

Big Calculator

Project documentation

Drăgoi Ștefan (ISS1)

Cunev Serghei (ISS1)

Condurache Andrei (ISS1)

Tiperciuc Ștefan (ISS1)

1. INTRODUCTION

Big Calculator is a web application which allows the user to easily make calculations on very large numbers. Users are able to make various computations of mathematical equations using basic operations (such as addition, multiplication, power etc.) or parentheses to assign operation order and the application will output the result along with each step of the computation. The calculations can be performed in two different ways:

- Interactive mode, using a simple User Interface to insert the values of the variables and of the operations
- Automatic mode, which reads the inputs from an XML file and computes the result

In order to help with the modularity of the code, we divided our application in two large projects, one for back-end and one for front-end, making it more easily testable. The back-end was written using .NET 6 and the front-end was made using JavaScript framework Vue.js, that allowed us to build a fast and responsive user interface as well as test it with the built in testing library Jest. It also followed our requirement of modularity, because every small UI part is a different component, therefore everything is modular, easy to reuse and, in our case, to test.

Considering the fact that the main objective for making this project was to perform various testing techniques, unit tests and assertions have been performed. Unit tests were made in the BigCalculator.UnitTests module using XUnit and cover 100% of the lines of code by testing various functionalities of the applications. Assertions were done using the System.Diagnostics library to validate various invariants, preconditions and postconditions of the operations implemented.

In the following documentation we will be presenting how to run the application ([Section 2](#)), the architecture of the projects ([Section 3](#)), the main modules present in the code ([Section 4](#)), the unit tests performed ([Section 5](#)), as well as the assertions made

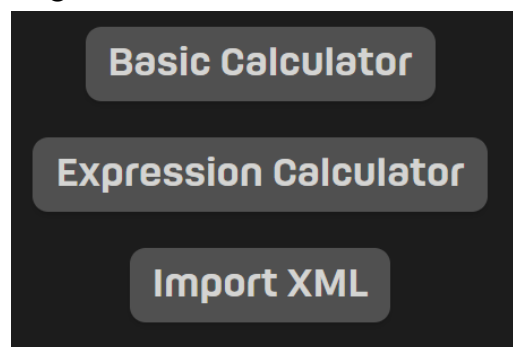
([Section 6](#)). The report will end with detailing the contribution of each member of the team ([Section 7](#)).

2. HOW TO RUN

2.1. Running app/tests on Front-end

In order to run front-end applications and tests we will need Node.js. After we've installed it, with the help of node package manager or yarn we need to install Vue CLI, by writing in cmd ***npm install -g @vue/cli***. When the process of installation is finished, we can run the following line in cmd ***vue ui***, which will open a project manager browser page where we can import the folder BigCalculator-frontend from the git repository. Once imported, it will open the project dashboard with a menu, which will have a link to tasks. In tasks we will find the following tasks we are interested in :

- ***serve*** - the task that runs our application, when we will run this task it will open our front-end application. The application will be available in the browser with the following address : localhost:8080/. On this link we will have the following menu, with the modules that will be described in later.



- ***test:unit*** - the task that is responsible for running unit tests with Jest, it will automatically search for all .spec files and run the tests inside them, and after the tests are run it will show us the results as well as a code coverage table.

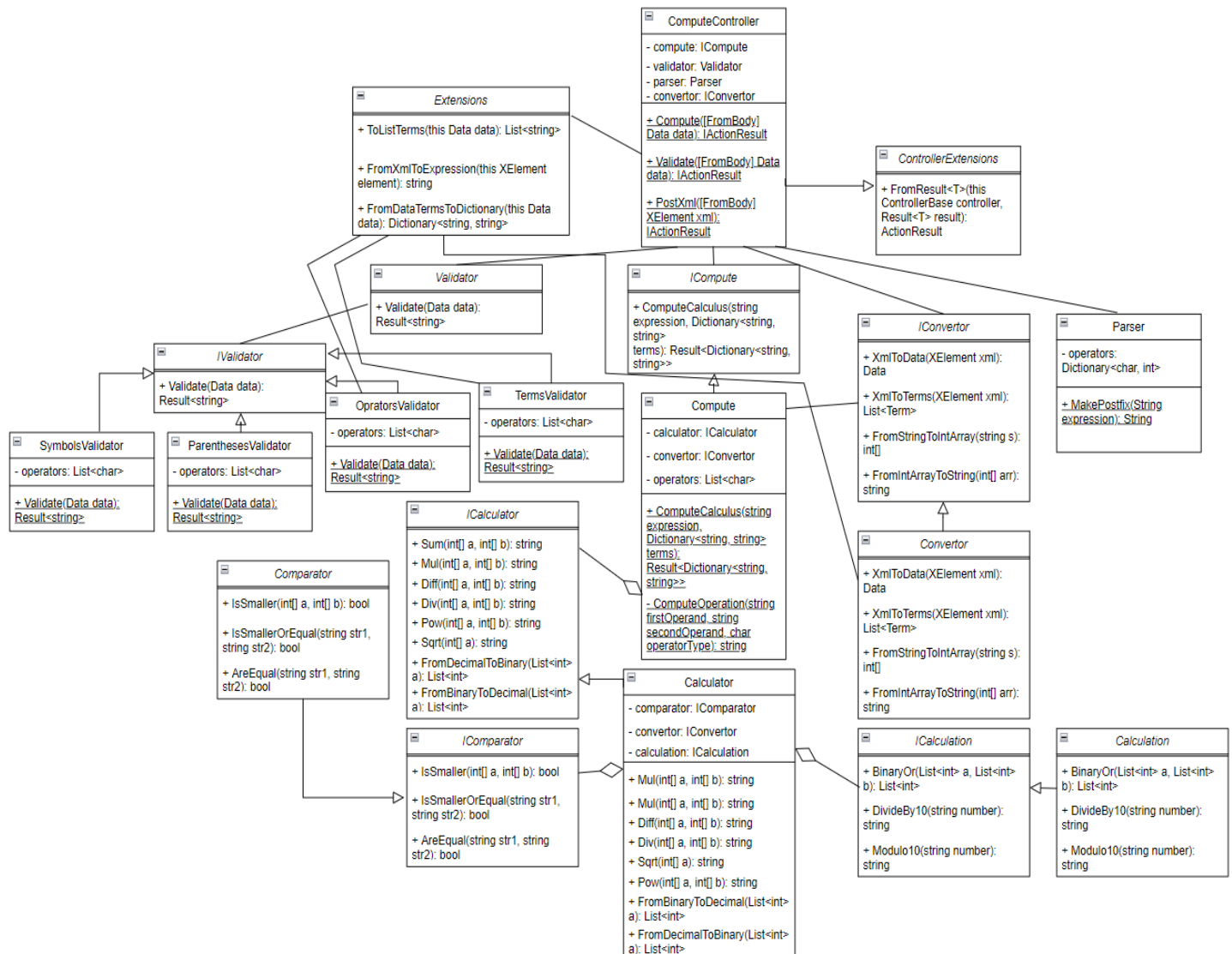
2.2. Running app/tests on Back-end

Running the project in the back-end individually is not necessary for a normal user, as all the functionalities implemented there can be more easily used in the front-end with the user interface. However, it can be useful for developers who want to test each endpoint individually. As the code was written in C# and .Net, any IDE that supports C# .Net 6 will work. Whenever we start the project using the BigCalculator.sln file, the program will open a local page using the Swagger API, where the developer can test functionalities separately, just as the Computer, Validate and ComputeXml modules. Pressing the execute button will then output a response code, as well the result of the computation, if applicable.

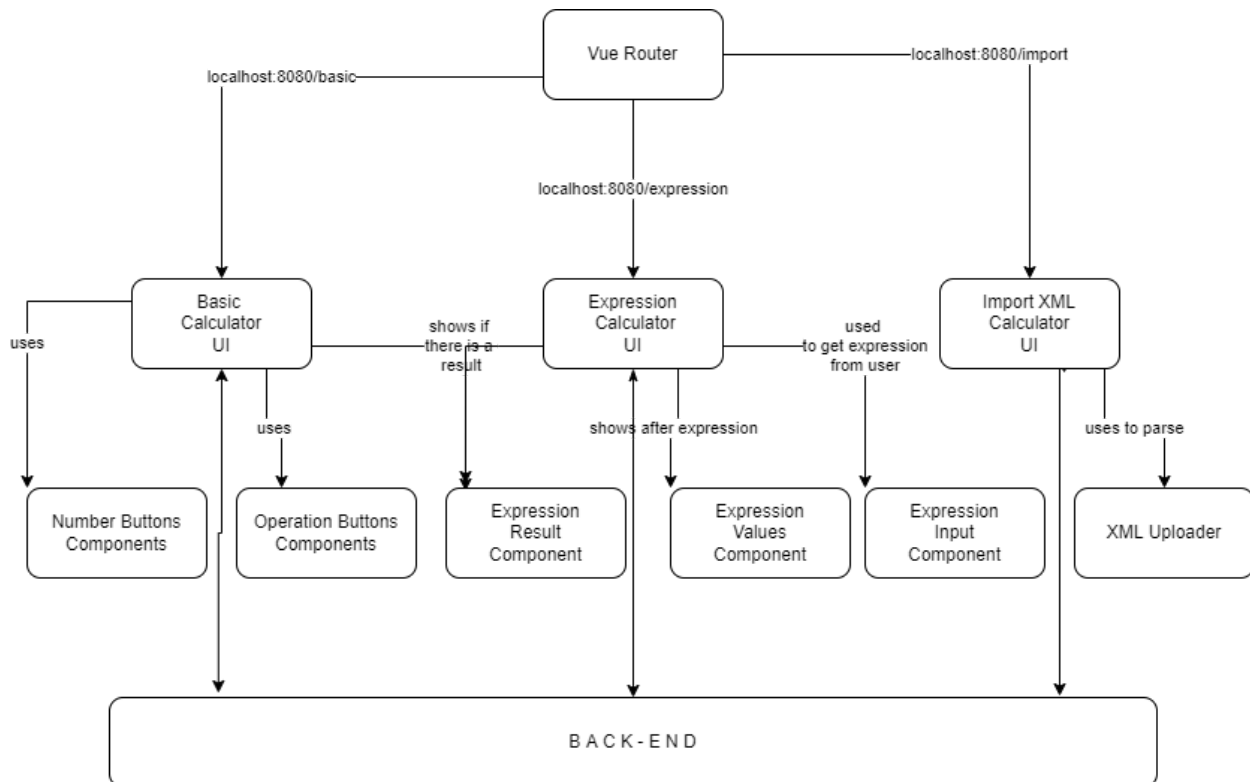
In order to run unit tests on the back-end, one option would be to open the project in VS 2022, right click on Solution 'BigCalculator' and click Run Tests.

3. ARCHITECTURE

3.1. Back-end Architecture



3.2. Front-end Architecture



4. MAIN MODULES

In this section from chapter 4.1 to 4.6 the modules on the backend will be described and then the modules on the frontend part will be presented starting with chapter 4.7.

4.1. Service Module

This module consists of the following main classes:

- **Parser.cs**: this class is responsible for transforming the arithmetic expression received as a string from the infix notation to the postfix one. The string it receives is already validated by the Validator module which will be explained later.

- **Converter.cs:** this class contains the implementation for the methods defined in the IConverter interface. These methods are responsible for mapping the content of the XML file to objects defined by us, in order to structure the data we will work with (more information about representing the data provided by the client is provided in the explanations for the *Core* module). There are also methods responsible for converting a number represented as a string to an int[] array (at one position in the array there is a single digit of the number) or vice versa. A structure of the XML file that contains the arithmetic expression together with the values of each term can be observed in Figure 1.

```
<data>
  <expression>
    <operand>a</operand>
    <operator>+</operator>
    <operand>b</operand>
    <operator>*</operator>
    <parenthesis>
      <operand>c</operand>
      <operator>+</operator>
      <operand>d</operand>
    </parenthesis>
    <operator>+</operator>
    <operand>e</operand>
  </expression>

  <terms>
    <term>
      <name>a</name>
      <value>100</value>
    </term>
    <term>
      <name>b</name>
      <value>101</value>
    </term>
    <term>
      <name>c</name>
      <value>102</value>
    </term>
    <term>
      <name>d</name>
      <value>103</value>
    </term>
    <term>
      <name>e</name>
      <value>104</value>
    </term>
  </terms>
</data>
```

Figure 1: XML file received from client

- **Comparator.cs :** this class contains the implementation for the methods defined in the IComparator interface. These helper methods are responsible for comparing two numbers represented as int arrays or as strings.

4.2. Adapter Module

This module exposes the interface *ICompute*, which defines the method (*ComputeCalculus*) for computing the given expression, and also provides the implementation of this interface in the *Compute* class. The method receives as parameters the posfixed representation for the arithmetic expression as *string* and a *dictionary*<*string*, *string*> containing the terms and their values. This class uses the methods (*Sum*, *Diff*, *Mul*, *Div*, *Pow*, *Sqrt*) defined in the *ICalculator* interface from the *Calculus* module in order to obtain the final result. What is returned is a *dictionary*<*string*, *string*> containing each step in the computation and the final result or in the case of an incorrect outcome (e.g. division by 0, negative result of a subtraction) it contains the the steps made until that moment and the type of error that occurred.

For example if the method *ComputeCalculus* receives the postfixed expression “ab+cad-^*” of “(a+b)*c^(a-d)” and the dictionary {“a” : ”18”, “b” : “14”, “c” : “2”, “d” : “8”} the following result represented as JSON will be returned to the client.

Response body

```
{
  "operation 1": "a + b = 32",
  "operation 2": "a - d = 10",
  "operation 3": "c ^ 10 = 1024",
  "operation 4": "32 * 1024 = 32768",
  "final result": "32768"
}
```

Figure 2: Results of each step of the computation and the final result

In Figure 3 it can be seen what is returned in case we change the value of “d” with “19” for example. First it is computed “a+b” and after that “a-d”, but the result of subtraction is negative and the execution of the method is stopped.

Response body

```
{
  "operation 1": "a + b = 32",
  "error": "Negative result of subsctraction: a - d"
}
```

Figure 3: Response returned to the client in case of negative result of subtraction

4.3. Core module

The client sends a JSON or XML file to the API via a post request, depending on whether the data (the arithmetic expression and the values of the variables) is entered

manually through the user dialog or not. The XML structure has already been shown in *Figure 2* and *Figure 4* shows how the JSON object is structured.

```
{
  "expression": "a+b*(c-d)+c",
  "terms": [
    {
      "name": "a",
      "value": "18324"
    },
    {
      "name": "b",
      "value": "144324"
    },
    {
      "name": "c",
      "value": "233"
    },
    {
      "name": "d",
      "value": "80000"
    }
  ]
}
```

Figure 4: JSON object send to the API containing the expression and the values of the variables

In order to map the JSON object or the XML content to .Net objects, classes *Data.cs* and *Term.cs* have been created. The *Data.cs* class contains a property of type string, representing the arithmetic expression and an *IEnumerable<Term>* for the terms of the expression. The *Term.cs* class has two properties of type string: one for the name of the variable and one for the associated value.

4.4. Calculus Module

In this module is present the classes that contain the methods that implement the operations for addition, subtraction, multiplication, division, power, square root. They are defined in the *ICalculator* interface and implemented in *Calculator* class. Each of these methods receives as parameters arrays of type int and the result returned is of type string. Because we chose to represent the numbers we work with as arrays of type int, each array position will contain one digit of that number. Also in the *ICalculation* interface we defined some methods like *BinaryOr*, *DivideBy10*, *Modulo10* which will be used to help implement basic operations.

4.5. Validator Module

An important step when receiving data from the client is to validate it and this is what this module does. The *IValidator* interface, that defines a method for validation, will be implemented by each of the following classes:

- **OperatorsValidator**: checks that the operators in the arithmetic expression are preceded or followed by the appropriate characters. For example, a symbol for the operator cannot be followed by ")" or the symbol for the square root must be followed by "(".
- **ParanthesesValidator**: verifies if the brackets are opened and closed properly
- **SymbolsValidator**: the only characters that are allowed in the expression are letters, parentheses, and symbols for operators. Anything other than that will make the expression invalid
- **TermsValidator**: As mentioned before, the client sends in addition to the arithmetic expression a list that contains each term and its value. If a term in the list is not found in the expression then the data received isn't considered valid.
- **Validator**: this class instantiates each of the classes mentioned above and calls *Validate* method for each of them. If for at least one of them the result is invalid then the data sent by the client is considered invalid.

4.6. API Module

This module contains ComputeController class which exposes three endpoints:

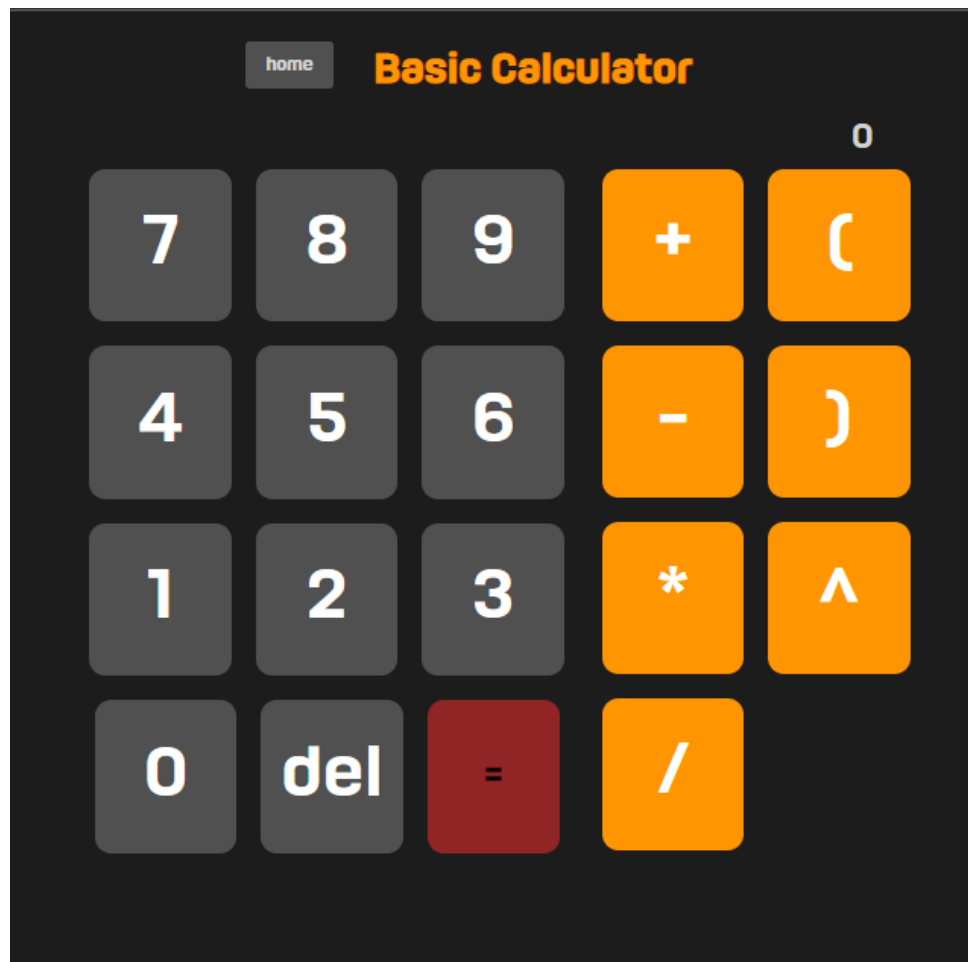
- one for computing the arithmetic expression when receiving in the request body a JSON as in Figure 4. As a response to the client it returns a JSON object as in *Figure 2* or *Figure 3*, depending on the outcome.
- one for validating the data received from the client.
- the last one has the same functionality as the first one, but an XML is sent in the request body instead of a JSON.

4.7. Basic Calculator UI Module

This module contains a basic vue component/module that acts like a basic calculator with the next operations : sum, subtraction, multiplication, divide, power as well as parentheses. This module is made out of the following components :

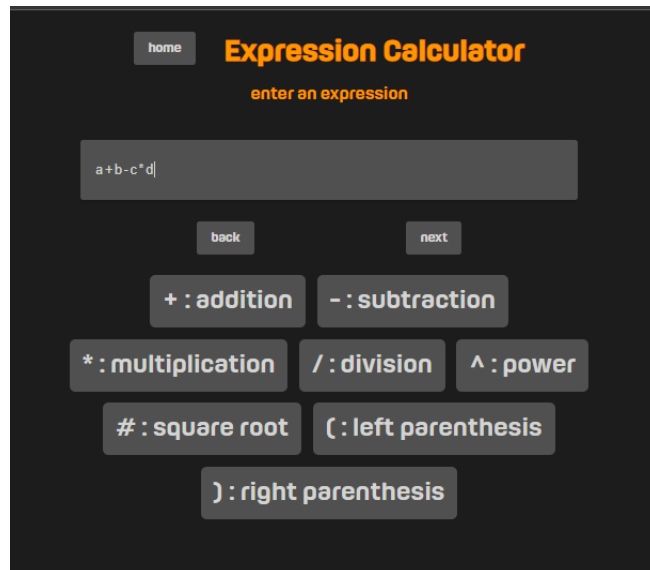
- **NumberButton.vue** : The component that is responsible for rendering the numbers on the screen and on the pressing of the button or when the user presses a number on the keyboard it adds it to the input field.
- **OperationButton.vue** : The component that checks if the user respects the correct order (e.g the user cant put divide after a sum sign), and on the press of the key or if the user presses it on the screen adds an operation.

After the user inputs the basic operation, the calculator transforms the operation into an expression that our back-end will understand and sends it. When it receives a response, it will then show it as well as offer the user the possibility to view an ‘advanced result’, where he could see the order of the operations as well as the outcome of each one, in the final leading to our final result.

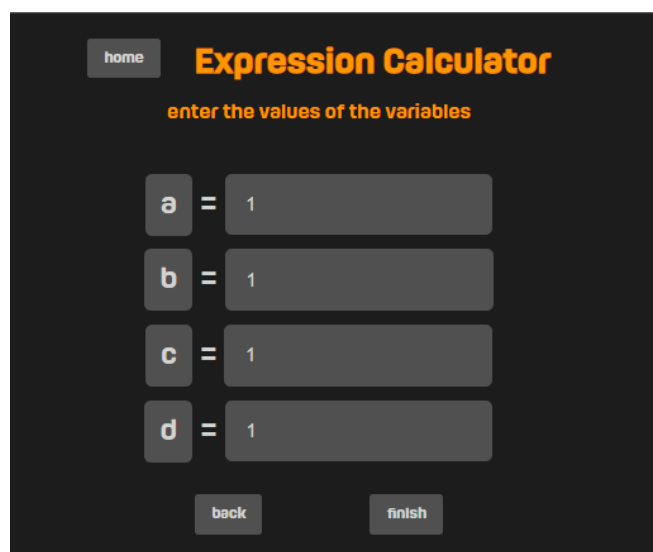


4.8. Expression Calculator UI Module

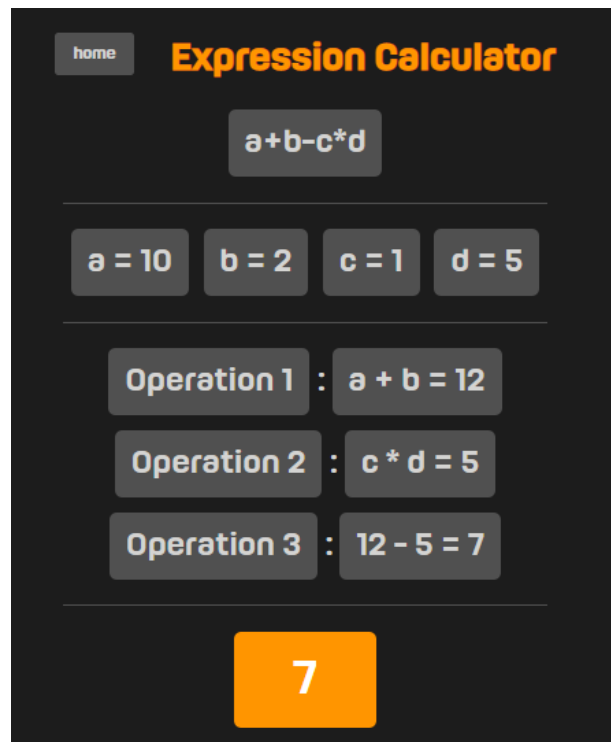
This module is made out of three parts that are shown in order of the flow of the application. First of all, the user has to introduce an expression that he wants to calculate.



After the user has introduced the expression, he can press the 'next' button, which will make a HTTP request to the Validator .NET Module. If the expression is bad, he will receive an error, otherwise he will be shown another component, ExpressionValue.vue, where he is supposed to put the values of the expression that he used in input earlier.



When the user has introduced all the values, he can press finish, which will again validate his expression if it contains negative numbers. If negative numbers are not present, and validation has returned a success response, the component will make a HTTP Request to compute in order to receive results and operations made in order to achieve the result.



5. UNIT TESTING

5.1. XUnit

xUnit.Net is an open-source testing framework based on the .NET framework. The creators of NUnit created xUnit as they wanted to build a better framework rather than adding incremental features to the NUnit framework. It is part of the .NET Foundation and operates under their code of conduct.

xUnit is created with more focus on the community; hence it is easy to expand upon. You can download xUnit from the NuGet gallery. So far, there are close to 7,500 questions tagged as xUnit on Stackoverflow.

5.1.1. Why XUnit?

The popular attributes [SetUp] and [TearDown] from NUnit are not a part of the xUnit framework. In our opinion, usage of [SetUp] and [TearDown] led to code duplication, so we used it because it provides the same features in a much more optimized manner than NUnit. For initialization, constructor of the test class is used, whereas, for de-initialization, IDisposable interface is used. This also encourages writing much cleaner tests. This makes this C# unit testing framework a much better option.

As far as NUnit vs. XUnit vs. MSTest is concerned, the biggest difference between xUnit and the other two test frameworks (NUnit and MSTest) is that xUnit is much more extensible when compared to NUnit and MSTest. The [Fact]

attribute is used instead of the [Test] attribute. Non-parameterized tests are implemented under the [Fact] attribute, whereas the [Theory] attribute is used if you plan to use parameterized tests.

In NUnit and MSTest, the class that contains the tests is under the [TestClass] attribute. This was not a very robust approach, hence [TestClass] attribute was also removed in xUnit. Instead, intelligence is built in the xUnit framework so that it can locate the test methods, irrespective of the location of the tests.

5.1.2. Given-When-Then naming

According to Martin Fowler, Given-When-Then is a style of representing tests. It's an approach developed by Daniel Terhorst-North and Chris Matts as part of Behavior-Driven Development (BDD). It appears as a structuring approach for many testing frameworks. You can also look at it as a reformulation of the Four-Phase Test pattern.

The essential idea is to break down writing a scenario (or test) into three sections:

- The **given** part describes the state of the world before you begin the behavior you're specifying in this scenario. You can think of it as the pre-conditions to the test.
- The **when** section is that behavior that you're specifying.
- Finally the **then** section describes the changes you expect due to the specified behavior.

5.1.3. Arrange Act Assert (AAA)

Arrange/Act/Assert (AAA) is a pattern for arranging and formatting code in Unit Test methods.

It is a best practice to author your tests in more natural and convenient way. The idea is to develop a unit test by following these 3 simple steps:

- Arrange – setup the testing objects and prepare the prerequisites for your test;
- Act – perform the actual work of the test;
- Assert – verify the result.

The benefits of using this pattern are:

- Clearly separates what is being tested from the setup and verification steps;
- Clarifies and focuses attention on a historically successful and generally necessary set of test steps;
- Makes some test smells more obvious:
 - Assertions intermixed with "Act" code;
 - Test methods that try to test too many different things at once.

In the Assert phase, when we have more than one assertion we make use of **AssertScope()** from the .NET library, Fluent Assertions. We batch multiple assertions into an AssertScope, so that FluentAssertions throws one exception at the end of the scope with all failures.

```
//Assert
using (new AssertScope())
{
    Assert.Equal(expected.Data, computationResult.Data);
    Assert.Equal(expected.ResultType, computationResult.ResultType);
    Assert.Equal(expected.Errors, computationResult.Errors);
}
```

5.1.4. Mocking

In order to instantiate some services that we are not interested in testing, we make use of mocking. The .NET library that we use is **Moq** and call the constructor of the **Mock** class to instantiate a service.

```
public ComputeUnitTests()
{
    var convertor = new Mock<Convertor>();
    var comparator = new Mock<Comparator>();
    var calculation = new Mock<Calculation>();
    var calculator = new Mock<Calculator>(comparator.Object, convertor.Object, calculation.Object);

    compute = new Compute(convertor.Object, calculator.Object);
}
```

5.1.5. Test sample

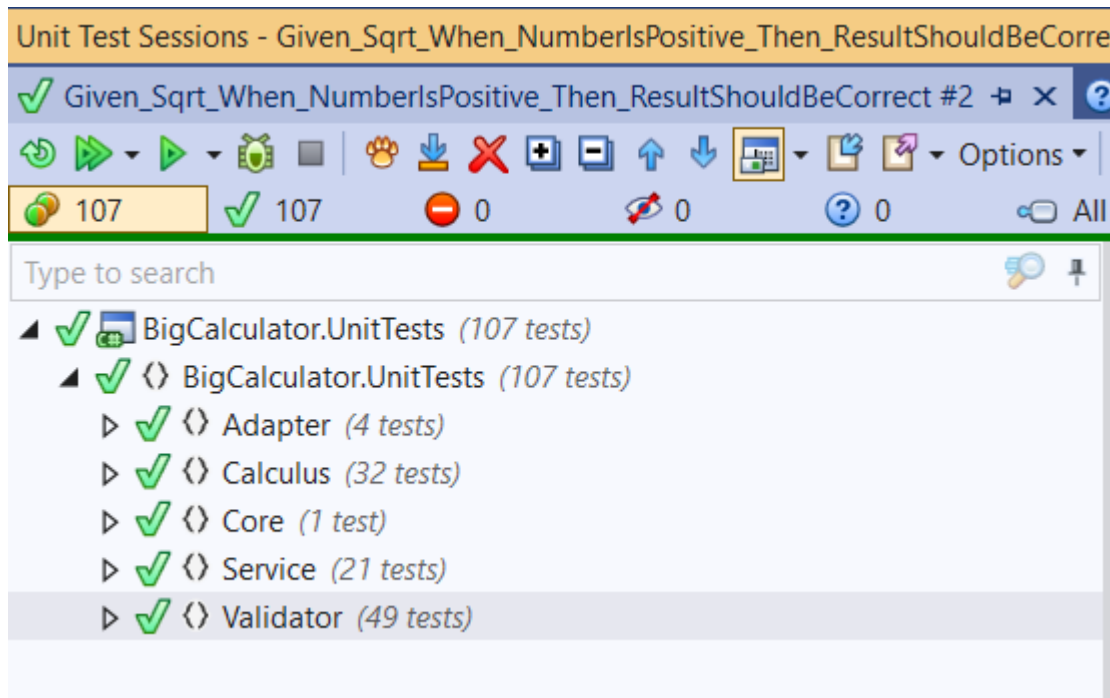
```
[Fact]
0 references
public void Given_ComputeCalculus_When_ValidPostfixedExpressionAndTerms_Then_ExpectedResultIsReturned()
{
    //Arrange
    string postfixExpression = "abcd-a^*+*#";
    Dictionary<string, string> terms = new Dictionary<string, string>()
    {
        { "a", "2" },
        { "b", "3" },
        { "c", "7" },
        { "d", "5" },
    };

    //Act
    var computationResult = compute.ComputeCalculus(postfixExpression, terms);

    //Assert
    Assert.Equal("3", computationResult.Data["final result"]);
}
```

5.1.6. Unit test results

We have written a total of 107 unit tests for the BigCalculator project. All of the passes successfully.



5.1.7. Back-end code coverage

For back-end code coverage we used library Fine Code Coverage which provides code coverage using one of 3 different coverage tools. We managed to write unit tests that cover the whole coverable classes in the BigCalculator project, achieving 100% line code coverage.

Name	Covered	Uncovered	Coverable	Total	Line coverage
+ BigCalculator.Adapter	71	0	71	118	100%
+ BigCalculator.Calculus	400	0	400	567	100%
+ BigCalculator.Core	73	0	73	318	100%
+ BigCalculator.Service	123	0	123	191	100%
+ BigCalculator.Validator	80	0	80	145	100%

Assemblies:	5
Classes:	16
Files:	14
Covered lines:	747
Uncovered lines:	0
Coverable lines:	747
Total lines:	1197
Line coverage:	100% (747 of 747)

5.2. Jest UI Testing

Jest is a JavaScript testing framework maintained by Facebook, designed and built with focus on simplicity and support for large web applications. We've decided to use it for our UI Modules because it has good tools for Vue testing (the framework we are using) as well as it doesn't require a lot of configuration for first-time users of a testing framework.

5.2.1. Why Jest ?

Jest is a test runner, an assertion library and a mocking library which provides built-in test coverage tools and features such as snapshot testing. In contrast with other testing frameworks for JavaScript, Jest does not require a real browser environment to run, and is running on its own Node.js process where the API's are emulated with jsdom. Such an environment speeds up the execution of tests and allows to run tests of different systems with the same results.

Also, the Vue CLI has a built-in Jest environment which helps us keep everything in the same place as well as see the coverage we have after each test we wrote.

5.2.2. Test runner

As a test runner, Jest finds and executes tests. Our job is to give a path to a folder where our tests are located, and Jest executes the tests inside those files. By default, without configuration, Jest will check every file inside the root directory of the project. We've configured the app to take into the account only the .spec files.

Jest CLI package is a command line runner used to execute tests with Jest. The `jest` command is used to execute tests with the options specified in the Jest configuration file, or with the default configuration options if no configuration file is provided. The `jest` command accepts multiple options that specify how the tests should be executed. These options can be combined for maximum precision. Each command line option can also be specified in the Jest configuration. Command line options can be useful to override or extend the configuration options for a single test run. Jest CLI options include `--watch` and `--watchAll`, used to automatically re-run tests when a file is modified. `--watch` reruns only the tests related to modified files, while `--watchAll` re-executes all tests every time any file is changed. Running Jest with these options is referred to as running Jest in watch mode.

Jest runs each test in a separate process, providing isolation and speeding up the execution. To further speed up developer feedback, Jest executes previously failed tests first and re-organizes the order in which tests are run based on how long each test takes

5.2.3. Assertion Library

As an assertion library, Jest provides a set of tools to write tests in a simple, human-readable way. These tools are functions called matchers. Matchers allow testing values in different ways. In order to test a specific value, an expectation object needs to be created. Then, a matcher is called on the expectation object. The **describe()** function defines a test suite, which is a group of tests. The first argument is the name of the describe block, while the second is a function that contains one or multiple tests. The **it()** function defines a test. The first argument is the name of the test, the second argument is a function containing the code of the test. Finally, the **expect()** function returns an expectation object. It takes a value as an argument (commonly a value generated by a piece of code under test) and returns an object that exposes a set of matchers. The **toBe()** matcher verifies strict equality between the argument of **expect()** and the argument of its own. We've also used the **mount** and **shallowMount** from **@vue/test-utils** which helped us 'render' the components in order to test them.

```
import { shallowMount, mount } from '@vue/test-utils'
import Calculator from '@components/Calculator.vue'

describe('Calculator', () => {
  it('has data', () => {
    expect(typeof Calculator.data).toBe('function')
  })

  it('on press number button on page adds an element', () => {
    const wrapper = mount(Calculator)
    wrapper.find('.bc_body_number-buttons_button').trigger('click')
    expect(wrapper.vm.charRow.length).toBe(1)
  })

  it('on add operation without number should not work', () => {
    const wrapper = mount(Calculator)
    wrapper.find('.bc_body_operation-buttons .bc_body_number-buttons_button').trigger('click')
    expect(wrapper.vm.charRow.length).toBe(0)
  })

  it('on press key button on keyboard adds number then an operation', () => {
    const wrapper = mount(Calculator)
    wrapper.find('.bc_body_number-buttons_button').trigger('click')
    wrapper.find('.bc_body_operation-buttons .bc_body_number-buttons_button').trigger('click')
    expect(wrapper.vm.charRow.length).toBe(4)
  })
})
```

5.2.4. Mocking library

As a mocking library, Jest provides a tool called mock functions to create test doubles. Mock functions allow replacing dependencies inside of a unit test with something that can be controlled and inspected. This includes erasing and/or replacing

the actual implementation of a function or a module inside of a test, setting a return value for a function, capturing function calls and parameters passed in those calls. With Jest, mock functions can be created in two ways: either a mock function is created within a test, or a manual mock is set up to substitute a module dependency. A mock function can be created by calling `jest.fn()`

Each mock function has a `mock` property. That property stores the information about how the function was called and what it returned. It also stores the value of the `“this”` keyword, which is the context in which the function was called. Calls to `toHaveBeenCalled()` access the `mock` property of the mock function under the hood, but that property can also be accessed directly. The alternative to creating mock functions directly within test code is manual mocks. Manual mocks are used to replace the functionality of collaborator modules with test doubles. Both custom user modules and Node modules can be manually mocked. In order to mock a user module, a `__mocks__` subdirectory must be created in the directory containing the module. A file with the same name as the mocked module must be placed into the `__mocks__` subdirectory. To replace the module with the manual mock within the test, the line `jest.mock('./name_of_module')` must be added to the test. In order to mock a Node module, the manual mock must be placed in the `__mocks__` directory located in one of the root directories specified by the `roots` property of the Jest configuration. If the `roots` option is not specified, the `__mocks__` folder must be placed in the same directory as the `node_modules` folder. Explicit call to `jest.mock('./name_of_module')` is not required for mocking Node modules. The Node module will be replaced with the mock automatically.

5.2.5. Built-in code coverage

Jest includes the Istanbul library for JavaScript code coverage. Istanbul provides a number of coverage reporters that allow them to generate reports in different formats. Reporters can be specified using the `coverageReporters` option of the Jest configuration. The default value of this option is `["json", "lcov", "text", "clover"]`.

- Clover option generates a report in XML format
- JSON option generates a report in JSON format
- LCOV option generates an LCOV coverage file with an associated HTML report.

```
at Object.<anonymous> (tests/unit/example.spec.js:2:1)
```

```
PASS tests/unit/expression.spec.js
PASS tests/unit/expressionvalues.spec.js
PASS tests/unit/expressionresult.spec.js
```

```
PASS tests/unit/calculator.spec.js
```

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	100	66.66	100	100	
Calculator.vue	100	100	100	100	
Expression.vue	100	0	100	100	1
ExpressionResult.vue	100	100	100	100	
ExpressionValues.vue	100	0	100	100	1
NumberButton.vue	100	100	100	100	
OperationButton.vue	100	100	100	100	

```
Test Suites: 1 failed, 4 passed, 5 total
Tests:       15 passed, 15 total
Snapshots:   0 total
Time:        4.884 s
Ran all test suites.
```

6. ASSERTIONS

6.1. Debug.Assert

Assertions in our code have been done using `Debug.Assert` from the library `System.Diagnostics`. This operation has been chosen as it is the most commonly used for assertions for its simplicity and effectiveness. The function is only tested during `Debug` mode and verifies a certain condition or state of a variable. If the condition holds, the program does not perform any action, but if it fails it will output an error on the stack alongside with an error message, which can be written as the 2nd function argument.

Assertions were used in our project in order to check the preconditions, postconditions and invariants of the operations implemented. Assertions were written in the `Calculator.cs`, `Convertor.cs`, `Calculation.cs`, `Extensions.cs` classes.

6.2. Preconditions, Postconditions and Invariants

Preconditions were added to test whether the input variables contain the values they are supposed to have and any other restrictions that are placed on them in order to verify the validity of the operation. On most of the computations, those include checks if the operands contain negative or multiple values on a single digit, if the divisor is 0 or whether a binary number contains values other than 0 or 1.

```

var distinctA = a.Distinct();
Debug.Assert(distinctA.Count() is <= 2 and >= 0, "Binary number a contains values that are not 0 or 1");
var distinctB = b.Distinct();
Debug.Assert(distinctB.Count() is <= 2 and >= 0, "Binary number b contains values that are not 0 or 1");

```

Postconditions are similar to preconditions, but instead of testing the validity of the input, they are performed at the end and check if the values outputted are correct. Most operations verify if the length of the result is correct by comparing it with the lengths of the input variable or check if all the characters in the number are digits without an additional symbol, such as “-”. For the binary operation, we also verify that the number of 1s in the result is the correct length and does not have a higher number of 1s than each individual input.

```

Debug.Assert(invA.Count(x => x == 1) >= a.Count(x => x == 1),
    "The number of 1s in the final array is lower than the first element");
Debug.Assert(invA.Count(x => x == 1) >= b.Count(x => x == 1),
    "The number of 1s in the final array is lower than the second element");
Debug.Assert(invA.Count == Math.Max(a.Count, b.Count),
    "The binary or result array length is not equal with the greatest array as parameter");

```

Invariants are added to the code to verify if loops function properly and that intermediary variables have the correct values. For example, in the Sum algorithm we check that the value of any position in the result is calculated as it would be in a normal addition.

```

Debug.Assert(maxLength - i > 0); //variant
Debug.Assert(result[i] >= 0, "Digit of result is negative"); //invariant

```

7. CONTRIBUTIONS

Drăgoi Ștefan - project architecture, implementing Validators, Calculator operations (Div), Convertors, Calculations and Comparators, writing unit tests and assertions.

Cunev Serghei - front-end project architecture, implemented front-end ui components using Vue.js framework, Basic Calculator UI Module, Expression Calculator UI Module, Import XML UI Module, all UI components, set up Jest environment for the project, wrote all (15) unit tests for the front-end using Jest framework.

Condurache Andrei - implemented Parser, Compute, Data, Term classes. Added functions in Convertor and Calculation classes and operation in Calculator (Pow), wrote unit tests and assertions and described the “Main Modules” chapter in the documentation (from 4.1 to 4.6).

Tiperciuc Ștefan - implemented Calculator operations fully(Sum, Diff, Mul, Sqrt), helped with Pow and Div, wrote the majority of the assertions in modules Calculator.cs, Convertor.cs, Calculation.cs, Extensions.cs