**React Unit Tests**

**Introduction to Unit Testing with Vitest**

In software development, especially within the blockchain and web3 space, correctness is paramount. Errors can lead to significant consequences, including financial loss or security breaches. Testing provides a safety net, allowing developers to iterate, refactor, and add features with confidence, knowing that existing functionality remains intact.

Unit testing is the foundational layer of a solid testing strategy. It involves testing individual, isolated pieces of code – known as "units" – typically functions or components, separate from the rest of the application. By verifying that each small unit works correctly according to its specification, we can catch bugs early and build more reliable systems. This lesson introduces unit testing for a utility function within a React/TypeScript application using the Vitest testing framework.

**Setting Up Your Testing Environment**

To write and run unit tests, we need a testing framework. We'll use Vitest, a modern, fast, Vite-native testing framework. We also need a couple of helper packages:

1. vitest: The core testing framework.
2. jsdom: Provides a simulated browser Document Object Model (DOM) environment, essential for testing code that might interact with browser APIs (even if our initial utility doesn't, it's good practice for React apps).
3. vite-tsconfig-paths: A Vite plugin that allows Vitest to understand TypeScript path aliases (e.g., @/utils) defined in your tsconfig.json.

These are development dependencies, meaning they are only needed during development and testing, not for the final production build. Install them using the -D flag:

pnpm add -D vitest jsdom vite-tsconfig-paths

**The Unit Under Test:**calculateTotal

Let's consider a simple utility function designed to calculate a total sum from a string containing numbers. This string might use newlines or commas as separators, contain extra whitespace, or even include invalid entries that should be ignored.

Here's the function, located in src/utils/calculateTotal/calculateTotal.ts:

// src/utils/calculateTotal/calculateTotal.ts

export function calculateTotal(amounts: string): number {

// Split by both commas and newlines, then clean up the results

const amountArray = amounts

.split(/[\n,]+/) // Split on commas or newlines (one or more)

.map(amt => amt.trim()) // Remove whitespace around each value

.filter(amt => amt !== '') // Remove empty strings

.map(amt => parseFloat(amt)); // Convert string to float (NaN if invalid)

​

// Sum valid numbers (filter out NaN)

return amountArray

.filter(num => !isNaN(num)) // Keep only numbers (filter out NaN)

.reduce((sum, num) => sum + num, 0); // Sum valid numbers, starting from 0

}

This function takes the amounts string, splits it, cleans up each potential number, attempts to parse them into floating-point numbers, filters out any resulting NaN values (Not-a-Number), and finally sums the valid numbers using reduce. The export keyword makes it available for use in other files, including our test file.

**Writing Your First Unit Test**

A common convention for unit tests is **co-location**: placing the test file directly next to the source file it tests. For our calculateTotal.ts function, we'll create a test file named calculateTotal.test.ts in the same directory (src/utils/calculateTotal/).

Here's a basic structure for our test file using Vitest's syntax:

// src/utils/calculateTotal/calculateTotal.test.ts

import { describe, expect, it } from 'vitest';

import { calculateTotal } from './calculateTotal'; // Import the function to test

​

// 'describe' groups related tests for the 'calculateTotal' function

describe('calculateTotal', () => {

​

// 'it' defines a specific test case or scenario

it('should sum numbers separated by newlines', () => {

const input = '100\n200\n50';

const expectedOutput = 350;

// 'expect' makes an assertion: does the actual output match the expected output?

expect(calculateTotal(input)).toBe(expectedOutput);

});

​

it('should sum numbers separated by commas', () => {

expect(calculateTotal('100,200,75')).toBe(375);

});

​

it('should handle a single number', () => {

expect(calculateTotal('500')).toBe(500);

});

​

it('should return 0 for an empty string', () => {

expect(calculateTotal('')).toBe(0);

});

​

});

* describe(name, callback): Creates a block that groups together several related tests under a common name.
* it(name, callback): Defines an individual test case. The name should clearly state what specific behavior is being tested.
* expect(actualValue).matcher(expectedValue): This is the core assertion mechanism. expect takes the value produced by your code (actualValue). It's chained with a matcher function (like toBe) that compares the actualValue to the expectedValue. If they match, the test passes; otherwise, it fails.

**Leveraging AI for Test Generation (Use with Caution!)**

AI tools like DeepSeek can assist in generating test code. You can provide the function's source code and prompt the AI to write Vitest unit tests. This can speed up the initial drafting of test cases.

**However, it is absolutely critical to treat AI-generated tests with skepticism.** You *must* thoroughly review, understand, and verify any code produced by an AI.

* **Verification is Non-Negotiable:** AI might misunderstand requirements, miss edge cases, or generate tests that wrongly pass even if the underlying code is incorrect.
* **Understand the Intent:** Ensure the generated tests accurately reflect the *intended* behavior of your function.
* **"Garbage In, Garbage Out":** If the function logic is flawed, AI might generate tests that pass against that flawed logic. Testing AI-generated code with unverified AI-generated tests is a recipe for false confidence.

Use AI as an assistant to generate boilerplate or suggest test cases, but always perform rigorous human review and correction.

**Expanding Test Coverage and Debugging**

Good unit tests cover various scenarios, including edge cases and invalid inputs. Let's add more tests for calculateTotal:

// Continuing in src/utils/calculateTotal/calculateTotal.test.ts

describe('calculateTotal', () => {

// ... previous tests ...

​

it('should handle mixed delimiters and extra whitespace', () => {

expect(calculateTotal(' 100 ,200\n 300 ')).toBe(600);

});

​

it('should ignore invalid entries and empty lines', () => {

expect(calculateTotal('100\n\n200,abc,\n,300')).toBe(600);

});

​

it('should handle floating-point numbers', () => {

expect(calculateTotal('10.5, 20.25')).toBe(30.75);

});

​

// Example of debugging a potentially incorrect assumption:

it('should handle numbers mixed with text correctly', () => {

// parseFloat('12three') actually returns 12, not NaN.

// parseFloat('abc12') returns NaN.

// parseFloat('123.45.67') returns 123.45 (stops at second decimal).

expect(calculateTotal('12three\n45,abc12,123.45.67')).toBe(12 + 45 + 123.45); // 180.45

});

});

Notice the last test case, should handle numbers mixed with text correctly. Initially, one might assume (or an AI might generate a test assuming) that 12three is entirely invalid and evaluates to NaN. However, JavaScript's parseFloat("12three") actually returns 12. It parses the leading numeric part. parseFloat("abc12") correctly returns NaN. parseFloat("123.45.67") returns 123.45.

This highlights the debugging process:

1. Run the tests.
2. Observe a failure (e.g., Expected 168.45 but received 180.45).
3. Examine the failing test case and the function's logic (parseFloat).
4. Understand *why* the actual output differs from the expected output.
5. Decide if the function's behavior is correct or needs changing. In this case, parseFloat's behavior is standard.
6. Update the test's expected value to match the *correct and desired* behavior of the function (12 + 45 + 123.45 = 180.45).

**Configuring Vitest**

Vitest needs a configuration file to understand our project setup, especially things like TypeScript paths and the testing environment. Create a vitest.config.mts file (using the .mts extension for ES Module syntax) in the project root:

// vitest.config.mts

import { defineConfig } from 'vitest/config';

import tsconfigPaths from 'vite-tsconfig-paths';

​

export default defineConfig({

plugins: [

// Apply vite-tsconfig-paths plugin to handle TS path aliases like @/

tsconfigPaths()

],

test: {

// Set the test environment to jsdom to simulate browser APIs

environment: 'jsdom',

// Define patterns for files/directories to exclude from testing

exclude: [

'\*\*/node\_modules/\*\*',

'\*\*/dist/\*\*',

'\*\*/test/\*\*', // Exclude potential separate E2E test folders

'\*\*/playwright-report/\*\*',

'\*\*/test-results/\*\*'

],

deps: {

// List dependencies that might cause issues if externalized by Vitest

// Process these directly during tests. Important for libraries like wagmi.

inline: ['wagmi', '@wagmi/core'],

},

},

});

* plugins: [tsconfigPaths()]: Integrates the vite-tsconfig-paths plugin so Vitest recognizes aliases from tsconfig.json.
* test.environment: 'jsdom': Configures Vitest to run tests in a simulated browser environment provided by jsdom.
* test.exclude: Specifies an array of glob patterns to ignore files or directories (like node\_modules or dedicated end-to-end test folders).
* test.deps.inline: Tells Vitest to process certain dependencies directly. This can be crucial for libraries like wagmi that might not work correctly if Vitest tries to externalize them.

**Running Your Tests**

To easily run your tests, add a script to your package.json:

// package.json

{

// ... other configurations ...

"scripts": {

"anvil": "anvil",

"dev": "vite",

"build": "tsc && vite build",

"start": "vite preview",

"lint": "eslint .",

"test:unit": "vitest" // Add this script

},

// ... dependencies ...

}

Now you can execute all your unit tests from the command line:

pnpm run test:unit

Vitest will discover and run all files matching its default test pattern (including \*.test.ts), applying the configuration from vitest.config.mts, and report the results (passes and failures).

**Organizing Tests and Utility Exports**

We've chosen **co-location** for our unit tests, placing calculateTotal.test.ts next to calculateTotal.ts. This makes it easy to find the tests relevant to a specific piece of code. Other strategies exist (like a single top-level tests/ directory, common in Python/Solidity, or tests within source files, common in Rust), but co-location works well for unit tests in TypeScript/JavaScript projects. End-to-end (E2E) tests are often placed in a separate top-level directory (e.g., test/ or e2e/), as configured in our vitest.config.mts exclusion rules.

To simplify importing utility functions like calculateTotal into other parts of the application (like React components), we can use a **barrel file**. Create an index.ts file inside the src/utils/ directory:

// src/utils/index.ts

export { calculateTotal } from './calculateTotal/calculateTotal';

// Add exports for other utility functions from this directory here

// export { anotherUtil } from './anotherUtil/anotherUtil';

This file simply re-exports modules from its directory. Now, instead of importing like this:

import { calculateTotal } from '@/utils/calculateTotal/calculateTotal';

You can use a cleaner import path:

import { calculateTotal } from '@/utils';

**Integrating the Tested Function into a React Component**

With our calculateTotal function thoroughly tested and easily importable via the barrel file, we can confidently use it in our React components. For example, in an AirdropForm component that takes a list of amounts:

// Example usage in: src/components/AirdropForm.tsx

import { useState, useMemo } from 'react';

import { calculateTotal } from '@/utils'; // Import using the barrel file path

​

// Inside your AirdropForm component function:

function AirdropForm() {

const [amounts, setAmounts] = useState<string>(""); // State for the textarea input

​

// Calculate the total using our tested utility function.

// useMemo ensures this calculation only runs when the 'amounts' state changes.

const total: number = useMemo(() => {

return calculateTotal(amounts);

}, [amounts]); // Dependency array: recalculate only if 'amounts' changes

​

return (

<form>

<textarea

value={amounts}

onChange={(e) => setAmounts(e.target.value)}

placeholder="Enter amounts, separated by commas or newlines..."

/>

<div>

<strong>Total Amount: {total}</strong>

</div>

{/\* Rest of the form \*/}

</form>

);

}

​

export default AirdropForm;

​

Here, we import calculateTotal from @/utils. The component uses useState to manage the input string amounts. The calculateTotal function is wrapped in useMemo. This React hook memoizes the result of the calculation. It will only re-run calculateTotal if the amounts value (listed in the dependency array [amounts]) has changed since the last render, preventing unnecessary recalculations and optimizing performance.

By following these steps – writing the function, setting up the testing environment, writing comprehensive tests (potentially with reviewed AI assistance), configuring the runner, and finally integrating the validated function – we build more robust and maintainable applications.