Design Rationale

# Introduction

In this document, I explain the design decisions I made while developing the NumberToWord Converter solution. This solution is an ASP.NET Core Web API that converts numerical values into their word representations, structured according to Clean Architecture principles. I will also describe why I chose this approach and why I decided against other possible solutions.

# My Chosen Approach

* 1. **Clean Architecture**
     + **Separation of Concerns:** I divided the solution into separate projects:
       - API Layer: This handles HTTP requests, responses, and global exception handling.
       - Application Layer: This encapsulates the core business logic for number conversion.
       - Helpers: This contains utility functions and extension methods.
       - Tests: This project holds the unit tests to ensure reliability and correctness.
     + **Maintainability & Scalability:** By separating the concerns, I ensured that the codebase remains maintainable and scalable. Each layer can evolve independently without affecting the others.
     + **Testability:** Placing the business logic in the Application layer and isolating it from infrastructure concerns made unit testing straightforward.
  2. **Dependency Injection & Configuration**
     + Dependency Injection (DI): I used DI to inject configuration values and services, ensuring loose coupling and adherence to SOLID principles.
     + Configuration via JSON: I chose to store settings such as place values in “appsettings.json”. This approach makes the solution flexible and easier to modify without altering the code.
  3. **Global Exception Handling**
     + Robust Error Management: I implemented a global exception handler as middleware in the API layer. This ensures that unexpected errors are handled gracefully and consistently, improving the user experience.
  4. **HTML Client**
     + For the client-side user interface I chose a simple HTML web page client designed with CSS and Java script.

# Reasons Against Alternative Solutions

* 1. **Business Logic in the API Layer**
     + **Problem:** Initially I had included the business logic directly in API project. I later realised that this would couple the business rules with HTTP-specific code.
     + **Reasoning:** This approach would have made testing more difficult and reduced reusability, leading to a hard to maintain, bloated controller.
     + **My Decision:** I created the Application project and moved the business logic to this layer to ensure a clear separation of concerns, better organization and straightforward testing.
  2. **Approaches for Number Processing**

I had evaluated 3 approaches for number processing.

* + - The first one that I designed relied on a technique which had explicit IF Else checks for millions, thousands, hundreds, etc.
    - The second approach I took was to use Logarithms to figure out how many 10n where present in the number and then based on the number of “n” decide the millions, thousands and so on.
    - The final approach that I decided to implement is a grouping approach that converts the input number into three-digit segments. Here the location of the segment decides the place value.

Below is the rationale of why I chose the third technique

* + - **Problem**:
      * While the first approach was very simple, but it made the code very verbose and was not easily scalable. For e.g. if the code was written as Billion being the highest place value and in future if user desired to go to a larger number, then it will involve code change.
      * In the second approach while logarithms can determine a number's magnitude, I found it required additional handling to avoid rounding errors and complicated the handling of edge cases (e.g., exact powers of ten or floating-point precision issues). The code also became relatively complex to understand compared to the 1st approach.
    - **Reasoning**:
      * **Keep It Simple**: Converting numbers to words requires precise extraction of digit groups. I found that division and modulus operations provide a more straightforward and error-resistant method. I used this logic for figuring out words for Hundreds.
      * **Scalability**: I implemented a dictionary of place values for the larger positions. By Iterating over groups, the current implementation can handle larger numbers more dynamically, if the PlaceValues dictionary is configured with enough keys.
    - **My** **Decision**: I chose the arithmetic approach for its clarity and reliability when breaking numbers into manageable groups (hundreds, thousands, etc.).
    - **Potential Drawbacks:** One consideration is that the current approach is dependent on the PlaceValues dictionary. If this dictionary isn’t maintained correctly or isn’t extended for numbers beyond a certain magnitude, it could limit the range of numbers that can be converted. With explicit checks, the behaviour is fully controlled by the logic in code rather than external configuration.
  1. **Hardcoding Values vs. Configurable Settings**
     + **Problem:** Hardcoding values like place values within the code reduces flexibility and increases the risk of errors when changes are needed.
     + **Reasoning:** Configuring these values via JSON (e.g., in `appsettings.json`) allows me to update settings without modifying the code and supports different locales or number formats if needed.
     + **My Decision:** I opted to inject configuration settings at runtime, enhancing both maintainability and flexibility.

# Conclusion

I chose to build the NumberToWord Converter solution using Clean Architecture, dependency injection, JSON configuration, and arithmetic-based number processing. This approach has provided me with a maintainable, scalable, and testable system. I rejected alternatives, such as embedding business logic in the API layer or using a logarithm-based method, due to concerns over maintainability, testability, and precision.