Ethan Bolton

TechTest Approach and Reasoning for My Solution

Approach Taken

I implemented an ASP.NET Web API in C# for the backend logic, with HTML, CSS, and JavaScript for the front end, served from the `wwwroot` folder for user interaction. Any file within `wwwroot` can be served directly to users without additional processing by the server.

The backend API receives a numerical input as a string, processes it, and returns the corresponding text representation. The front-end HTML page uses JavaScript to interactively call this API and display the results to the user.

The controller route "convert" splits the number into dollars and cents. Each is converted into words separately, with appropriate suffixes (“DOLLARS” or “CENTS”). The API also handles singular and plural forms correctly.

The conversion uses a lookup-based approach for numbers below 20 (e.g., "ONE", "TWO", "THIRTEEN") and predefined words for the tens (e.g., "TWENTY", "THIRTY") to construct the word representation for larger values. The algorithm recursively processes thousands, millions, billions, and beyond, ensuring correct conversion for even very large numbers (up to quintillions). It also handles edge cases like zero and supports negative numbers. The application is hosted using Docker, allowing consistent deployment across different environments.

I also created a test file to extensively test the application's logic, verifying that various examples match the expected output, including invalid numbers, out of range values, and the existence of the index.html file to ensure the content can be viewed.

Reasons for Selecting This Approach

I chose this approach for its modularity and separation of concerns. Using an ASP.NET Web API for backend logic allows the server-side code to focus purely on computation and data transformation, while the front-end handles user interaction. This separation makes the application easier to maintain, test, and extend. I am also not familiar with using Java.

The lookup-based approach for converting numbers to words simplifies the logic for small numbers and handles large numbers effectively. Using predefined words for numbers below 20 and for the tens makes the conversion efficient and readable. Recursively processing larger units ensures concise, maintainable code for very large numbers.

I also included support for negative numbers and large values to make the application robust, improving the user experience and what I believe would lead to better quality for a customer. Invalid or extreme inputs are handled gracefully, providing clear error messages when needed.

Serving the front-end files from the `wwwroot` folder leverages the built-in capabilities of ASP.NET to serve static files efficiently, reducing complexity and enhancing performance. This allowed me to focus on core functionality instead of building a separate mechanism for serving static content, such as Nginx or Apache. By containerising the application, I simplified the management of dependencies and ensured reliable execution across different host systems. This approach also makes deployment, scaling, and management in production easier, using Render as the hosting provider.

Considered Alternatives

One alternative I considered was performing the number-to-words conversion entirely on the client side using JavaScript. However, I rejected this because it could lead to inconsistencies, particularly with edge cases and very large numbers. Additionally, managing complex business logic on the client side would increase potential security vulnerabilities and make the front-end code harder to maintain and would defeat the purpose of having the backend API.

Another option I considered was using Blazor for the front-end development. Blazor would have allowed me to write the entire application in C#, which could have simplified development and reduced the need for JavaScript. However, I decided against this approach to keep the front-end stack more traditional and accessible, given my familiarity with HTML, CSS, and JavaScript.

I considered using a third-party library, such as Humanizer, for the conversion logic. While this could have saved development time, I decided to implement it myself for greater flexibility and control and because it made more sense given its a test.

I also spent time trying to configure a Nginx web server to host the frontend and route to the .NET API, all within Docker. After many issues, I decided to scrap that idea in favor of a simpler approach, which I later discovered could be achieved using the `wwwroot` folder.