Computer & Systems Engineering Department

CSE 223: Programming 2

Assignment 2

Name: Mahmoud Attia Mohamed ID: 20011810

Problem Statement:

Build a web-based calculator similar to that of windows. The buttons should be web buttons.

- No need for fancy styling.
- The calculation should be done on the server side.
- For simplicity, you can ignore the difference between the C and CE buttons.
- Repeating pressing the = button does not issue new calculations Handle exceptions such as dividing by 0, by displaying an E.

Technology used:

For the frontend part (view part), we used HTML, CSS, and typescript throw the angular framework.

For the backend (model and controller), we used Java language throw the spring framework.

How to run the project:

- extract the compressed project folder.
- Open the spring file on spring boot using IntelliJ ide, there will be 3 classes in the package, run them on port 8082 or you can change the

port from the project resources → application.properties if the 8082 port was already used in your device.

- Open the angular project file using visual studio ide, then open the terminal of the ide, and write npm install in the terminal.
- Then write “ng serve --open” in the terminal to open the project, on port “http://localhost:4200/”.
- Then you can use the calculator as you want.

Note: Make sure you downloaded NodeJs and Angular-CLI.

Assumptions and features:

- If the user didn’t enter the first operand it will default to "zero", also if the user didn’t enter a second operand and he clicked equal, the second operand will be considered the same as the first operand.
- If the user enters the first operand and then enters a binary operator and wants to change this operator before entering the second operand he/she can change the operator easily by clicking on the binary operator he/she wants.
- The user can perform multiple operations by easily clicking on the next operator button after ending the first calculation without clicking on the equal button.
- I made the functionality of the (%) button as it is in the windows calculator (previous operand * (current operand/100):
  - If the user didn’t enter any binary operator like 2% , the previous operand will be 0 and the result will be 0 * 2% = 0.
  - And if 2 + 3% so the result will be = 2 * 3/100.
- I handled errors as it is in windows calculator:
  - If the user tries to divide by zero or take the root of a negative number or try to calculate a process its result will exceed the limit of numbers that can be represented by my system (Overflow) I print an error message to him.

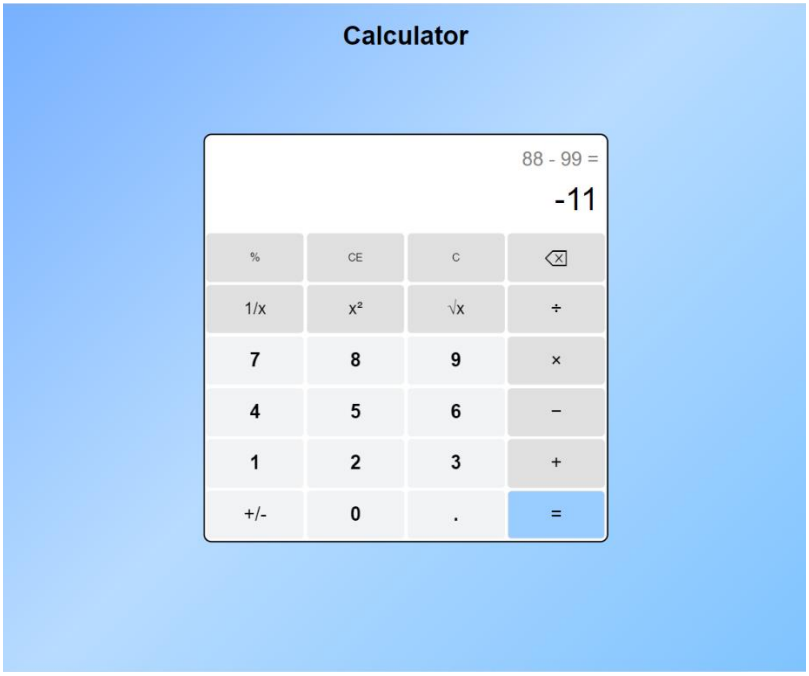
    - After printing the error message, I clear the calculator and reset the system so the user can start using it again.
- I neglected the difference between C and CE buttons as it was declared in the assignment, so I made both clear everything and start from the beginning.
- I assumed that the maximum number of digits that the user can input is 14 digits.
- Repeating pressing the "=" button does not issue new calculations.

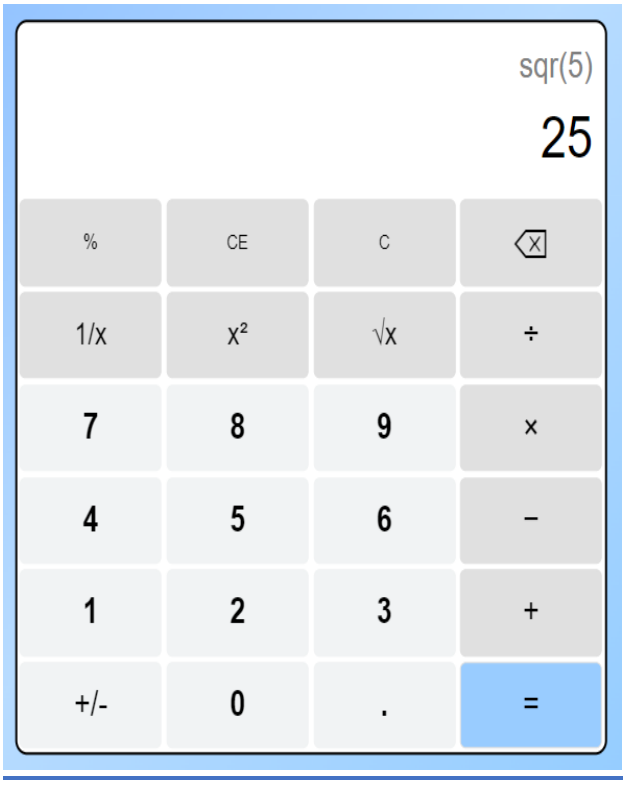
How to use:

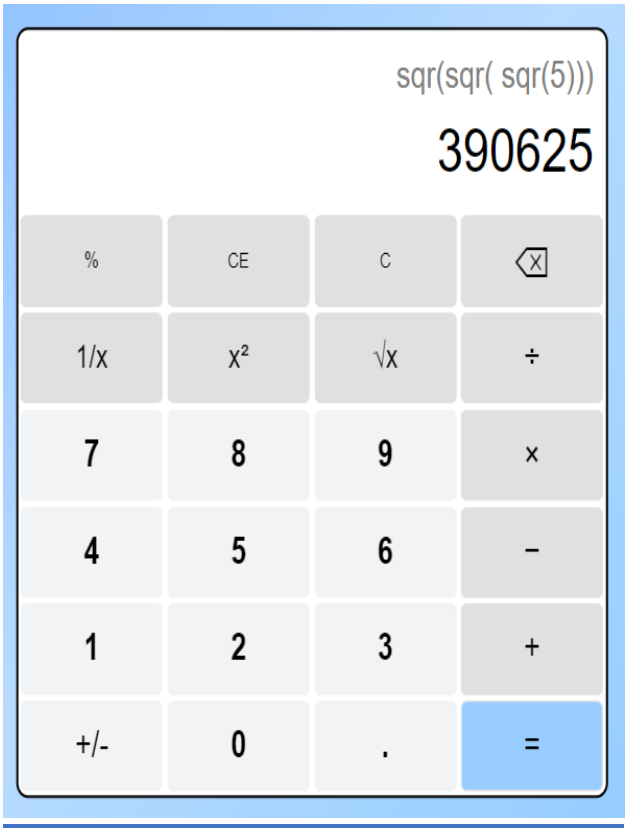
- Simply you can use this programmed calculator as any other calculator but you should follow the assumptions I declared previously.
- This is a link to a video I provided to show how to use the calculator: how to use video link- click here

---

Calculator snippets:




sqr(5)
25
%
CE
C
1/x
x²
√x
÷
7
8
9
×
4
5
6
−
1
2
3
+
+/-
0
.
=


sqr(sqr(sqr(5)))
390625
%
CE
C
1/x
x²
√x
÷
7
8
9
×
4
5
6
−
1
2
3
+
+/-
0
.
=

About the code design style:

I used **Model-View-Controller (MVC)** style to design this program:

- The **View** was represented by the frontend part which is used to view the buttons and the result.
- The **controller** was represented by the CalculatorController class in the backend which receives the request from the view part and sends it to the model and then receives the result from the model and sends it back to the View.
- The **model** was represented by the calcService class in the backend which is used to calculate our result.

I used this model so if I need to change the view of the calculator, I won't change any of the logic of the program and if I need to change the logic of any function I won't change the view of the calculator.

Frontend components:

- **app.components.html:**
  this is the html file for the components of my program. Here I wrote the display part and the buttons part and organized them together.
- **app.components.css:**
  this is the css file for the components of my program. Here I styled the display part and each button in my program.
- **app.components.ts:**
  - this is the typescript file for the components of my program. Here I made each listener to each button and took the text content of each button so I can handle it to form the expression that should be sent to the backend to evaluate its value and return it back to the front.

- This file contains the **sendExpression function**:
  - It takes the first and second operands and the operator between them and then uses the **evaluateOp function** (that we will discuss in the services part) to send them to the backend
  - Then take the returned result from the back to use it in the front either to display or to be used in any other calculation.

```
constructor(private calcObj: calcServer) { };

public sendExpression(op1: string, op2: string, operator: string): void {
  this.calcObj.evaluateOp(op1, op2, operator).subscribe({
    next: (x) => {
      if (x == null) {
        this.errorFlag = true;
        if (operator == "div" || operator == "dx") {
          this.curr = "Cannot divide by zero";
        } else if (operator == "root") {
          this.curr = "Invalid input";
        } else {
          this.curr = "Overflow";
        }
        this.prev = "";
        this.operator = "";
      } else {
        this.curr = x;
        this.prev = x;
        this.sc = x;

      }
    }
    ,
    error: (error: HttpErrorResponse) => {
      alert(error.message);
    }
  })
}
```

- **<u>app.services.ts:</u>**
  Here is the **calcServer** class which contains the **evaluateOp** method:

- This method takes the first and second operands and the operator from the frontend and sends them to the backend using the **post** method of the **http** object of the **HttpClient** class:
    - This method takes the port link of the spring server, operands, and the operator
    - then receive the returned observable using and parse it into a string to use it in the frontend.

```
import { Injectable } from '@angular/core';
import { Observable } from "rxjs";
import { HttpClient } from '@angular/common/http';
import { environment } from "src/environments/environment";


@Injectable({
    providedIn: 'root'
})
export class calcServer{
    constructor(private http: HttpClient){}

    public evaluateOp(op1: string, op2: string, operator: string): Observable<string>{
        return this.http.post<string>(`http://localhost:8082/evaluate/${op1}/${op2}/${operator}`, JSON);
    }

}
```

- **app.modules.ts:**
  here I imported the **HttpClientModule** and add it to the imports.

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import {HttpClientModule} from '@angular/common/http';


@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    HttpClientModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }

```

Backend classes:

- **CalculatorLabApplication class:**
  Which contains the main method

```java
package com.example.calculator.lab;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;


1 usage
@SpringBootApplication
public class CalculatorLabApplication {

    public static void main(String[] args) {
        SpringApplication.run(CalculatorLabApplication.class, args);
    }

}
```

- **<u>CalculatorController class:</u>**
  - Here I used **crossOrigin** to specify the origin port of the front end to receive the sent message.
  - Then I used **postMapping** to map the sent message to the 2 operands and the operator.
  - Then I used **pathVariable** to pass the 2 operands and the operator to the **handler function**.
  - The **handler function** contains an object of the **calcService** class and uses its method **getResult** to calculate the value of the expression and return it to the front end as a string.

```java
package com.example.calculator.lab;
import org.springframework.web.bind.annotation.*;


@RestController
@CrossOrigin(origins = "http://localhost:4200")
public class CalculatorController  {
    @PostMapping(value = "/evaluate/{op1}/{op2}/{operator}")

    public String handler(@PathVariable("op1") String op1, @PathVariable("op2") String op2,
                          @PathVariable("operator") String operator ) {
        final calcService calc = new calcService();
        return calc.getResult(op1, op2, operator);
    }
}
```

- **calcService class:**

  which contains the **getResult** function which takes the first and second operands and the operator and checks if there was an error or not:
  - if there was an error it returns null.
  - Else it calculates the result of the operation and returns the result as a string.