The Problem

The task was to convert a numerical currency amount like "123.45" into its written English equivalent: "ONE HUNDRED AND TWENTY THREE DOLLARS AND FORTY FIVE CENTS".

Algorithm Design

The Three Lookup Tables

The entire English number system from zero to 999 trillion (constraint) can be expressed using just three lookup tables:

- **Table 1 (Ones):** 0 to 19 these are all unique words
- **Table 2 (Tens):** TWENTY, THIRTY, FORTY, FIFTY, SIXTY, SEVENTY, EIGHTY, NINETY (8 unique words for the tens)
- **Table 3 (Scale):** HUNDRED, THOUSAND, MILLION, BILLION, TRILLION (scale labels applied to groups of three digits)

Step-by-Step Algorithm

For the dollar part of the input, the algorithm works as follows:

1. **Validation and Split:** Separated the input into dollars and cents at the decimal point. Validated that the input contains only digits and at most one decimal point.

2. **Padding to multiple of 3:** Padded the dollar string with leading zeros until its length is a multiple of 3. For example, "1542" becomes "001542".

3. **Split into chunks of 3:** Split the padded string into groups of three digits. "001542" becomes ["001", "542"].

4. **Assign scale index:** Each chunk gets a scale index from right to left. "542" gets index 0 (no label), "001" gets index 1 (THOUSAND).

5. **Convert each chunk:** Convert each 3-digit chunk to words using the three lookup tables. Skip chunks with a value of zero entirely.

6. **Apply the AND rule:** Insert "AND" when the remaining part of a number is less than 100. For example, 105 becomes "ONE HUNDRED AND FIVE" but 500 becomes "FIVE HUNDRED".

7. **Combine dollars and cents:** Join the dollar words and cent words with "AND": "X DOLLARS AND Y CENTS".

Approaches Considered and Why I Decided Against Them

Stack-Based Approach

My initial instinct was to push each digit onto a stack and pop them one by one to build the output. While this works conceptually, I decided against it for a few reasons.

A stack follows Last In First Out ordering, which means you push digits left to right and pop them (three at a time) right to left and add a label to that chunk. But then you have to reassemble the output left to right anyway so you end up going forwards, backwards, and forwards again for no reason. This adds unnecessary memory usage and complexity without any benefit. A simple array with an index achieves the same result more cleanly.

Existing Libraries or NuGet Packages

There are existing libraries that handle number-to-words conversion. I decided against using any of them because the assessment requirements.

Recursive Approach

A recursive solution is possible you could recursively break numbers down into smaller pieces. However, for this problem the iterative chunk-based approach is simpler to read, easier to debug, and avoids potential stack overflow issues with very large numbers.

Technical Decisions

C# with ASP.NET Core Minimal APIs

The assessment specified C# as the preferred language. I used ASP.NET Core with the Minimal APIs approach, which lets you define endpoints in a few lines. For a single endpoint API this is much cleaner and easier.

Separation of Concerns

I separated the conversion logic (NumbersToWords.cs) from the API layer (Program.cs). This follows the Single Responsibility Principle each file has exactly one job. The converter converts numbers. The API handles HTTP requests.

React Frontend

React was chosen as specified in the requirements. I kept the frontend simple. A single component with four pieces of state: the input number, the result, any error message, and a loading flag. The loading state disables the button and shows "Converting..." while the API call is in progress, which improves the user experience by making it clear something is happening. (A subtle nostalgic sound effect was added to enhance the user experience.)

Input Validation

The backend validates that:

- Input is not empty or whitespace
- Input contains only digits and at most one decimal point
- Cents are at most two decimal places
- Number does not exceed 999 trillion (the maximum supported scale)

Inputs like ".45" are treated as "0.45", rather than rejecting it.

Unit Testing with xUnit

I wrote 13 unit tests using xUnit covering edge cases, and invalid inputs. Writing the tests actually caught a real bug, a scale index issue where zero-value chunks were being skipped

but the position counter wasn't being decremented, causing incorrect scale labels on subsequent chunks.

90000009 was being interpreted as NINETY MILLION AND NINE THOUSAND DOLLARS.

The tests caught this immediately and it was fixed by decrementing the scale index by 2 for 000 chunks.

CI Pipeline with GitHub Actions

I added a GitHub Actions CI pipeline that automatically builds the project and runs all unit tests on every push to main.