Big Number Calculator - CSS Project

by Cioată Matei, Rezmeriță Mihnea,
Turcu Ana-Maria, Vararu Cristian

1. Introduction

This application is a big number calculator, meaning that it can perform basic, complex and compound operations with large operands.

The application consists of 2 components: a restful API, that can be called from everywhere, and a web client that offers an enjoyable user experience.

The restful API was built using Java, specifically Springboot framework, running on the embedded Tomcat server.

The restful API can be built using the simple **mvn clean package** command followed by a **java -jar bigNumber.jar** in order to run the application.

On the front-end side, we used Angular in order to build a single web page application that communicates with the restful API, mentioned above.

2. Contributions

a. Rezmeriță Mihnea:
- Addition service;
- Full project architecture over the rest API, including class design or exception handlers;
- Front-end client.

b. Turcu Ana-Maria & Vararu Cristian:
- Subtraction service;
- Multiplication service;
- Division service;
- Power service;
- Square root service.

c. Cioată Matei-Alexandru:
- Validating input expressions;
- Parsing and evaluation of string expressions;
- Parsing and evaluation of XML expressions.

3. Operations

a. Exceptions

Of course, there are different types of exceptions that might occur during the execution, but the application has its own set of custom exceptions that are thrown, like: SquareRootFormatException or InvalidNumberException. These exceptions will be caught by a global exception handler, that will return a specific status code and message to the client.

b. Validators

The rest API has its own set of validators for numbers, in order to ensure a good user experience but also a good execution.

Different type of validations are performed before the actual operation is executed:

i. Checking if the numbers are present
ii. Checking if the first digit is correct
iii. Checking if the number actually contains only digits

If one of these conditions fail a custom exception will be thrown

c. Addition Service

This functionality is implemented in the class AdditionCalculatorService.java.

The AdditionService performs the addition operation between 2 big numbers, obviously. The 2 numbers are received as strings.

As a preliminary step, the algorithm reverses these 2 strings and swaps them, if needed, so that the second number is the smaller one according to the string length.

After this step, we add each character from the 2 strings(basically, each digit of the first number is added with the corresponding digit of the second number), and that is the new digit from the result, while taking care of the digits that might be bigger than 10(which are no longer digits).

The result is reversed and returned to the controller.

d. Subtraction service

This functionality is implemented in the class SubstractionCalculatorService.java.

In order to use the subtraction operation you will have to put 2 big numbers as input. The numbers are received as strings.

The operate method that is called from the Controller firstly checks if the 2 numbers are valid and also if the second number is smaller than the first one due to the fact that the application does not support subtraction with negative result, otherwise an exception will be throwed.

Also, as in the addition case, the numbers are firstly reversed and after that the operation is performed. In this way the subtraction is performed digit by digit starting from the last digit of the numbers. Also the case when we have to “loan” from the next unit is implemented.

The result is reversed and returned to the controller.

e. Multiplication service

This functionality is implemented in the class MultiplicationCalculatorService.java.

As the addition and subtraction, this operation takes 2 big numbers as input. The numbers are received as strings.

The operate method that is called from the Controller firstly checks if the 2 numbers are valid and after that it does a swap between the numbers in the case that the first number is

shorter than the second one. This helps our algorithm because we will compute as many simple multiplications as digits are in the second number, so as shorter is the second number as faster will be the calculation process.

After this step, the numbers are reversed and the operation is performed. The multiplication is performed as follows: we take the digits from the second number one by one starting from the last one and multiply them with the first number's digits (taking into account also the carry if it exists). These results are shifted to right with as many zeros as is the unit order for the second number's digit that was used in this step and after that added one by one to the result using the addition service.

The result is reversed and returned to the controller.

f. Division service

This functionality is implemented in the class DivisionCalculatorService.java.

This operation, as the previous ones, uses 2 numbers as input. Here, the dividend is a big number, but the divisor may be an integer or a big number and these 2 cases are treated differently. The numbers are received as strings.

The operate method that is called from the Controller firstly checks if the 2 numbers are valid, if the second number is not 0 because the division by 0 is not supported and if the second number is smaller than the first one, otherwise 0 is directly returned. We implemented the division returning only the int part of the result, not also the rest of the division.

For the actual performance of the result we separate the algorithm into 2 different functions by the second number given as parameter, the divisor. If it can be cast to an int value then we call the performDivisionForSmallDivisor function, otherwise we will use performDivisionForBigDivisor. The second variant is quite slow because it implements the division as repeated subtractions, so that's why we chose to perform in another way for the small numbers. We used our implementation for subtraction in the big divisor cases.

The result is reversed and returned to the controller.

g. Power service

This functionality is implemented in the class PowerCalculatorService.java.

In order to use the power operation you will have to put 2 big numbers as input. The first one represents the base and the second the exponent for this operation. The numbers are received as strings.

The operate method that is called from the Controller firstly checks if the 2 numbers are valid and after that the operation is performed.

For this operation we optimise the classical way that uses repeated multiplications to compute the power with an Divide et Impera algorithm. By default the result for this operation is 1 so we also cover the cases with 0 as exponent. Also, at this operation we used our implementation for multiplication and division.

The result is reversed and returned to the controller.

h. Square root service

This functionality is implemented in the class SquareRootCalculatorService.java.

The square root operation is the only one implemented in this application that takes only one big number as input. The numbers are received as strings.

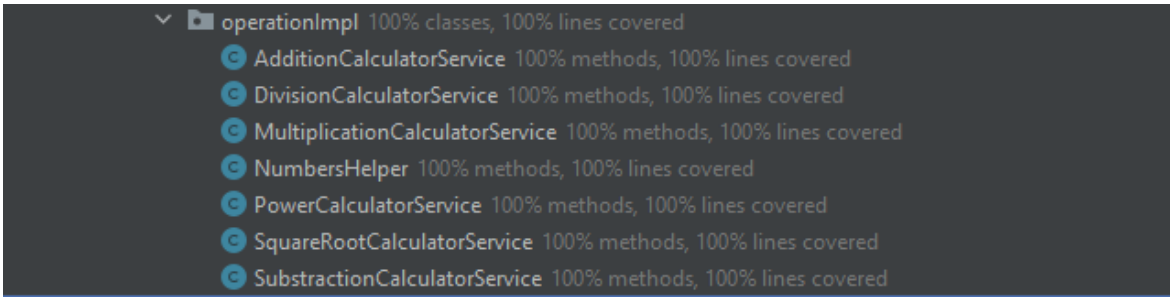
The operate method that is called from the Controller firstly checks if the number given as parameter is valid and after that the square root is calculated.

We implemented this operation also in a Divide et Impera manner, trying to find the number that multiplied by itself will give a result equal to input. For numbers that are not perfect squares we return the closest perfect square that is smaller than the input. For implementing this operation we used our implementation for addition, subtraction, multiplication and division.

The result is reversed and returned to the controller.

i. Unit testing for operations

For the operation services we wrote unit tests in order to cover all the functionalities implemented, including corner cases. Code coverage for this module can be see below:



Tests were organised into test classes for each operation and named in order to suggest the test scenario, for example:

```
10  public class AdditionCalculatorServiceTest {
11
12      private final AdditionCalculatorService additionCalculatorService;
13
14      public AdditionCalculatorServiceTest() {
15          this.additionCalculatorService = new AdditionCalculatorService();
16      }
17
18      @Test
19      public void operate_whenNumbersAreOfSameOrder_NoCarry() {
20          OperandsBody givenOperandsBody = new OperandsBody("1", "1");
21
22          additionCalculatorService.operate(givenOperandsBody);
23
24          assertThat(additionCalculatorService.operate(givenOperandsBody)).isEqualTo("2");
25      }
```

Tests have a simple structure, respecting the arrange - act - assert pattern.

We have taken into account multiple possible inputs including cases when exceptions might be thrown and we also checked if the type of those exceptions is as expected:

```
@Test
public void operate_secondNumberIsBigger_thenThrowNegativeResultException() {
    OperandsBody givenOperandsBody = new OperandsBody("1", "10");
    expectedEx.expect(InvalidNumberException.class);
    expectedEx.expectMessage("Negative Result");
    substractionCalculatorService.operate(givenOperandsBody);
}
```

j. Assertions for operations

In this project we also add some validations of inputs or processed values using assertions. We added both preconditions and postconditions for all the operations and auxiliars components (validators).

In this scope we added asserts to check if the result is longer/shorter or equal in the number of digits with the values that are given as input, taking into account the operation's logic. For example if we add 2 numbers that have 4 digits we have to obtain a result with at least 4 digits. We put those postconditions right before to return the result that have to be returned:

```
assert result.length() >= firstNumber.length : "Result should be grater then first number";
assert result.length() >= secondNumber.length : "Result should be grater then second number";
return result;
```

Beside this we also added postcondition asserts to check if the swap function works as expected.

As preconditions it was checked in the validators if the parameter given was not null:

```
public static void validateDivisionBy0(String number) {
    assert number !=null : "Number is null!";
    if(number.charAt(0) == '0' && number.length() == 1){
        throw new InvalidNumberException("Division by 0!");
    }
}
```

4. Expressions

The application accepts expressions as input, returning the correct result together with all the steps executed in the calculation process. In order to get to the final result, an expression must go through several processes:

- parsing it (string of XML format) in order to compute a token list;
- validating the token list;
- building a binary expression tree;

- evaluating the expression tree in order to obtain the result along with the intermediary steps.

The program presents two different service implementations for expression processing: **ExpressionCalculatorServiceImpl** (handles the string format inputs) and **XmlExpressionCalculatorServiceImpl** (handles the XML format inputs). The only differences consist of the parsing method and the result format (JSON for string input and XML for XML input. After obtaining the list of tokens, the implementation of the other steps is reused.

In the rest of this chapter, these operations will be explained from a detailed point of view.

a. Parsing string expressions

The string expression is traversed character by character. The parsing algorithm looks for valid expression tokens and retains them in a list that is returned at the end of the function execution. The following items are considered valid tokens:

- operators: '+'. '-'. '*', '/', '^' and the sequence 'sqrt';
- operands: any sequence of digits containing at least one character;
- parentheses.

Invalid characters (letters or other symbols) are ignored.

In the case of sequences (numbers or 'sqrt'), the token must be built of several characters.

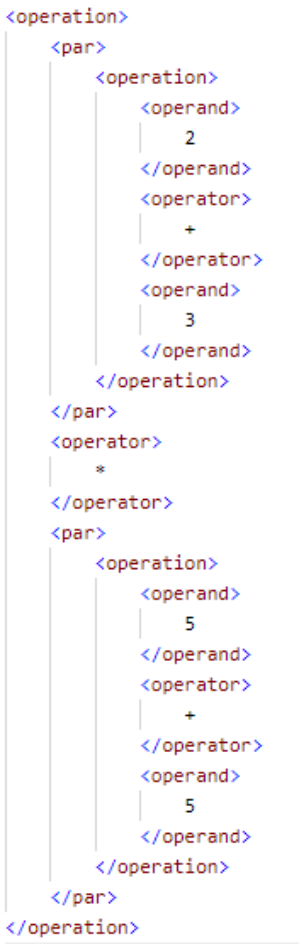
For example, the expression "sqrt (52 + 1928)" contains five tokens: 'sqrt', '(', '52', '+', '1928'.

b. Parsing XML expressions

The tokens obtained from XML inputs are the same as the ones obtained from string inputs. The parsing algorithm eliminates spaces and indentations from the input and then looks for the following XML tags:

- <par> - open parenthesis;
- </par> - closed parenthesis;
- <operator> - marks an operator token;
- <operand> - marks a number;
- <operation> - must contain an operator and operands (or a single operand for square root);

Example: for the following input, the token list will be: '(', '2', '+', '3', ')', '*', '(', '5', '+', '5', ')'



c. Building the expression tree

In order to calculate the expressions, the program builds a tree based on the list of tokens with the following rules:

- numbers are located in the leaf nodes. After setting the value, the algorithm returns to the parent node;
- open parenthesis marks a left child creation (and movement to it);
- closed parenthesis returns to the parent of the current node;
- operators set the token value in the current node, then the algorithm moves to the right child;
- the left child of a node tagged with ‘sqrt’ is NULL. The right child represents its only operand.

d. Evaluating the expression tree

The tree is evaluated recursively. Its nodes have either operators or operands (numbers) as values. When an operator is encountered, it is applied between the result of the left subtree evaluation and the result of the right subtree evaluation. Therefore, this recursive method performs the operations from bottom to top and writes each step in a list that will be sent to the client.

e. Validating input expressions

The validation step is working with the list of tokens resulted after the parsing operation. The following checks are performed:

- parentheses closed before opening;
- number of open parentheses is not equal to the number of closed parentheses;
- the expression in its entirety must NOT be surrounded by parentheses;
- operation order is not defined by parentheses.

f. Unit tests for expressions

For the unit testing phase, we tried to check all of the functionalities described above (string parsing, XML parsing, tree building and tree evaluation). The following cases have been treated:

i. String parsing (verifying that the token lists are extracted correctly in multiple cases):

- simple expression with 2 operands (e.g., '5 ^ 2') ;
- simple expression with one operand (e.g., 'sqrt 9');
- expression with parentheses (e.g., '(4/2)+1');
- expression with wrong characters in its structure, to check that they are ignored (e.g., 'asb10 -(%&5 @+ !@2)').

ii. XML parsing (verifying that the token lists are extracted correctly in multiple cases):

- simple expression with 2 operands;
- simple expression with one operand;
- expression with parentheses;
- expression with wrong characters in its structure, to check that they are ignored (this unit test fails, the XML parsing algorithm does not ignore incorrect characters).

iii. Building expression trees (checking that expected behaviour is reproducing in different cases):

- behaviour for 'sqrt' operator (having one operand);
- behaviour for operation with 2 operands;
- behaviour when encountering parentheses.

iv. Evaluating expression trees (checking that correct results and steps are obtained):

- one unit test per type of operator ('+', '-', '*', '/', '^', 'sqrt').

v. Validating token lists:

- each validator is checked if it throws the right error;
- no error should be thrown if the input is correct;

vi. Testing the entire flow:

- string input returns correct result AND steps in JSON format;
- XML input returns correct result AND steps in XML format.

g. Assertions for expressions

Assertions have been added into the development phase code in order to assure that certain preconditions / postconditions / invariants for our methods hold as expected:

- expression input is not null;
- parsers detect only valid tokens (using regex matching);
- XML parser eliminates indentations as expected (using regex matching);
- list of tokens is not empty after parsing;
- there is no NULL token in the list;
- expression tree is initialized as expected;
- building of expression tree stops when reaching the end of the token list;
- left child of 'sqrt' operator is NULL in expression trees;
- 'else' blocks handle the expected execution branches;
- the evaluated tree's root is not NULL;
- operation result is not empty and contains only numbers;
- list with all of the expression calculation steps is populated as expected.

5. Front-end

As stated above, there is also a web client that communicates with the rest API, providing a nice way to use the application.

The web client was built using Angular 6, making use of its components, styled using CSS.

Each displayed component is, in fact, just an operation. So, the user will enter his numbers/expression and will get his result. From a technical point of view, when the user requests the result the application makes a HTTP call to the rest API, mentioned above.

When trying to post an expression, the application will also display the intermediary steps taken in order to calculate the final result, steps that are received from the rest API.

There are also some validations on the client side, just so the user won't send empty or invalid data. If something is wrong with the request a specific message will be displayed.

6. Conclusions

Building the application we manage to combine an intuitive user interface with the functionalities of a simple computer, but for big numbers.

The application is capable to compute complex operations that were implemented using expressions by parsing the input and building the expression's tree and after that step by step computing the result using the simple operation services. It supports both string and XML inputs and it also returns the intermediary steps through which we obtained the result.

Invalid inputs or other logical exceptions were treated and relevant messages were sent as results in order for the user to understand that something went wrong.

All functionalities were tested through unit tests and also assertions were used for checking postconditions and precondition on both operations and expressions modules.