller names**.



The de-facto standard **View Engine** in Symfony is Twig.

The base configuration of the project is placed in **`app/config`**, where a configuration files for the Doctrine connection are defined, Security management, Routing rules, registering Services and so forth.



The **`parameters.yml.dist`** file is very important to contain the **same** keys as in **`parameters.yml`**, because installing new bundle will delete unused pairs.

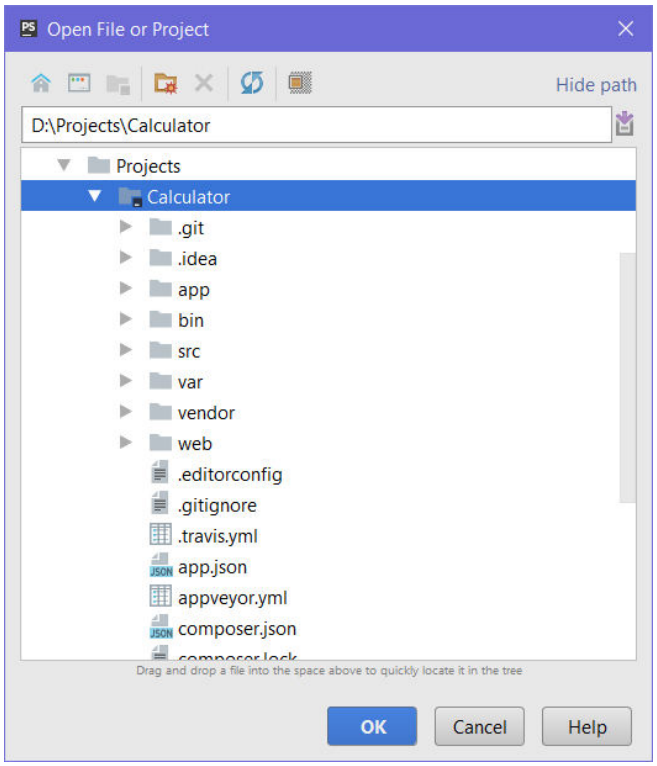
II. Initial steps

1. Open the Project

For this step, we will open the project with **PhpStorm** or **IntelliJ** Idea. Starting from the home screen, click on "**Open**":

Note: extract the folder into a short path (e.g. D:\Projects\Calculator), otherwise you might face random errors due to the Windows operating system having a path length limit.

Locate the skeleton folder that we gave to you and select the "**Calculator**" **folder** from the extracted folder (e.g. **D:\Projects\Calculator**):



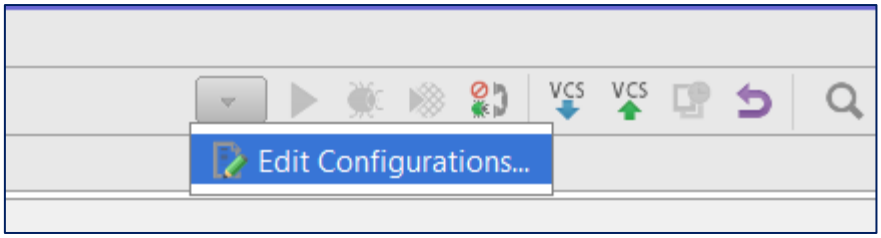
After you click "**OK**" the project should start loading and indexing. After a few seconds/minutes depending on your pc, you will be able to work with the project.

2. Run the Project

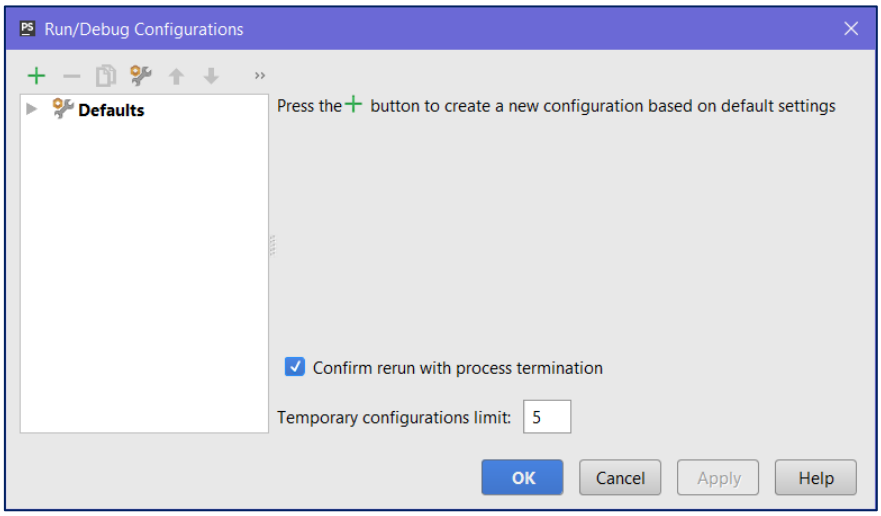
After we open the project, let's try to run it. Before doing anything, make sure you've set the PHP root folder environment variable, so you can call the PHP executable from anywhere. If you haven't done so before opening the project, you can do it by following the info in this link.

After you're certain you've added PHP to the environment variables, it's time to create a **run configuration**.

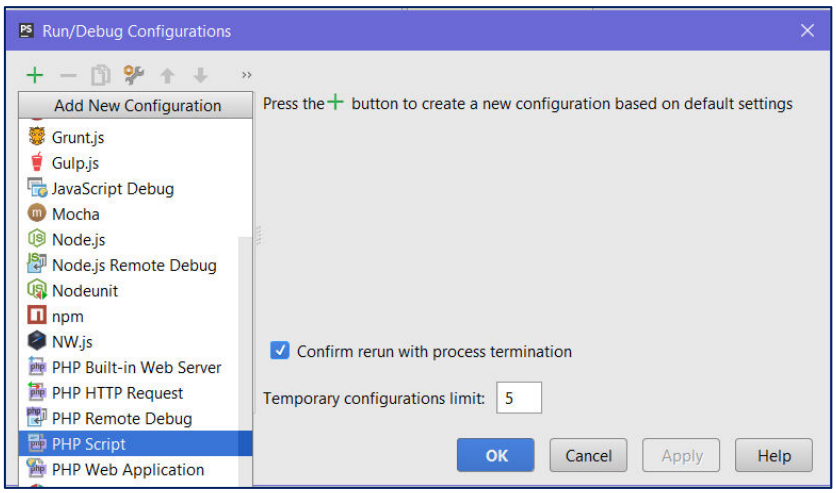
Open the **edit configurations** screen from the **top right** corner of PHPStorm/IntelliJ:

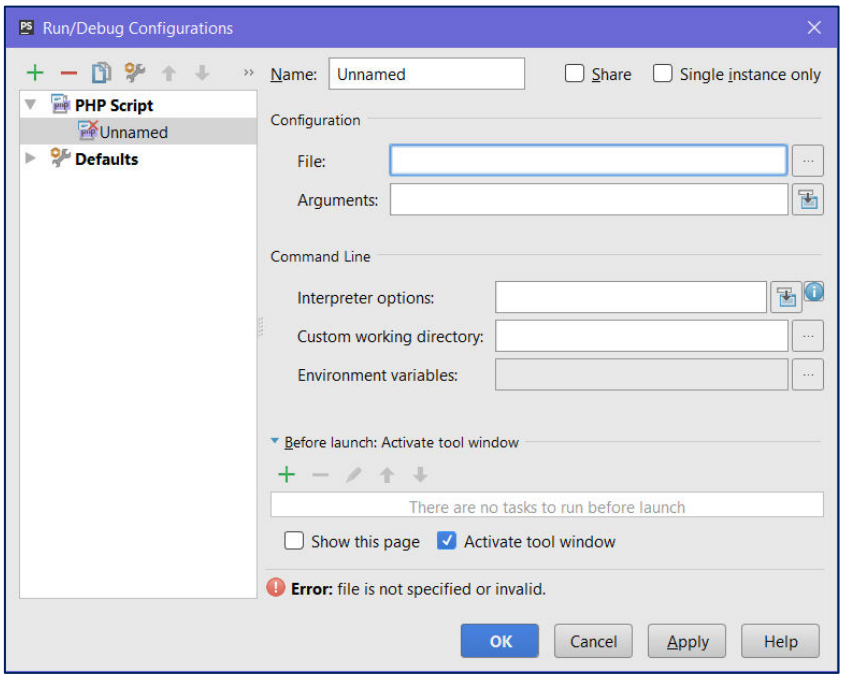


You will be greeted by this screen:

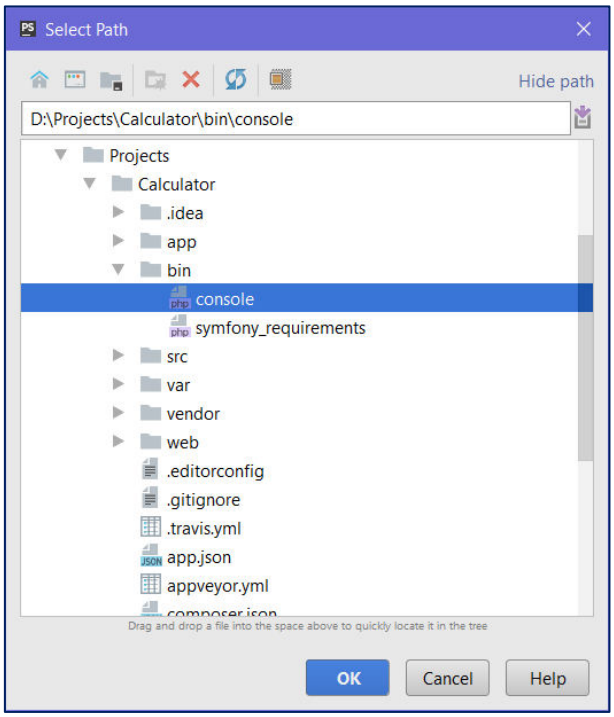


Open the add menu (+) and add a **PHP Script** run configuration:

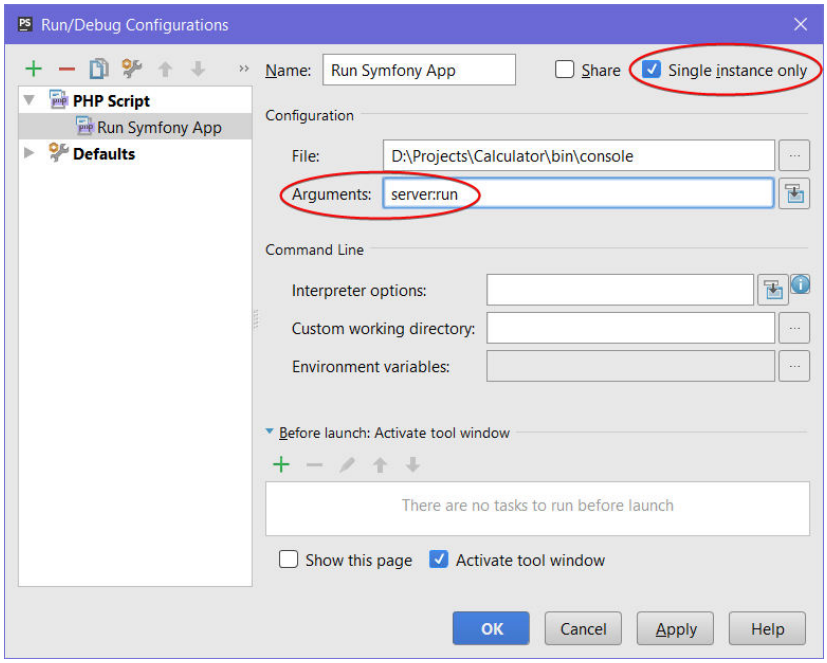




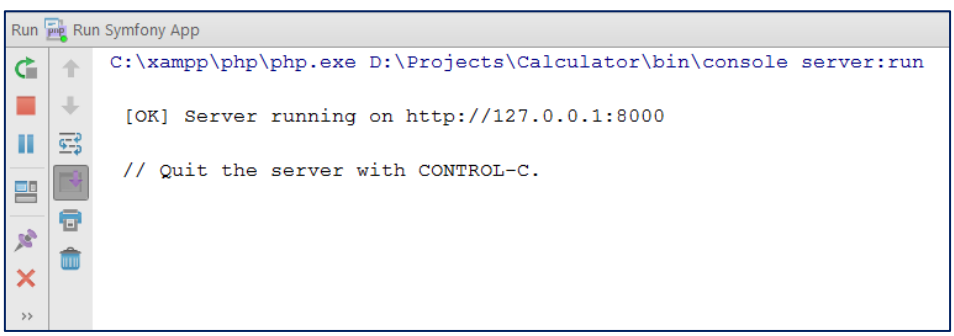
Check the "**Single instance only**" checkbox. After that, point the **File** textbox to the **bin/console** file, which is inside your project directory:



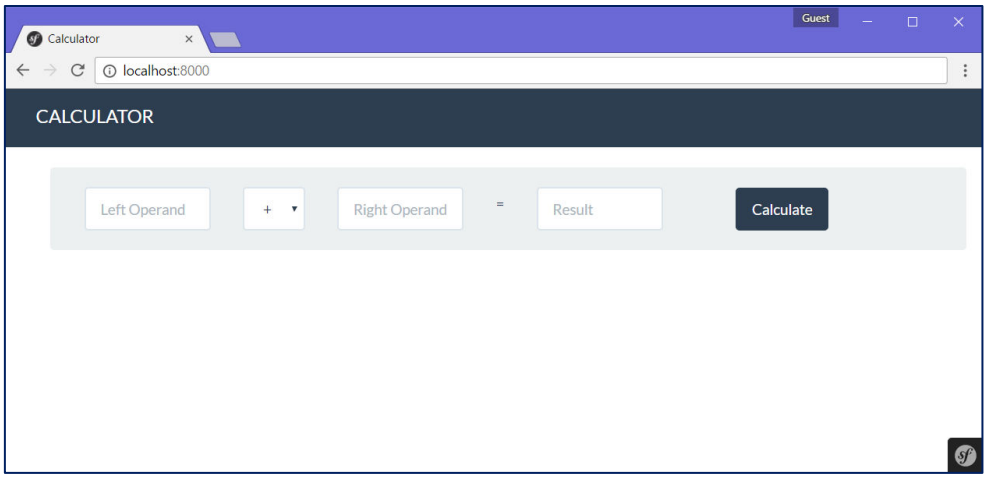
After that, add the "**`server:run`**" argument to the **arguments**. In the end, make sure your run configuration looks something like this:



Now, if you attempt to run it by using the play button on the top right, if everything works correctly, you should be greeted by this screen on the bottom:



All the run configuration does is simply run the command line colored in blue, so you don't have to type it into a command prompt every time. If you visit **`localhost:8000`** in your web browser, you will be greeted by the calculator application!



It looks great, but it **doesn't work**. So, let's go in and write some code to make those textboxes interact.

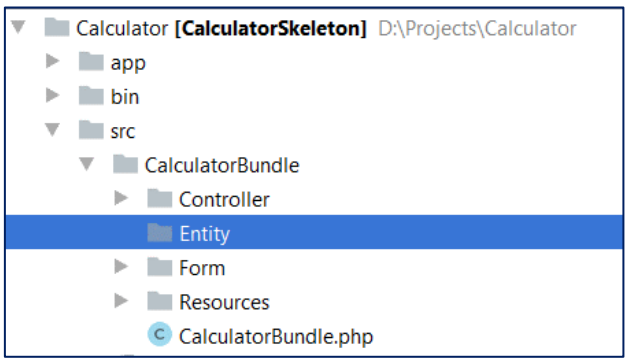
III. Adding Functionality

i. Create the Calculator Entity

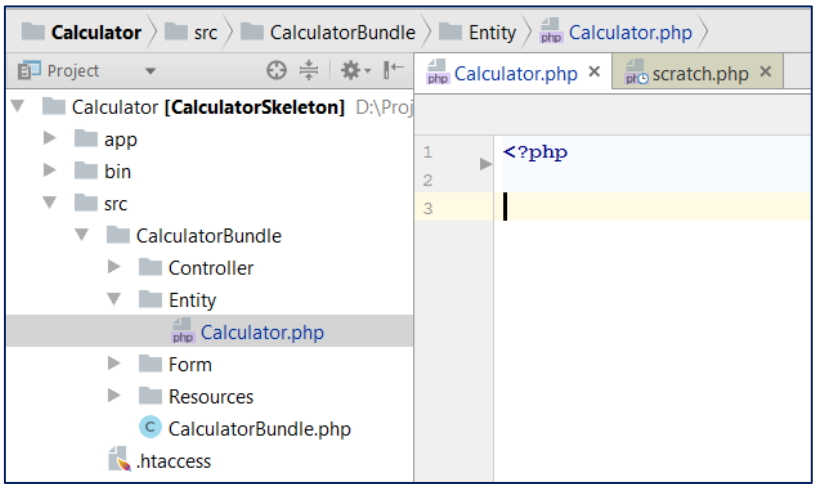
Before we can do any calculations, we're going to create a file which will store the data from the requests we'll be receiving, such as the operands and the operator. In order to do this, we will define a simple **class** and use it in our **controller**.

1. Create Calculator Class File

Since we won't be using a database in this exercise, we **won't** be using **Doctrine** to generate our entities. Instead, we'll **make them ourselves**. Head on over to the **`src/CalculatorBundle/Entity`** folder:



Create a new PHP file inside it, called **`Calculator.php`**:



We have an empty PHP file now. Let's fill it with stuff. Add the namespace, so **Symfony** knows it's an **entity**. After that, make a **class** inside, called **`Calculator`**:

```
<?php

namespace CalculatorBundle\Entity;

class Calculator
{

}
```

2. Create Fields

Now it's time to start adding **fields**, which **describe** our entity, and **properties**, so it can be accessed from the outside world. Our calculator will have three fields:

- **`leftOperand`** ➔ the left operand of the calculation. It will have a **`float`** data type.
- **`rightOperand`** ➔ the right operand of the calculation. It'll have the same type as **`leftOperand`**.
- **`operator`** ➔ the operator of the calculation (**`+`**, **`-`**, **`*`** or **`/`**). It will have a **`string`** type

Let's add the **`leftOperand`** field:

```
class Calculator
{
    /**
     * @var float
     */
    private $leftOperand;
}
```

The comment above it is actually a **PHP annotation**. Make sure to add it, otherwise the application won't work correctly.

Let's add the other two fields as well – the **right operand** and the **operator**:

```php
<?php

namespace CalculatorBundle\Entity;

class Calculator
{
    /**
     * @var float
     */
    private $leftOperand;

    /**
     * @var float
     */
    private $rightOperand;

    /**
     * @var string
     */
    private $operator;
}
```

Note that the **`$operator`** field has a string type instead of a float type. This is because we're using it to store the operator as a string (e.g. **`"+"`**, **`"-"`**, **`"*"`** or **`"/"`**).

3. Create Accessors and Mutators (Getters and Setters)

Generally, fields have a **private** access modifier, so nothing can access them from the outside. In order to make other classes able to access it, we need to make what are called **getter** and **setter** methods (also called Mutator and Accessor methods). The **purpose** of **getters** and **setters** is that, since they're **methods**, we can add **other logic** inside, such as **validation**, which can keep the user from interacting directly with the field's value and potentially use that access for **malicious purposes**.

Let's make the **getter** method for the **`leftOperand`** field first:

```php
/**
 * Get left operand
 *
 * @return float
 */
public function getLeftOperand()
{
    return $this->leftOperand;
}
```

All this getter method does is return the field from the class. It doesn't contain any validation per se, but we could add it in the future by just inserting logic into the method before returning the field.

Let's create the **setter** method for **`leftOperand`** too:

```php
/**
 * Get left operand
 *
 * @return float
 */
public function getLeftOperand()
{
    return $this->leftOperand;
}

/**
 * Set left operand
 *
 * @param float $operand
 *
 * @return Calculator
 */
public function setLeftOperand($operand)
{
    $this->leftOperand = $operand;

    return $this;
}
```

The setter takes one **$operand** as a parameter, sets the **leftOperand** field to the parameter's **value** and **returns** the object itself. This will be the blueprint for all classes we make from now on (with different fields and mutators/accessors, of course). We can also further extend the class by adding functions, which operate on the class and so on. But more on that later…

For now, let's just create the rest of the getter and setter methods – for the **rightOperand** and **operator** fields:

```php
/**
 * Get right operand
 *
 * @return float
 */
public function getRightOperand()
{
    return $this->rightOperand;
}

/**
 * Set right operand
 *
 * @param float $operand
 *
 * @return Calculator
 */
public function setRightOperand($operand)
{
    $this->rightOperand = $operand;

    return $this;
}
```

```php
/**
 * Get operator
 *
 * @return float
 */
public function getOperator()
{
    return $this->operator;
}

/**
 * Set operator
 *
 * @param string $operator
 *
 * @return Calculator
 */
public function setOperator($operator)
{
    $this->operator = $operator;

    return $this;
}
```

We are done with our Calculator entity, so let's move on to implementing the action, which makes the app work.

ii. Implement the Calculate Action

Now that we have an actual **class** to put our left operand, right operand and operator inside, we need to make an **action** which can **take them** and perform an **actual calculation**. We'll do this by **modifying** the **Calculator Controller** to suit our needs.

1. Create a Form In-app

Let's go into the **`CalculatorController.php`**:

```php
<?php

namespace CalculatorBundle\Controller;

use ...

class CalculatorController extends Controller
{
    /**
     * @param Request $request
     *
     * @Route("/", name="index")
     *
     * @return \Symfony\Component\HttpFoundation\Response
     *
     */
    public function index(Request $request)
    {
        return $this->render('calculator/index.html.twig');
    }
}
```

Looks pretty empty at the moment. All we have is one function, called **`index`**, which returns the **index view**. No calculation going on here. Let's make it work like a calculator!

The **http response** we're giving **the client** works in the following way: When the client gets a **response**, the **web browser**'s rendering engine **renders** it onto their screen, turning the HTML, we gave the client to a full webpage.

We're going to edit the **`index`** function to make use of the **form**, which gets **sent by** the client and have it **use** the **values** the user sent us through the **POST** request by clicking the **Submit** button. In order to do that, we must first **acknowledge** the **form** is **being sent** at all:

Let's start by creating a **calculator variable** where we'll **store** our **operands** and **operator**:

```
public function index(Request $request)
{
    $calculator = new Calculator();

    return $this->render('calculator/index.html.twig');
}
```

We're not quite done yet. We have a bit more work to do. Next, we need to create a **form** variable, which will create a **special token** for the user and also, more importantly, **take** the **values** from **the form the user sent** us, and stick them in the **`$calculator`** variable, so we can work with them:

```
public function index(Request $request)
{
    $calculator = new Calculator();

    $form = $this->createForm(CalculatorType::class, $calculator);

    return $this->render('calculator/index.html.twig');
}
```

Next, before we implement the logic for checking if what the user sent us was valid, we have to actually **process** the **request**. We do this with the **`$form->handleRequest()`** method:

```
public function index(Request $request)
{
    $calculator = new Calculator();

    $form = $this->createForm(CalculatorType::class, $calculator);

    $form->handleRequest($request);
```

Afterwards, we need to check two things:

1. The user sent us a form **at all**
2. The user sent us a **valid form** (a form with **two operands** and an **operator** is considered a valid form):

```
public function index(Request $request)
{
    $calculator = new Calculator();

    $form = $this->createForm(CalculatorType::class, $calculator);

    $form->handleRequest($request);

    if ($form->isSubmitted() && $form->isValid()) {

    }

    return $this->render('calculator/index.html.twig');
}
```

After that, we can be sure that if we got in-between those two parentheses, the user sent us something we can actually work with, and the only thing which remains is to **implement** the actual calculator logic.

2. Implement Calculator Logic

If our form was **submitted** and **valid**, Symfony automatically **inserts** the form values into our **`$calculator`** variable. So we can now use that to implement the calculator logic. Let's start by **extracting** our **operands** and **operator** into variables for easier typing:

```
if ($form->isSubmitted() && $form->isValid()) {
    $leftOperand = $calculator->getLeftOperand();
    $rightOperand = $calculator->getRightOperand();
    $operator = $calculator->getOperator();
```

Next, we need somewhere to store the result, right? Right. So let's make a variable for that:

```
if ($form->isSubmitted() && $form->isValid()) {
    $leftOperand = $calculator->getLeftOperand();
    $rightOperand = $calculator->getRightOperand();
    $operator = $calculator->getOperator();

    $result = 0;
}
```

After that, let's implement some calculator logic by using a **switch case** on the **operator**:

```
if ($form->isSubmitted() && $form->isValid()) {
    $leftOperand = $calculator->getLeftOperand();
    $rightOperand = $calculator->getRightOperand();
    $operator = $calculator->getOperator();

    $result = 0;

    switch ($operator) {
        case '+':
            $result = $leftOperand + $rightOperand;
            break;
        case '-':
            $result = $leftOperand - $rightOperand;
            break;
        case '*':
            $result = $leftOperand * $rightOperand;
            break;
        case '/':
            $result = $leftOperand / $rightOperand;
            break;
    }
}
```

Almost there now… After implementing the logic, shouldn't that logic yield some result from our little web app? The answer is yes. After a valid calculation – if everything went well, we should return a HTTP response to the user with the calculated **value**:

```
switch ($operator) {
    case '+':
        $result = $leftOperand + $rightOperand;
        break;
    case '-':
        $result = $leftOperand - $rightOperand;
        break;
    case '*':
        $result = $leftOperand * $rightOperand;
        break;
    case '/':
        $result = $leftOperand / $rightOperand;
        break;
}

return $this->render('calculator/index.html.twig',
    ['result' => $result, 'calculator' => $calculator, 'form' => $form->createView()]);
```

Let's break down this return statement:

- **`$this->render()`** tells the **controller** which **view** to return.
- The **`render()`** function accepts **two parameters**:
  - A **string**, indicating the **view** we need to return (in our case – **`calculator/index.html.twig`**)
  - An **associative array**, indicating the **data** we're handing to the **view**. In our case, that would be:
    - the **result value**
    - the **calculator** itself (so we can keep our **operands** and **operator** in-between requests)
    - the **form** we're going to **create** for the user with its **special token** (**`$form->createView()`**)

Note: Make sure this return is **inside the if**, which checks if a **valid form** is being sent. There's **no use** in returning a result if we **weren't** given **valid data** to calculate with, right?

At this point, **all** of our logic is implemented **correctly** but this **won't work**. Why???

The reason is that, before we accept a form from the user, we need to **create the form**, using that **special token** we talked about earlier. If we don't **have** the token to begin with in our **html**, our "**is this form valid?**" check will **fail**. So as such, the last thing we need to do is edit the **return** statement, for when we **don't** have a form to process:

```
return $this->render('calculator/index.html.twig');
```

↓

```
return $this->render( view: 'calculator/index',
    ['form' => $form->createView()]);
```

iii. * Play Around with your Calculator App

Now that you've implemented the basic functionality, try to implement some extra functionality by yourself.

1. Add a New Operator

Some ideas for extra functionality include:

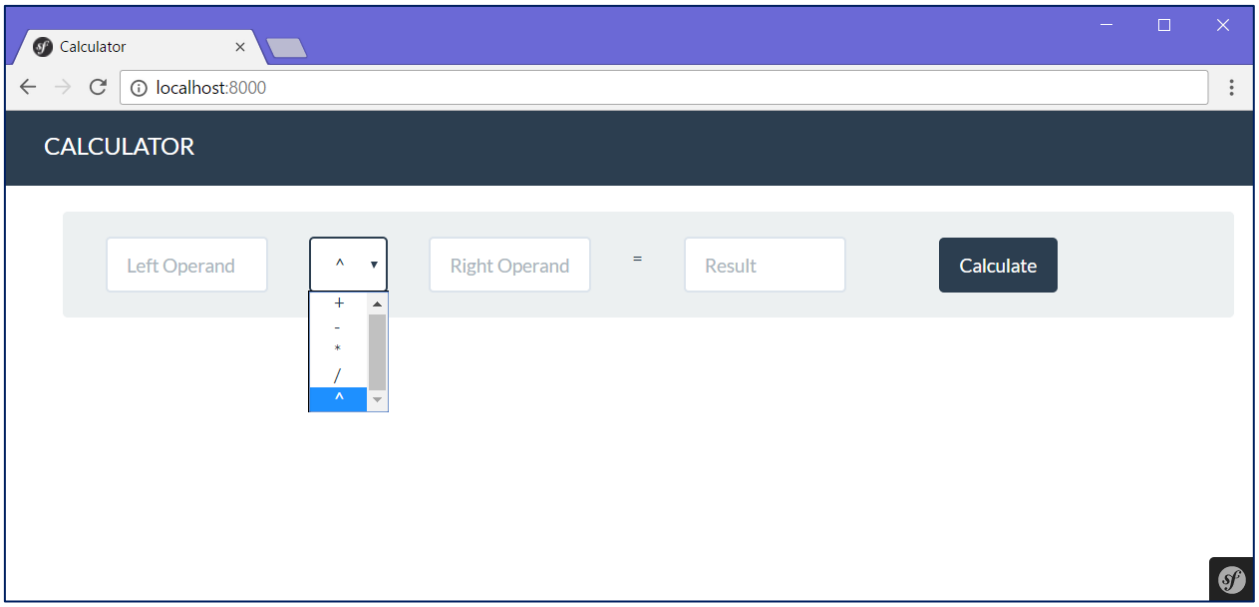
- Implementing **exponentiation** (4^2 = 16, etc.)
- **Bitwise** operations (**OR**, **AND**, **XOR**, etc.)

Hint: to add an operator, you must edit the form in the **`index.html.twig`** template, located in the **`app/Resources/views/calculator/index.html.twig`** path inside your project. There you can find the part of the form, which is responsible for listing the operators:

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

It has some extra logic inside each **`<option>`** tag, in the form of **twig syntax**, which will **select** the **last used** operator when transitioning between calculations. In order to add an operator, we can just copy one of the **`<option>`** tags and edit it to suit our needs:

```
<div class="form-group">
    <div class="col-sm-4 ">
        <select class="form-control" name="calculator[operator]">
            <option value="+" {{ calculator is defined and calculator.operator == '+' ? 'selected' : '' }}>+</option>
            <option value="-" {{ calculator is defined and calculator.operator == '-' ? 'selected' : '' }}>-</option>
            <option value="*" {{ calculator is defined and calculator.operator == '*' ? 'selected' : '' }}>*</option>
            <option value="/" {{ calculator is defined and calculator.operator == '/' ? 'selected' : '' }}>/</option>
            <option value="^" {{ calculator is defined and calculator.operator == '^' ? 'selected' : '' }}>^</option>
        </select>
    </div>
</div>
```



After which, we can go back to the **Calculator Controller** and **extend** the logic to suit our needs:

```
switch ($operator) {
    case '+':
        $result = $leftOperand + $rightOperand;
        break;
    case '-':
        $result = $leftOperand - $rightOperand;
        break;
    case '*':
        $result = $leftOperand * $rightOperand;
        break;
    case '/':
        $result = $leftOperand / $rightOperand;
        break;
    //todo: case for an exponentiation operator?
}
```

The possibilities with MVC frameworks are endless. Happy coding!