# Calculate Demo – design and architecture

Code: <https://github.com/tgralex/CalculateDemo>

Live demo: <https://www.test.tgralex.net/>

## Project Requirements:

Be two daemons:

* One web server
* One backend service

For the web server:

* It must serve up one page that allows the user to request an answer to a maths problem with virtually unlimited operators (at least 100)
* The operators supported are: +, -, /, \*
* It will submit the requested problem to the backend service and produce an answer when available
* Once solved and available from the backend, the web server will display the result

For the backend service:

* It must listen on a different port from the web server.
* It must be secure
* It must log all transactions

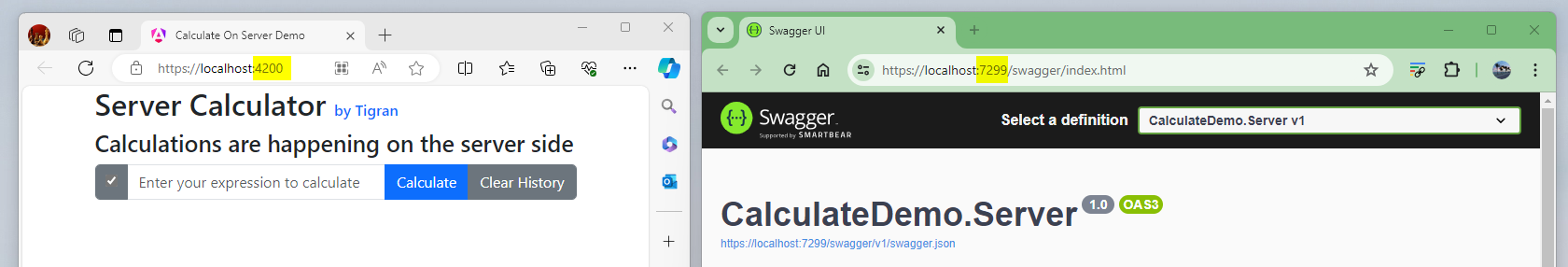
Both frontend and backend services should have some high-quality unit testing.

## Project Implementation

The solution consists of three separate projects:

1. Front-end (calculatedemo.client) –
   1. Angular 17.0
   2. Typescript
   3. Bootstrap 5.3
   4. JQuery 3.7
   5. Karma-Jasmin for tests
2. Back-end (CalculateDemo.Server) –
   1. Asp Net Core
   2. Dot Net 8.0
   3. NLog for logging
3. Back-end Unit tests (CalculateDemo.ServerTests)
   1. MSTestFramework

## Meeting the requirements:

1. When the solution is run locally, the client operates on port 4200, and the back-end service runs on port 7299  
   
2. Detailed logs are recorded for every significant application event, including
   1. Start and Stopevents
   2. Every API request and response
   3. Any internal errors
3. Back-end security
   1. Enforces the use of HTTPS protocol
   2. Disallows CORS, with an exception for requests from localhost:4200 during development.
   3. Limits the size of any HTTP request body to prevent buffer overflows. The size limit is configurable in the project's **appsettings.json**, with a default set to an arbitrary 100,000 bytes.
   4. Implements IP Rate Limitation to prevent DDoS attacks. The rate is configurable in the project's **appsettings.json**, with a default setting of one call per second per IP.
   5. Uses PUT method instead of GET for increased security.
   6. Sanitizes user input by preprocessing the expressions passed through the API. This involves stripping all non-numeric characters except decimal points, parentheses, and the four arithmetic operators.
   7. Processes expression calculations within a try-catch construct, gracefully handling incorrect expressions and calculation overflows."

# Development details.

## The Web-Client

The client, developed in Angular, features a single page with one api-service component injected into it.   
This abstraction decouples the client from direct HTTP requests, facilitating seamless transitions between receiving results from the server or simulating them locally ('faking it').   
This setup enables independent testing of the client separate from the server.   
Detailed instructions on how to use and test the client are provided in a separate document titled **“Web-Client Demo User’s manual”**.

## Back-end

The back-end service is developed using ASP.NET Core, and all server configurations are accessible in the **program.cs** file.   
For better readability, each security feature is encapsulated within its own method.   
The Calculate controller includes one asynchronous PUT method that instantiates a Calculate class, passes the input parameter to it, and returns an object (exposing only its public properties) back to the client.

The Calculate class operates as follows:

1. **Preprocessing the Expression:**   
   It sanitizes the input by stripping all non-numeric values except for decimal points, parentheses, and the four arithmetic operators. If the expression changes during this process, it halts further processing.
2. **Calculation Using Polish Notation:**   
   This system, also known as prefix notation, places the operator before its operands. The calculation involves:
   1. Preprocessing the expression.
   2. Calculating the final amount during the second pass.

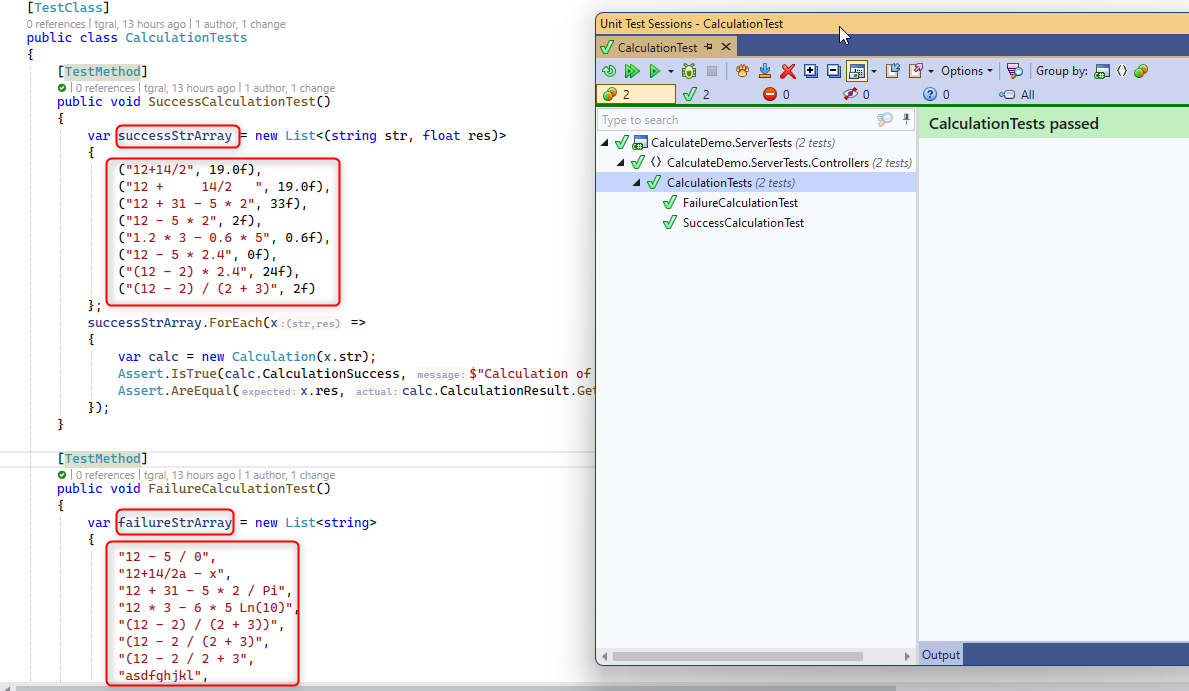
## Back-end Unit Test

Tests were conducted on two distinct sets of expressions

1. **Successful Expressions:**   
   These include expressions with known results.   
   The tests verify that the calculated results match the precalculated values.
2. **Incorrect or Overflow Expressions:**   
   This set tests expressions that are incorrect or cause overflow, as well as other edge cases.   
   The expectation for these tests is to fail appropriately.

All tests were conducted successfully, confirming the reliability and robustness of the application under various scenarios.

Below is a screenshot showing the successful execution of the tests



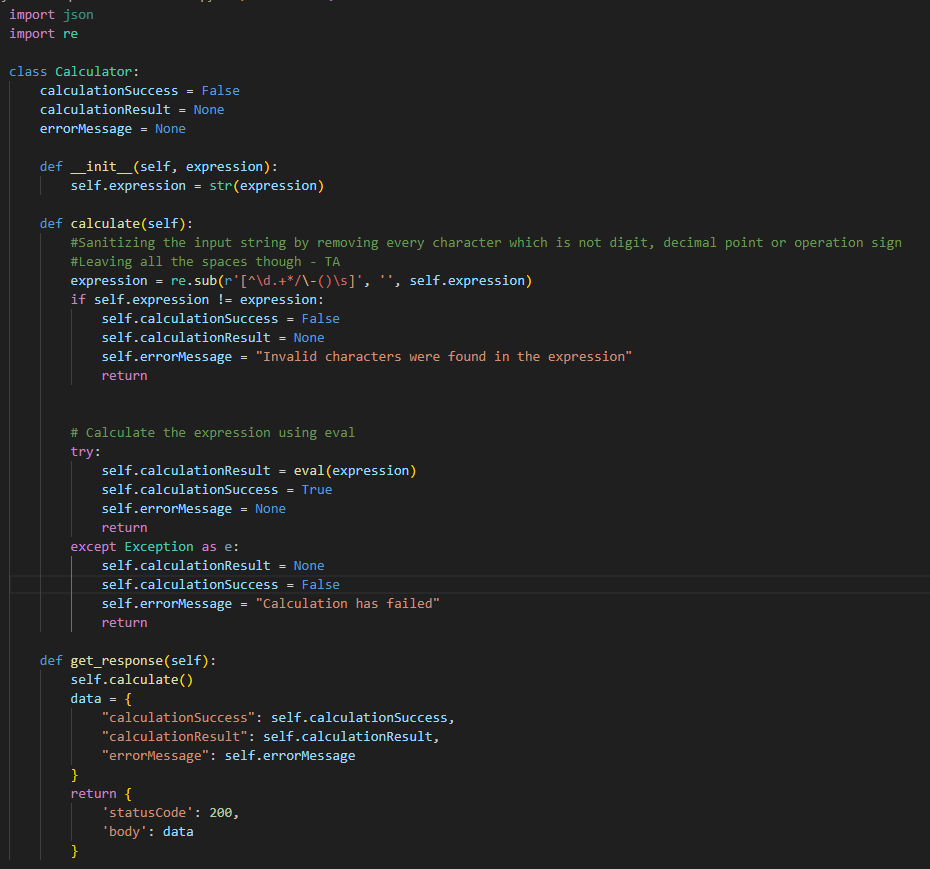
## Further research

I researched what could be done to increase security if we wanted to expand and enhance. I looked to Google Maps public API and researched AWS components

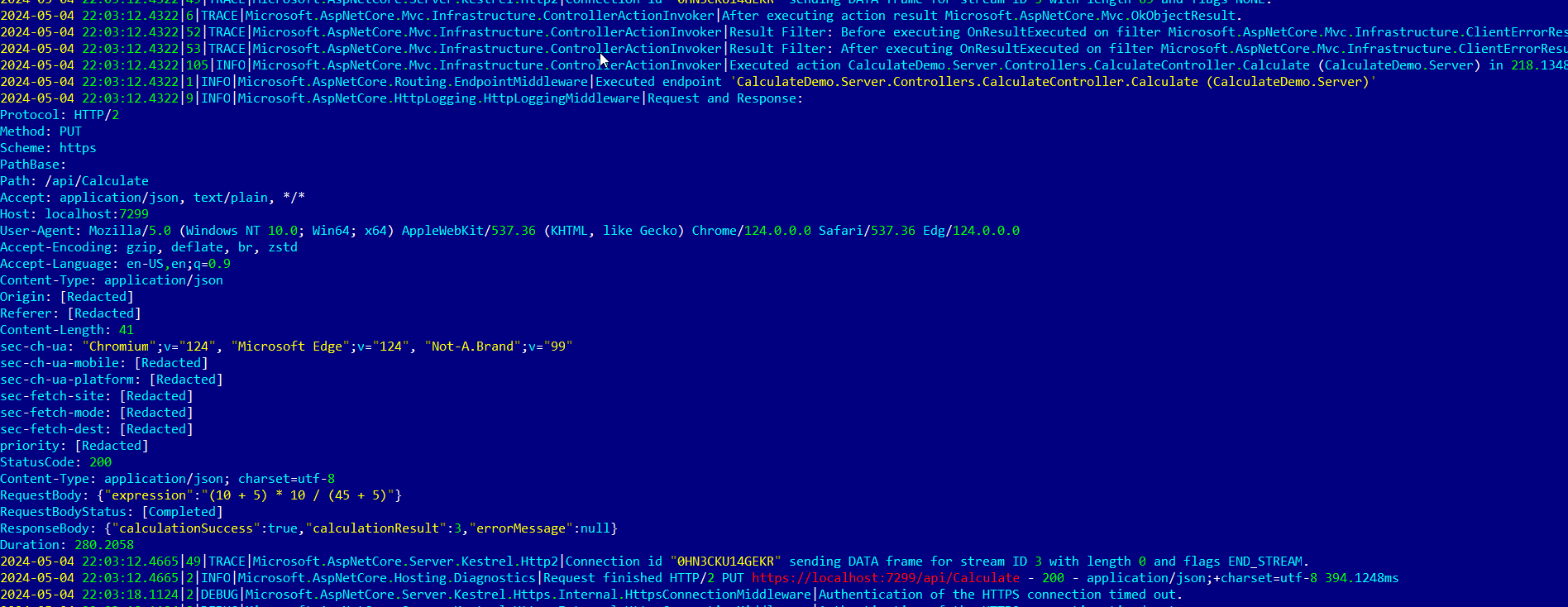
So here is an architecture I found might work for a public open API service:

1. **AWS API Gateway with CloudFront:**
   1. Entry Point Configuration: Use AWS API Gateway as the public-facing endpoint. It can be integrated with a Lambda function to manage incoming requests
   2. CloudFront Integration: Enable AWS CloudFront, a content delivery network (CDN), to work with the API Gateway.   
      CloudFront offers response caching and additional security features such as DDoS protection.
   3. Configure CloudFront to capture the client's IP address
2. **Passing Client's IP Address to Lambda Function:**
   1. CloudFront adds the client’s IP address to the X-Forwarded-For header, which is forwarded to the API Gateway.
   2. Set the API Gateway to forward this header to the Lambda function within the event payload.
3. **Validate the client's IP address in the Lambda function:**
   1. Retrieve the client's IP address from the X-Forwarded-For header within the Lambda function.
   2. Since this header is added by the trusted AWS service CloudFront, you can rely on its accuracy.   
      Validate the IP address against an allowlist or perform other necessary checks.
4. **Implement additional security measures:** 
   1. Use strong authentication mechanisms such as API keys, OAuth, or JSON Web Tokens (JWT) for identifying and authorizing API callers, in addition to IP address validation.
   2. Implement rate limiting based on trusted IP addresses or API keys, and monitor for unusual behavior or excessive requests from certain sources.
5. **Enable HTTPS/TLS Everywhere:**   
   Ensure that all communications are secured with HTTPS/TLS to protect data in transit.
6. **Whitelisting or Blacklisting IPs by Region or Country:**Limit access to the service by allowing or blocking IP addresses based on their geographical regions or countries.
7. **Dynamic IP Ban System:**
   1. **Temporary Bans**: Implement a sliding scale ban for IPs that send excessive requests. Initially, impose temporary bans with increasing duration overtime.
   2. **Permanent Ban:** Escalate to a permanent ban if the behavior persists.
   3. **Unbanning Mechanism:** Offer the ability to lift the ban upon successful completion of human verification challenges, such as CAPTCHAs or solving simple puzzles

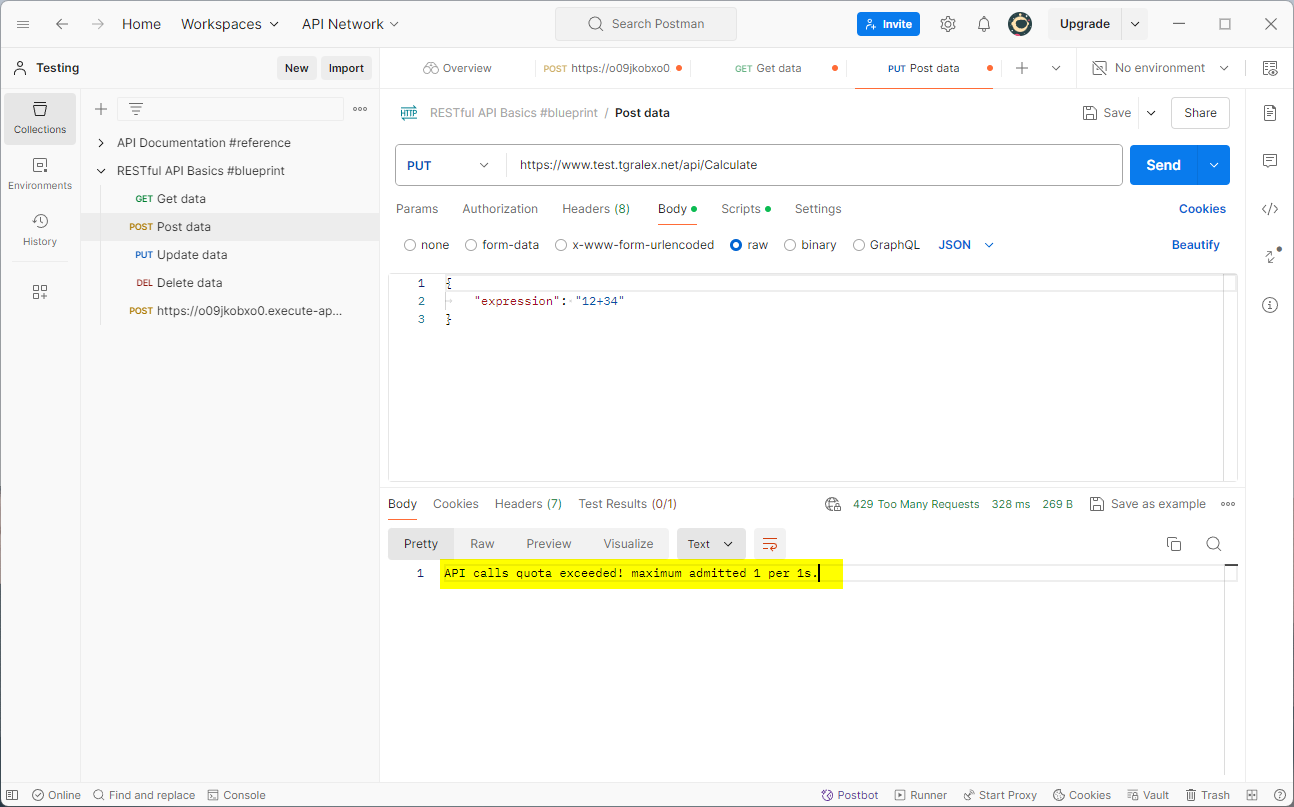
I also extended the implementation to include Python code for the Lambda function, which benefits from the simplicity and power of the built-in Eval function. This allows for the evaluation of a broader range of operations in just a single screen of code. For comparison, in C#, to parse and evaluate expressions, I implemented the Polish notation (prefix notation), which requires placing the operator before its operands. This involved preprocessing the expression and calculating the final amount in two passes. The Python code, along with its corresponding unit tests, has been included in the project submission.



## More screenshots

Example of log:  


Appsettings.json:   


Exceeding IP rate per period quota (using Postman):  


Exceeding API calls body limit   
(was set to 100 bytes to see):  
