TESTS

This document explains the automated test strategy, how to run the test suite, and what each test covers for the **Number to Words Converter**.

🧪 Test Stack

- **Test framework:** xUnit
- **Assertions:** FluentAssertions
- **Project layout:** Unit tests live under `NumberToWordsApp.Tests` in `UnitTests/`

▶ How to Run

From the solution root:

```
dotnet restore
dotnet test --configuration Release
```

To run with detailed logging:

```
dotnet test -v n
```

To run a single test class:

```
dotnet test --filter
"FullyQualifiedName~NumberToWordsApp.Tests.UnitTests.NumberToWordsConver
terTests"
```

📐 Test Strategy

We follow an **Arrange–Act–Assert** style with **small, isolated unit tests**:

1. **Model tests** verify default state and property behavior.
2. **Controller tests** verify MVC flow, validation, and view models.
3. **Converter tests** verify the core business logic for number→words, including pluralization and edge cases.

This keeps the pyramid wide at the unit layer and ensures fast feedback.

✅ What's Covered

1) `ConvertViewModelTests`

Purpose: Ensure the MVC view model has sensible defaults and behaves correctly when properties change.

Key checks:

- Default constructor initializes all properties to `null`/`false`.
- Properties are settable (e.g., `NumberInput`, `Result`, `HasError`, `ErrorMessage`, `RequestId`).
- Accepts empty / whitespace / null inputs for `NumberInput`.
- `HasError` toggles correctly; `Result` and `ErrorMessage` are nullable.

2) `ConvertControllerTests`

Purpose: Verify end-to-end controller behavior for GET/POST of the main Convert page.

Key checks:

- **GET** `/Convert` returns `ViewResult` with an empty `ConvertViewModel`.
- **POST** with valid numbers returns expected strings:
  - `"123"` → `ONE HUNDRED AND TWENTY-THREE DOLLARS`
  - `"123.45"` → `ONE HUNDRED AND TWENTY-THREE DOLLARS AND FORTY-FIVE CENTS`
  - `"0"` → `ZERO DOLLARS`
  - `"0.01"` → `ZERO DOLLARS AND ONE CENT`
  - `"-100"` → `MINUS ONE HUNDRED DOLLARS`
- **POST** with empty or invalid input returns error:
  - Error message: `please enter a valid number`
  - Result: empty string
  - `HasError: true`
- Decimal precision handling example:
  - Input `"123.456"` yields `... FORTY-FIVE CENTS` (rounded to two decimals).
- Large number handling:
  - `"1000000"` → `ONE MILLION DOLLARS`.

3) `NumberToWordsConverterTests`

Purpose: Validate the core conversion logic (singular/plural, negatives, decimals, boundaries).

Key checks:

- **Whole numbers** across ranges (0, teens, tens, hundreds, thousands, millions).
- **Decimals:** cents appended with correct pluralization (e.g., `0.99` → `NINETY-NINE CENTS`).
- **Negatives:** prefix with `MINUS` and keep grammar correct.

- **Whole-dollar amounts:** omit cents wording entirely (`100.00` → `ONE HUNDRED DOLLARS`).
- **Singular vs plural forms:**
  - `1` → `ONE DOLLAR`
  - `1.01` → `ONE DOLLAR AND ONE CENT`
  - `2.01` → `TWO DOLLARS AND ONE CENT`

🧭 Test Case Matrix (Representative)

Area	Input	Expected Output / Behavior
Model defaults	—	All properties `null` except `HasError=false`.
GET /Convert	—	`ViewResult` with empty `ConvertViewModel`.
Valid whole	`123`	`ONE HUNDRED AND TWENTY-THREE DOLLARS`.
Valid decimal	`123.45`	`... AND FORTY-FIVE CENTS`.
Rounding	`123.456`	`... AND FORTY-FIVE CENTS` (two decimals).
Large number	`1000000`	`ONE MILLION DOLLARS`.
Empty / invalid	`""`, `"abc"`	`HasError=true`, message `please enter a valid number`, result empty.
Negative	`-10.50`	`MINUS TEN DOLLARS AND FIFTY CENTS`.
Singular cents	`0.01`	`... ONE CENT`.
Plural cents	`0.99`	`... NINETY-NINE CENTS`.

🧩 Adding New Tests

1. Create a new test class in `NumberToWordsApp.Tests/UnitTests/`.
2. Use xUnit `[Fact]` for single cases and `[Theory]/[InlineData]` for multiple inputs.
3. Prefer **FluentAssertions** for readable assertions.
4. Keep tests deterministic—avoid culture-dependent formatting in assertions.

Example skeleton:

```
using Xunit;
using FluentAssertions;

public class MyNewTests
{
```

```
    [Theory]
    [InlineData("2500.10", "TWO THOUSAND FIVE HUNDRED DOLLARS AND TEN
CENTS")]
    public void Convert_ShouldHandleNewEdgeCases(string input, string
expected)
    {
        var result =
NumberToWordsApp.Models.NumberToWordsConverter.Convert(decimal.Parse(inp
ut));
        result.Should().Be(expected);
    }
}
```

📈 Future Enhancements

- **Property-based testing** (FsCheck) to fuzz random values across ranges.
- **Culture variants** (e.g., British vs US wording conventions) behind feature flags.
- **Boundary coverage** for maximum supported magnitude and precision.
- **Performance micro-benchmarks** for very large test sweeps.

❓ Troubleshooting

- Restore packages first (`dotnet restore`) if tests fail to build.
- Ensure SDK is aligned with the project's target framework.
- If assertion messages show case mismatches, verify the converter always produces **uppercase** (as expected by tests).
