**Approach**

NumToWordsConverter web application is built on a ASP.NET MVC template. The user input is read into the Form model and the output is stored in the Fom model. In this case, the Form model acts as a **DTO** to transport data from the application layer to the service layer and vice versa. Ideally, the DTO is separate from the model (or entities), however, for this specific application use case, no database is needed. Therefore, the model acts in place of a DTO. The user input is also not directly retrieved via the url parameters to reduce the likelihood of **SQL injection attacks**. The business logic is **decoupled from the controller by introducing a service layer** NumToWordsConverter.cs. This class has 2 private methods and is introduced to HomeController via **dependency injection** to ease mockability and testability as well.  
  
   
A naïve implementation of will be a brute-force approach.Checking if the value is <10 and retrieve the correct word from the mapping and another checking for 10-100 and so on.This approach is not scalable to larger numbers as we have to extend the logic for numbers in thousands and extend the mapping.   
  
The approach I have opted for is an improved version of the brute force approach with has a time complexity of O(log N) and O(1) space complexity.According to the International Number System, I parse the number 3 digits at a time from the least significant digit. Therefore, I created four readonly static mappings for the NumToWordsConverter service.Ones for word mappings below 10 , Hundreds mapping for below 100, another mapping (DpLessthanThousand)to store ready calculated values (to improve efficiency using **dynamic programming** approach) and another mapping Groups, for different groups of numbers above 100 ,(e.g. thousands, millions). After input sanitization and validation in isInputValid, the number is passed 3 digits at a time, and passed to a recursive function ConvertLessThanThousand where it will retrieve the word equivalent from the mappings if value is less than 10 or 100 it recursively divides the remainder. The final result is stored in the DpLessThanThousand in key value pairs where the key is the number and value is the result.If the number(key) exists in the dictionary, it returns the results instead. For every 3 digit, we increment the groupIndex and get the corresponding group (e.g. Thousands, Millions) and concatenate the results. I opted for a **recursive approach** for ConvertLessThanThousand for readability, and for the range of values allowable, it would not risk a recursive stack overflow.To extend the algorithm, we would just need to add an additional group to the group mapping, therefore increasing readability and maintainability.  
  
Another approach is to iteratively parse each digit in the number and check if the number is in the millions, thousands, or hundreds and concatenate the results.This approach reduces the time complexity to O(1) and space complexity for O(1).However, this approach will require extension of the if logic, which reduces the code readability and maintainability.

Test plan

Test cases were drawn from black box testing methods. Methods used were equivalent partitioning and boundary value analysis.  
The ranges identified are:

Range 1: <0  
Range 2: 0- 999,999,999.99(max value accepted by programme)

Range 3: 1000000000 (exceed max value)  
For the cents denomination, the ranges identified are:  
Range 1: 0 or null-99  
  
Non numeric input  
No input

Note: The unit test was implemented for the service layer as that is where the business logic resides.Further improvements can be made to introduce unit test also at the controller layer to test if controller is calling the right function, testing for model validation(e.g. if input is null/empty string) and the exact error messages returned to the view.

## Test Cases

### Test Case 1: Range 1: Negative number

| **Input** | **Expected Output** |
| --- | --- |
| -123.45 | Raise Format Exception(“Input cannot be negative”) |
|  |  |
|  |  |

### Test Case 2:Range 2: Valid Range(0-0- 999,999,999.99)

| **Input** | **Expected Output** |
| --- | --- |
| 123.45 | ONE HUNDRED AND TWENTY-THREE DOLLARS AND FORTY-FIVE CENTS |
| 0.00 | ZERO DOLLARS AND ZERO CENTS |
| 100 | ONE HUNDRED DOLLARS AND ZERO CENTS |
| 999999999.99 | NINE HUNDRED AND NINETY-NINE MILLION NINE HUNDRED AND NINETY-NINE THOUSAND NINE HUNDRED AND NINETY-NINE DOLLARS AND NINETY-NINE CENTS |

### Test Case 3: Range 3 (Exceed max limit)

| **Input** | **Expected Output** |
| --- | --- |
| 1000000000 | Raise FormatException(“Input exceed maximum allowed value 999,999,999.99); |
|  |  |
|  |  |
|  |  |

### Test Case 4:Cents:Range 1: Valid Range(0 or null-99)

| **Input** | **Expected Output** |
| --- | --- |
| 123.45 | NINE HUNDRED AND TWENTY-THREE DOLLARS AND FORTY-FIVE CENTS |
|  |  |
| 100 | ONE HUNDRED DOLLARS AND ZERO CENTS |
| 0.00 | ZERO DOLLARS AND ZERO CENTS |
|  |  |

### Test Case 5:Cents:Range 2: Invalid Range(>99)

| **Input** | **Expected Output** |
| --- | --- |
| 99.999999 | raiseFormatException(“Input is not in the correct format”) |
|  |  |
|  |  |
| 99.100 | raiseFormatException(“Input is not in the correct format”) |
|  |  |

### Test Case 6:Non numeric

| **Input** | **Expected Output** |
| --- | --- |
| abc | Input is not a valid number |
|  |  |
|  |  |
| 123.45.29 | Input is not a valid number |

### Test Case 7:No input

| **Input** | **Expected Output** |
| --- | --- |
| - | Input not set |
|  |  |

## Test Environment

* Operating System: Windows  
  Browser: Chrome and Edge