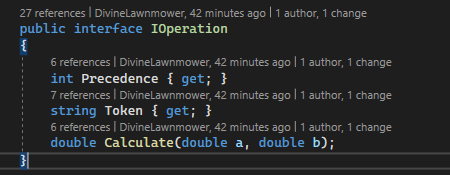
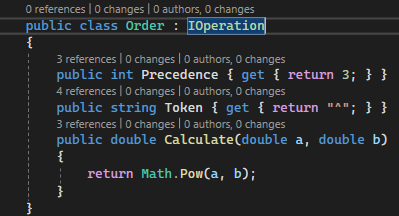
Calculator

I was tasked with creating an ASP.NET API implementation of a calculator. The implementation required that the expression be evaluated using BODMAS rules (Brackets Order Division Multiplication Addition Subtraction in order of precedence). Other rules which were to be considered included the lack of requirement to handle negative numbers as well as the lack of requirement to handle bracelets within the implementation.

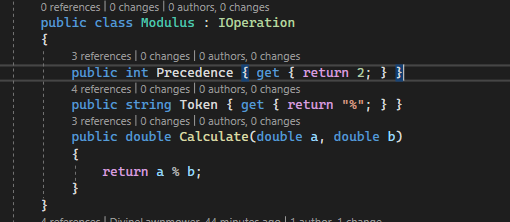
SOLID

In order to ensure I followed SOLID principles I implemented my core functionality such that it could be extended with minimal to no modification of the base class. For example my calculator relied upon several base interfaces such as:

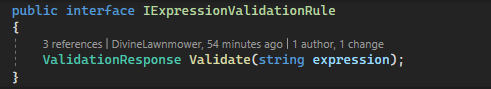
* IOperation - The base interface for any operations to be implemented in future e.g. order ^ or modulus %  
  

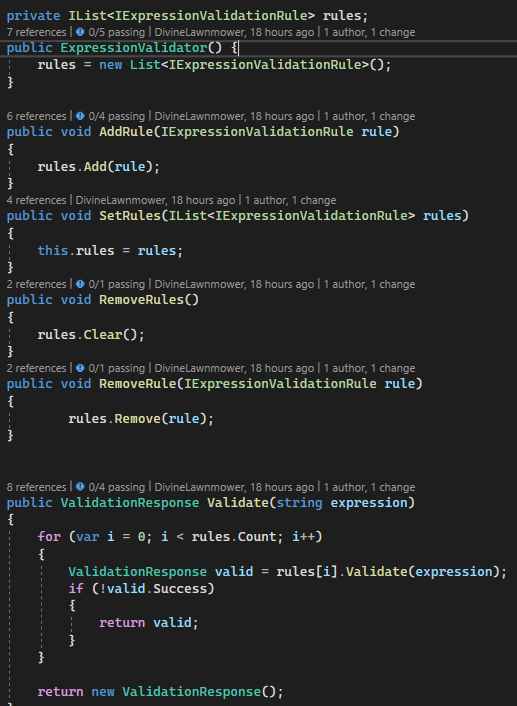
An example implementation of the above would be:   


OR



* IValidationRule - The base interface for the implementation for any future validation rules to be applied when attempting to validate the expression passed by a User.



These rules can be passed into the Expression validator using the following methods:  


This ensures that the validation for the calculator is extendable with minimal code changes required outside of creating a new validation rule.

Initial research taught me about the Shunting Yard algorithm which is commonly used to parse mathematical expressions by:  
1) Passing each number or operand to a stack e.g.:

The expression "4+5/2-1" being passed results in the following stack:

4,5,2,1,/,+,-.

2) Popping the operand from the stack, whenever the respective operand is read, performing the respective operation then pushing the result back to the stack to be used in future operations.

4

4, 5

4,5,2

4,5,2,/

4,5,2,/,+

4,5,2,/,+,1,

4,5,2,/,+,1,-

4,

4,5,

4,5,2

New stack:

5,

5,2

5,2,/

2.5 (5/2 == 2.5)

2.5,4,

2.5,4,+

6.5 (2.5+4 == 6.5)

6.5,1

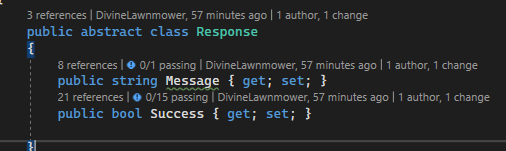
6.5,1,-

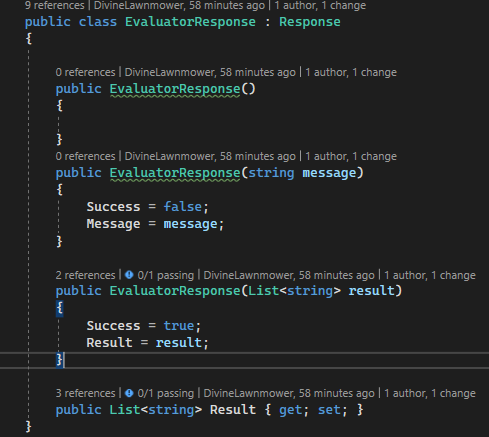
5.5 (6.5-1 == 5.5)

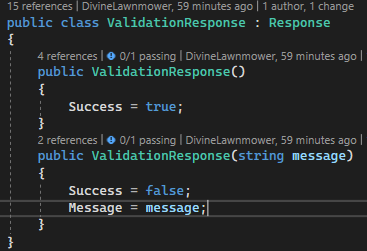
3) When there are no further tokens (e.g. operators, numbers) to read. The resultant value on the stack is the answer to the passed expression.

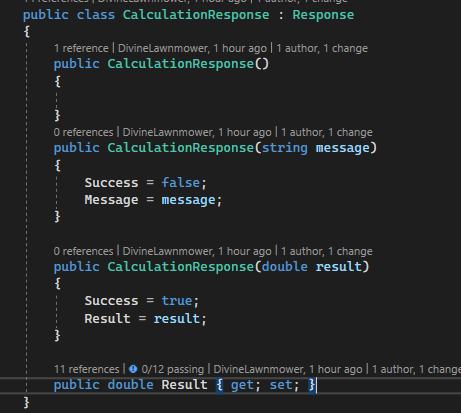
Result: 5.5

There were numerous other possible implementations to follow when determining a solution for the posed problem e.g. Datatables or ANTLR however the subsequent implementations would not have shown off any skills as they either immediately determine the answer or require a third party library respectively.

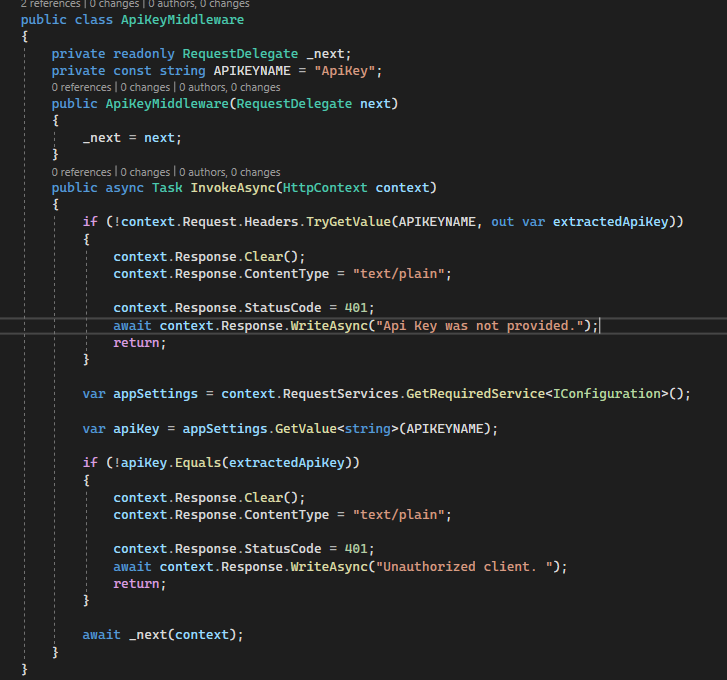
I chose to implement a common response object IResponse so that each step in the process could easily return a boolean success flag as well as a string method error which could be returned to the user as a user friendly error. Furthermore, the IResponse was extended by the relevant classes to include a Result property to be passed onto the next step in the process of evaluating/calculating the result of the expression provided:  








The Calculator Service takes a Dependency Injected Evaluator and Validator so that the implementation of either differs dependending on requirements.

Securing the API:  


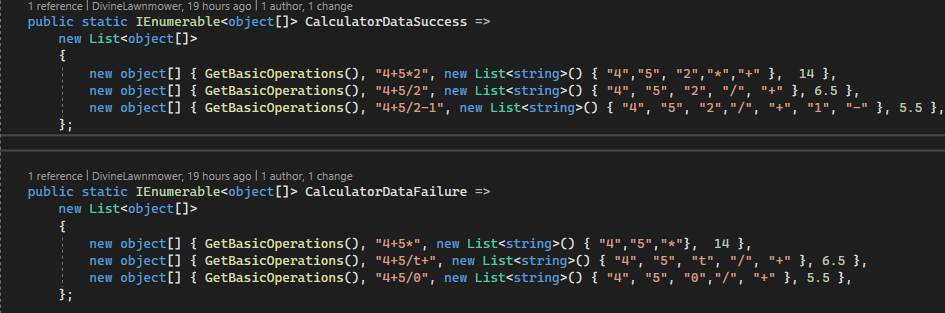
I opted for a quick and easy api key authorisation on the calculator api with the key currently stored in the appsettings.json. The lack of an api key in the header of a request will result in a “Api Key was not provided” 401 response while an non-matching api being provided will return a 401 “Unauthorised client” response.

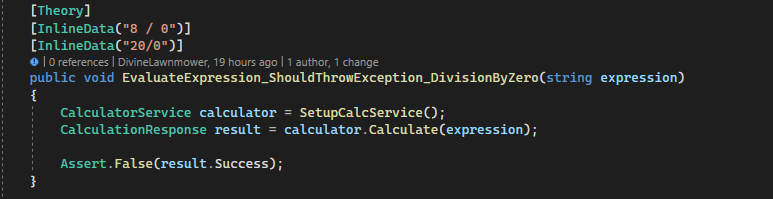
The middleware works across all controllers within the affected project. Should a more specific implementation be required you could create Attributes to markup the controller methods with. This way you can specify set methods to authorise against.

Furthermore, I decided that the api should be rate limited so as to provide an additional protection against DOS attacks. As I used .NET 6 (Current LTS) there was no inbuilt implementation as there is in .NET 7 and 8. Instead I implemented a rate limiting middleware which limited the user based on their IP address which gets stored in a distributed memory cache and checked against each upon request.

Unit Tests:

As I had split out the implementation of the validator, shunting yard evaluator and the calculator it was possible to test all of these parts individually.

I mostly used MemberData to provide test cases as this enabled me to pass all the required data to each test as well as provide multiple cases per test easily, however I did use InLineData to pass the more specific cases such as ensuring that divide by zero errors were caught. E.g.  




Testing the shunting yard evaluator I only needed to provide the input expression string and ensure that the returned string list matched.

Testing the validator consisted of ensuring that the rules could be added, removed etc as well as a passing test case and a failing test case for a “No letters allowed” validation rule. As the rules list is private a helper method for reflecting the list of rules was required to access them.