Approach and Reasoning for Numbers to Words

Agam Sood

September 2025

Summary

The goal was to build a small, reviewable web application that converts a dollar amount (e.g. `123.45`) into uppercase English words (e.g. `ONE HUNDRED AND TWENTY-THREE DOLLARS AND FORTY-FIVE CENTS`). I implemented a minimal Razor Pages UI backed by a small C# class library that contains the conversion logic. The design favours correctness, testability, and clarity and is not overly complex.

1 Chosen Approach

Stack: Three projects: (i) Razor Pages web app (*UI + handlers*), (ii) Class library (*converter*), (iii) xUnit tests. (and Solution Items for README.md)

Why Razor Pages

For a single page with a form, Razor Pages keeps the web layer small and explicit. There is no need for extra controllers/views structure or a SPA runtime. Handlers (`OnGet/OnPost`) and the PageModel keep the flow easy to review.

Why a separate class library

The converter lives in a tiny library exposing one method (`MoneyToWords(decimal)`). This enforces separation of concerns, makes unit testing straightforward, and allows reuse without having to play with the UI.

Why `decimal` and server-first validation

`decimal` avoids binary floating-point pitfalls (*e.g.* 2.675). Server-side validation is authoritative (format, range, rounding). Light client checks improve UX but can be bypassed, so they never replace server rules.

2 Alternatives Considered (and not chosen)

Console-only app
Pros: fast to write, trivial to test, can be a starting point to check the algorithm **Cons:** no web UI, and does not meet the requirements of the test. **Decision:** Not chosen because no web UI.

ASP.NET Core MVC
Pros: familiar pattern; great for multi-page apps. **Cons:** extra controllers/views for one page. **Decision:** Razor Pages is leaner for the scope of this test.

Web API + SPA (React/Vue)
Pros: strong separation, modern FE. **Cons:** NPM toolchain, routing, bundling, overhead for a single form. **Decision:** overkill here.

Third-party number-to-words library
Pros: speed. **Cons:** the task favours original implementation, style control ("AND"/hyphenation) and licensing can also conflict. **Decision:** implement from scratch.

Client-only validation
Pros: snappy UX. **Cons:** bypassable; locale quirks; no security. **Decision:** keep client checks as hints, enforce rules on server.

3 Trade-offs Accepted

- Chose simplicity over a richer front end stack; easier to review and maintain.
- Limited to billions with cents, which is appropriate for the problem and keeps code compact.

4 Development Process (Order of Work)

I kept the workflow short and incremental so each step was easy to review.

1. **Clarify rules & scope.** "AND" inside hundreds; hyphenation for 21 to 99; singular/plural; non-negative; two decimals; sensible maximum.
2. **Scaffold solution.** Create three projects: web (Razor Pages), library, tests. Wire project references.
3. **I**n the library, implement `MoneyToWords` as a stub that throws `NotImplementedException`. This let me compile early and wire the page.
4. **Wire the page.** Build the `Index.cshtml` form and `Index.cshtml.cs` handler that calls the converter, plus friendly error messages.
5. **Server-side validation.** Add format/parse/rounding/range checks on the server (source of truth). Keep light client filtering for UX.
6. **Write tests first for core behaviours.** Create xUnit cases for hyphenation, "AND" rule, plural, cents, rounding edges, and large scales.
7. **Implement converter in small layers.**
   - `TwoDigitToWords` (0 to 99),
   - `HundredsToWords` (0 to 999) with "AND",
   - `IntToWords` (thousand/million/billion),
   - `MoneyToWords` (currency wording, rounding, join).
8. **Polish UI.** Small Bootstrap card and hints.
9. **Docs.** README (build/run/test), this rationale, and comments in the code.

I treated the converter as a pure function and leaned on **deterministic xUnit tests**.

5 Why this works well

The structure is very simple, the web surface is thin, and the converter is a small, pure unit with comprehensive tests. Reviewing is easy, can run, check a few inputs (including edge cases), and immediately see correctness without having to go in too hard.