# **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.

Този документ дефинира пълен пътеводител за създаването на приложение за калкулатор с Symfony Framework  
Уверете се, че сте инсталирали XAMPP и сте добавили коренна директория на PHP към променливата за пътната среда.  
Можете да изтеглите скелета на калкулатора от тук.

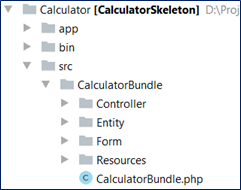
# I. Symfony Base Project Overview

I. Общ преглед на проекта на Symfony

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…)

Symfony е модулна корпоративна уеб-рамка, която се предлага с поддръжка на солидни доставчици, система от пакети, корпоративни механизми и е най-подходяща за архитектура на MVC.  
Първоначално проектът е снабден с основен пакет, който по-късно може да се третира като приставка. Пакетът често има контролер, субекти и свързани компоненти (например хранилища, формуляри, команди ...)



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

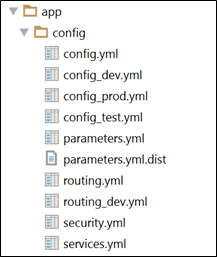
Стандартните шаблони (изгледи) се намират в папката (приложението) и обикновено са разделени в папка, наречена след имената на контролерите.



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.

Де факто стандартният View Engine в Symfony е Twig.  
Основната конфигурация на проекта е поставена в app / config, където са дефинирани конфигурационни файлове за връзката Doctrine, управление на сигурността, правила за маршрутизиране, регистриране на услуги и т.н.



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.

Файлът parameters.yml.dist е много важен, за да съдържат същите клавиши като parameters.yml, тъй като инсталирането на нов пакет ще изтрие неизползваните двойки.

# I. Initial steps

II. Първоначални стъпки  
1. Отворете проекта

## 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**”:

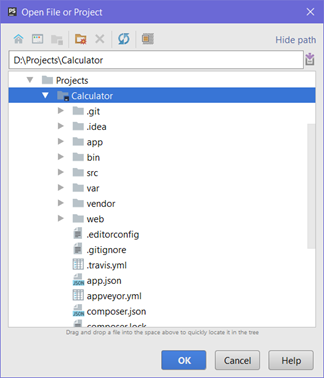
За тази стъпка ще отворим проекта с PhpStorm или IntelliJ Idea. Започвайки от началния екран, кликнете върху "Отвори":



***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**):

Забележка: извлечете папката в кратък път (например D: \ Projects \ Calculator), в противен случай може да се натъкнете на случайни грешки поради операционната система Windows, която има ограничение за дължина на пътя.  
Намерете папката на скелета, която ви дадохме, и изберете папката "Калкулатор" от извлечената папка (например 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.

След като кликнете върху "OK", проектът трябва да започне да се зарежда и индексира. След няколко секунди / минути в зависимост от вашия компютър, вие ще можете да работите с проекта.

## 2. Run the Project

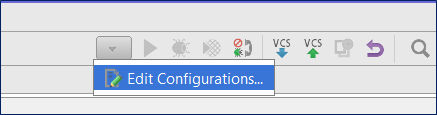
2. Изпълнете проекта

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](http://php.net/manual/en/faq.installation.php#faq.installation.addtopath).

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

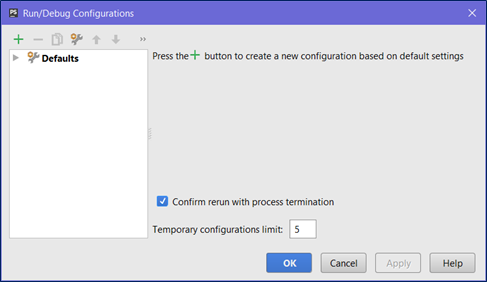
Open the **edit configurations** screen from the **top right** corner of PHPStorm/IntelliJ:

След като отворим проекта, нека се опитаме да го стартираме. Преди да направите каквото и да е, уверете се, че сте задали променливата за PHP коренна папка, така че можете да се обадите на изпълнимия PHP от всяко място. Ако не сте го направили преди да отворите проекта, можете да го направите, като следвате информацията в тази връзка.  
След като сте сигурни, че сте добавили PHP към променливите на средата, е време да създадете конфигурация за изпълнение.  
Отворете екрана за конфигурации за редактиране от горния десен ъгъл на PHPStorm / IntelliJ:



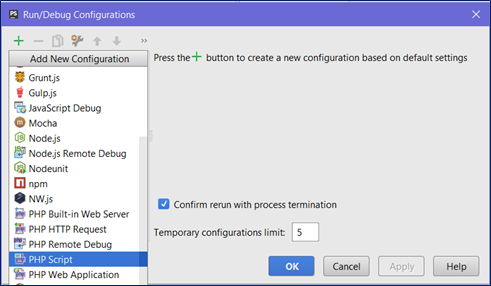
You will be greeted by this screen:

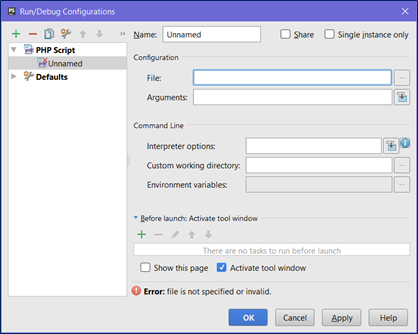
Ще бъдете посрещнати от този екран:



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

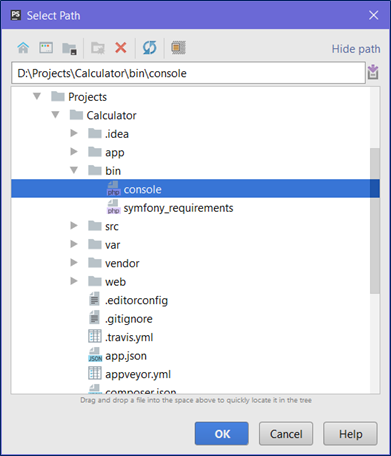
Отворете менюто за добавяне (+) и добавете конфигурация на Run Script PHP:





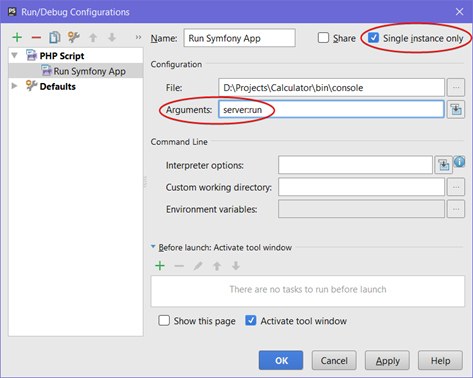
Check the “**Single instance only**” checkbox. After that, point the **File** textbox to the **bin/console** file, which is inside your project directory:

Поставете отметка в квадратчето "Единствено копие само". След това насочете текстовото поле "Файл" към файла "бин" / "конзола", който се намира в директорията на проекта ви:



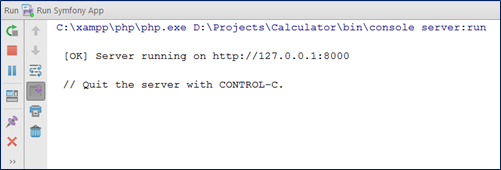
After that, add the “server:run” argument to the **arguments**. In the end, make sure your run configuration looks something like this:

След това добавете аргумента "server:run" към аргументите. Накрая се уверете, че конфигурацията на текущата ви програма изглежда така:



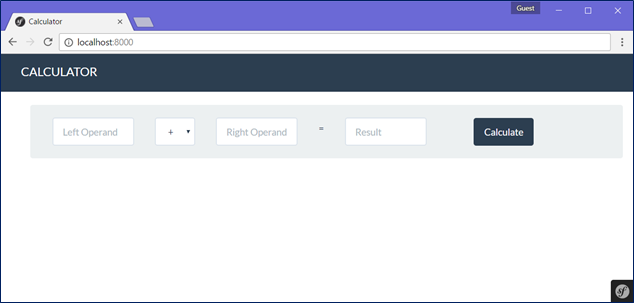
Now, if you attempt to run it by using the play button on the top right, if everything works correctly, you should be greeted by this screen on the bottom:

Сега, ако се опитате да го стартирате, използвайки бутона за възпроизвеждане горе вдясно, ако всичко работи правилно, трябва да бъдете посрещнати от този екран в долната част:



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!

Цялата конфигурация на изпълнението е просто да стартирате командния ред, оцветен в синьо, така че не е нужно да го въвеждате в командния ред всеки път. Ако посетите localhost: 8000 във вашия уеб браузър, ще бъдете посрещнати от приложението за калкулатори!



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

Изглежда страхотно, но не работи. Така че, нека да влезем и да напишем някакъв код, за да накарат тези текстови кутии да си взаимодействат.

# III. Добавяне на функционалност. Adding Functionality

## i. Create the Calculator Entity

i. Създайте елемента на калкулатора

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**.

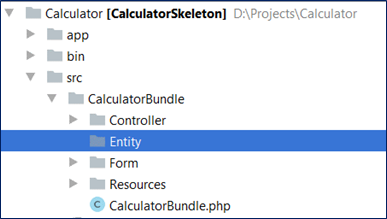
Преди да направим каквито и да било изчисления, ще създадем файл, който ще съхранява данните от исканията, които ще получим, като операндите и оператора. За да направим това, ще определим проста класа и ще я използваме в нашия контролер.

### 1. Create Calculator Class File

1. Създайте класов клас на калкулатора

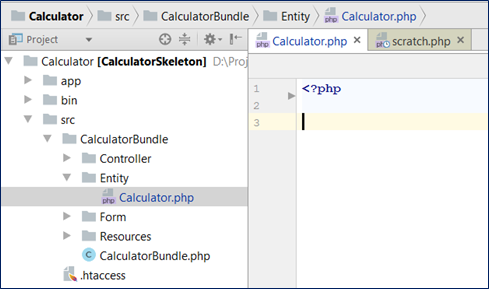
Since we won’t be using a database in this exercise, we **won’t** be using **Doctrine** to generate our entities. Instead, we’ll **make them ourselves**. Head on over to the src/CalculatorBundle/Entity folder:

Тъй като няма да използваме база данни в това упражнение, няма да използваме доктрина, за да генерираме нашите обекти. Вместо това ще ги направим сами. Насочете се към папката src / CalculatorBundle / Entity:



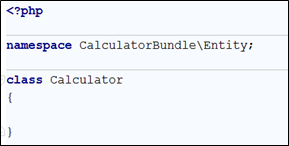
Create a new PHP file inside it, called Calculator.php:

Създайте нов PHP файл вътре в него, наречен 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:

Сега имаме празен PHP файл. Нека я запълним с неща. Добавете the namespace, така че Symfony знае, че това е **entity**. След това направете класа вътре, наречен Калкулатор:



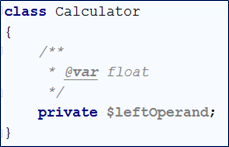
2. Create Fields Създаване на полета

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:

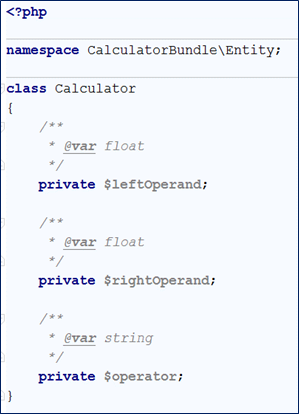
Сега е време да започнете да добавяте полета, които описват нашата организация и свойства, така че да може да бъде достъпен от външния свят. Нашият калкулатор ще има три полета:  
• **leftOperand ->** левия операнд на изчислението. Тя ще има тип данни за плаващ флот.  
• **rightOperand -> десният** операнд на изчислението. Той ще има същия тип като leftOperand.  
• **operator** -> операторът на изчислението (+, -, \* или /). Той ще има тип низ  
Да добавим полето 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**:

Коментарът по-горе всъщност е PHP анотация. Уверете се, че сте го добавили, в противен случай приложението няма да работи правилно.  
Да добавим и другите две полета - дения операнд и оператора:



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 "/").

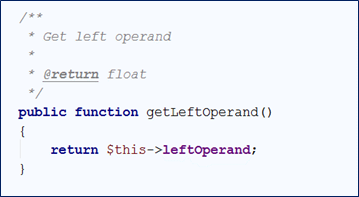
Обърнете внимание, че полето $ оператор има тип низ вместо тип "флоут". Това е така, защото го използваме, за да съхраняваме оператора като низ (например "+", "-", "\*" или "/").

### 3. Create Accessors and Mutators (Getters and Setters)

3. Създаване на аксесоари и мутатори (Getters и Setters)

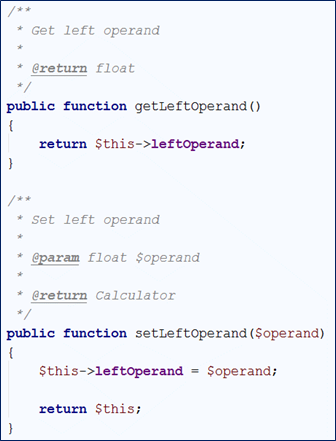
Generally, fields have a **private** access modifier, so nothing can access them from the outside. In order to make other classes able to access it, we need to make what are called **getter** and **setter** methods (also called [Mutator and Accessor](http://www.refulz.com/mutator-and-accessor-methods-in-php/) 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:

Обикновено полетата имат модификатор на частния достъп, така че нищо няма достъп до тях отвън. За да могат другите класове да имат достъп до тях, трябва да се направят така наречените методи на getter и setter (наричани още методи за Mutator и Accessor). Целта на getters и setters е, че тъй като те са методи, ние можем да добавим друга логика вътре, като например валидиране, която може да предпази потребителя от взаимодействие директно с стойността на полето и потенциално да използва този достъп за злонамерени цели.  
Нека първо да направим метода getter за полето 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:

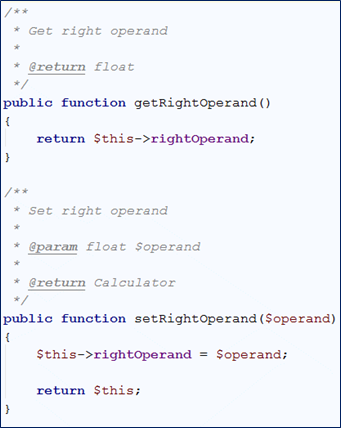
Всичко това метод на getter е връщане на полето от класа. Той не съдържа никаква проверка по своему, но бихме могли да го добавим в бъдеще, просто като вмъкнем логика в метода, преди да върнем полето.  
Нека да създадем и метода за настройка за leftOperand:

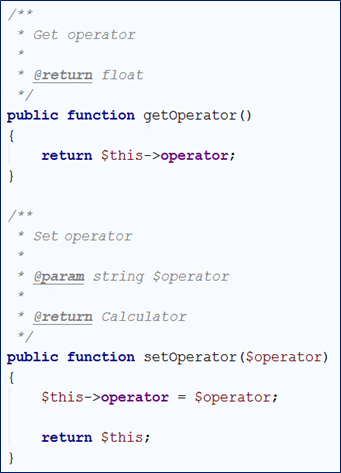


The setter takes one $operand as a parameter, sets the leftOperand field to the parameter’s **value** and **returns** the object itself. This will be the blueprint for all classes we make from now on (with different fields and mutators/accessors, of course). We can also further extend the class by adding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

Настройващото устройство задава параметър $ операнд, настройва полето leftOperand на стойността на параметъра и връща самия обект. Това ще бъде планът за всички класове, които правим оттук нататък (разбира се, с различни полета и мутатори / принадлежности). Също така можем да разширим класа чрез добавяне на функции, които работят на класа и т.н. Но повече за това по-късно ...  
Засега нека просто създадем останалите методи на getter и setter - за полетата rightOperand и operator:





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

Завършихме с нашата калкулаторна единица, така че нека да продължим към изпълнението на действието, което прави приложението да работи.

## ii. Implement the Calculate Action

ii. Прилагане на действието "Изчисляване"

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.

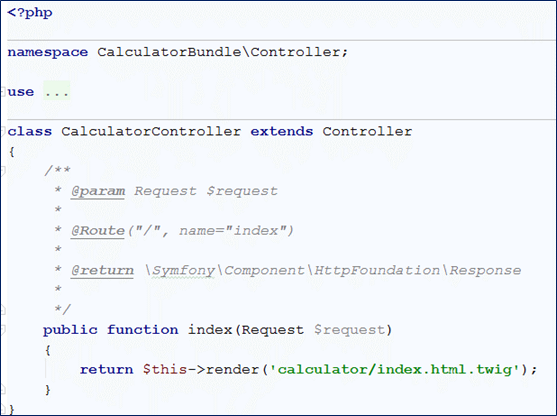
Сега, когато имаме действителен клас, за да поставим нашия ляв операнд, операнд и оператор вътре, трябва да предприемем действие, което може да ги вземе и да направи реално изчисление. Ще направим това, като променим контролера на калкулатора, така че да отговаря на нашите нужди.

### 1. Create a Form In-app

1. Създайте формуляра в приложение

Let’s go into the CalculatorController.php:

Нека да отидем в CalculatorController.php:



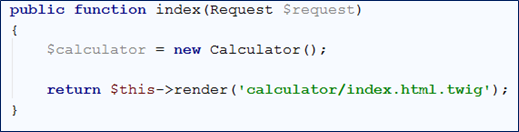
Looks pretty empty at the moment. All we have is one function, called index, which returns the **index** **view**. No calculation going on here. Let’s make it work like a calculator!

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:

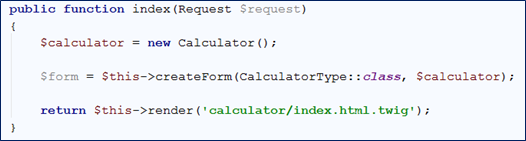
Let’s start by creating a **calculator variable** where we’ll **store** our **operands** and **operator**:

Изглежда доста празен в момента. Всичко, което имаме, е една функция, наречена индекс, която връща изгледа на индекса. Тук няма никакви изчисления. Нека да станем като калкулатор!  
Отговорът на HTTP, който даваме на клиента, работи по следния начин: Когато клиентът получи отговор, двигателят за изобразяване на уеб браузъра го предава на екрана си, превръщайки HTML, давайки на клиента пълна уеб страница.  
Ще редактираме функцията за индекси, за да използваме формуляра, който се изпраща от клиента и да използваме стойностите, които потребителят ни е изпратил чрез заявката за POST, като кликнете върху бутона "Изпращане". За да направим това, трябва първо да потвърдим, че формулярът се изпраща изобщо:  
Нека да започнем, като създадем променлива на калкулатора, където ще съхраняваме операндите и оператора:



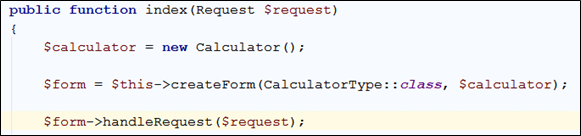
We’re not quite done yet. We have a bit more work to do. Next, we need to create a **form** variable, which will create a **special token** for the user and also, more importantly, **take** the **values** from **the form the user sent** us, and stick them in the $calculator variable, so we can work with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

Още не сме приключили. Имаме още малко работа. След това трябва да създадем променлива на формуляра, която ще създаде специален символ за потребителя, а също и по-важното - да вземе стойностите от формата, който потребителят ни е изпратил, и да ги залепи в променливата "$ calculator", за да можем да работим с тях:



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:

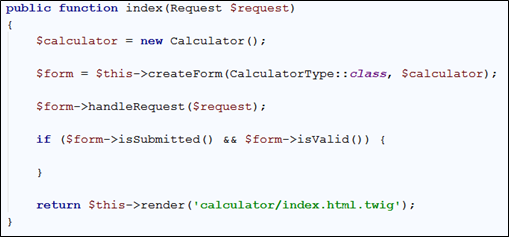
След това, преди да приложим логиката, за да проверим дали това, което потребителят ни изпрати, е валидно, трябва да обработваме искането. Правим това с метода $ form-> handleRequest ():



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):

След това трябва да проверим две неща:  
1. Потребителят ни изпрати формуляр изобщо  
2. Потребителят ни изпрати валиден формуляр (формуляр с два операнда и оператор се счита за валиден):



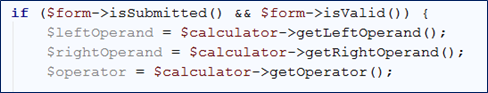
After that, we can be sure that if we got in-between those two parentheses, the user sent us something we can actually work with, and the only thing which remains is to **implement** the actual calculator logic.

След това можем да сме сигурни, че ако имаме между тези две скоби, потребителят ни изпрати нещо, с което действително можем да работим, и единственото, което остава, е да приложим действителната логика на калкулатора.

### 2. Изпълнете логиката на калкулатора. Implement Calculator Logic

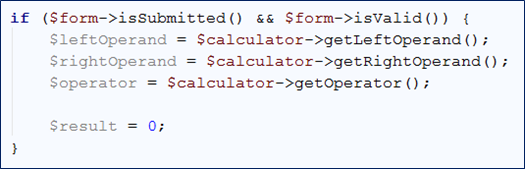
If our form was **submitted** and **valid**, Symfony automatically **inserts** the form values into our $calculator variable. So we can now use that to implement the calculator logic. Let’s start by **extracting** our **operands** and **operator** into variables for easier typing:

Ако нашият формуляр е изпратен и валиден, Symfony автоматично вмъква стойностите на формуляра в нашата променлива $ калкулатор. Така че сега можем да го използваме, за да приложим логиката на калкулатора. Нека започнем като извадим операндите и оператора си в променливи за по-лесно написване:



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

След това ни трябва някъде да съхраняваме резултата, нали? Точно така. Така че нека направим променлива за това:



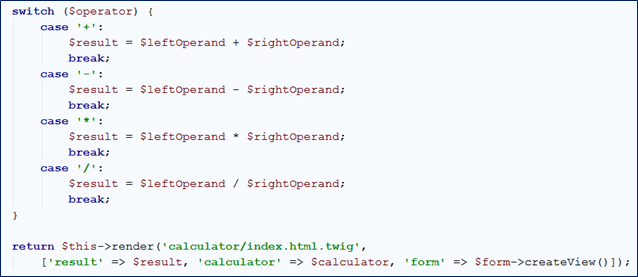
After that, let’s implement some calculator logic by using a **switch case** on the **operator**:

След това нека изпълним някаква логика на калкулатора с помощта на ключа за превключване на оператора:



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**:

Почти там ... Сега, след като приложим логиката, не трябва ли тази логика да донесе някакъв резултат от нашето малко уеб приложение? Отговорът е "да". След валидно изчисление - ако всичко върви добре, трябва да върнем HTTP отговор на потребителя с изчислената стойност:



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:

Нека разбием тази декларация за връщане:  
• $ this-> render () указва на контролера кой view да се върне.  
• Функцията render () приема два параметъра:  
o низ, показващ изгледа, който трябва да се върне (в нашия случай - калкулатор / index.html.twig)  
o Асоциативен масив, посочващ данните, които предоставяме на изгледа. В нашия случай това би било:  
 стойността на резултата  
 самия калкулатор (така че да можем да задържим операндите и оператора между заявките)  
 формуляра, който ще създадем за потребителя със специалния си символ ($ form-> createView ())  
Забележка: Уверете се, че това връщане е в полето if, което проверява дали е изпратен валиден формуляр. Няма смисъл да връщаме резултат, ако не ни бяха дадени валидни данни, за да се изчисли, нали?  
В този момент цялата наша логика се изпълнява правилно, но това няма да стане. Защо???  
Причината е, че преди да приемем формуляр от потребителя, трябва да създадем формуляра, като използваме това специално означение, за което говорихме по-рано. Ако нямаме token, за да започнем в html, нашият "е валиден ли е този формуляр?" Проверката ще се провали. Ето защо последното нещо, което трябва да направим, е да редактираме декларацията за връщане, защото нямаме форма за обработка:



**↓**



## iii. \* Play Around with your Calculator App

iii. \* Играйте наоколо с приложението си за калкулатор

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

След като сте въвели основната функционалност, опитайте се да приложите някои допълнителни функции сами.

### 1. Add a New Operator

1. Добавете нов оператор

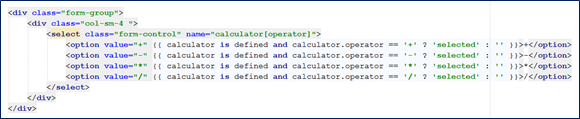
Some ideas for extra functionality include:

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

Някои идеи за допълнителна функционалност включват:  
• Изпълнение на експоненцията (4 ^ 2 = 16 и т.н.)  
• Битови операции (OR, AND, XOR и т.н.)

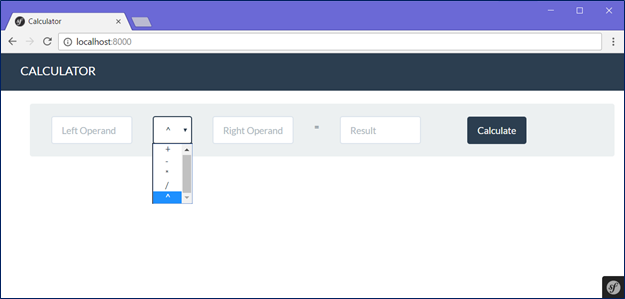
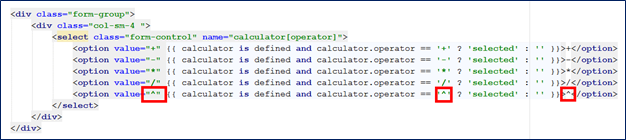
Hint: to add an operator, you must edit the form in the index.html.twig template, located in the app/Resources/views/calculator/index.html.twig path inside your project. There you can find the part of the form, which is responsible for listing the operators:

Съвет: За да добавите оператор, трябва да редактирате формуляра в шаблона index.html.twig, разположен в пътя на приложението / Ресурси / изгледи / калкулатор / index.html.twig във вашия проект. Там можете да намерите частта от формуляра, който отговаря за включването на операторите:



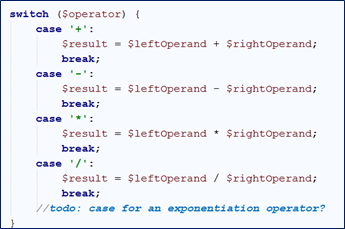
It has some extra logic inside each <option> tag, in the form of **twig syntax**, which will **select** the **last used** operator when transitioning between calculations. In order to add an operator, we can just copy one of the <option> tags and edit it to suit our needs:

Той има някаква допълнителна логика във всеки маркер <option>, под формата на синтекст на клони, който ще избере последния използван оператор при преход между изчисленията. За да добавим оператор, можем просто да копираме един от маркерите <option> и да го редактираме според нуждите ни:



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

След това можем да се върнем към контролера на калкулатора и да разширим логиката, за да отговорим на нашите нужди:



The possibilities with MVC frameworks are endless. Happy coding!

Възможностите с MVC рамки са безкрайни. Честит кодиране!