HyperionDev

React – Testing a React App

Task

Visit our website

Introduction

To deliver high-quality code, thorough application testing is essential. In this task, you will focus on two formal testing methods: snapshot tests and unit tests. Additionally, you will gain a basic understanding of coverage reports. Using Vitest, a modern testing framework, you will learn to effectively test your React applications to ensure they function as expected and meet desired behavior standards.

Testing

Testing helps ensure that your code works correctly and that components are bug-free. There are various ways of testing code. Some of these include:

- **Manual testing**

  Trying to execute a piece of code manually to see if it does what you expect. You have done manual testing by now. `console.log()` statements are often used in manual testing to check whether variables contain the data that you expect or whether a function is performing as required.

- **Documented manual testing**

  Ensures that you have a documented plan for conducting tests.

- **End-to-end testing**

  Automated tests that simulate the user's experience.

- **Unit tests**

  Instead of testing the system's functionality as a whole, this type of test focuses on testing one unit or a single function, class, component, etc., at a time.

- **Integration testing**

  Once you're confident that individual units work separately, you can do integration testing to ensure these units work together.

- **Snapshot testing**

  Ensures that your UI does not change unexpectedly.

All the tests listed above check the functionality of your code. Other aspects of your code should also be tested to improve their quality. These include:

- **Performance testing**

  Tests the stability and responsiveness of the code. Performance testing checks how well your code is working and how fast it is.

- **Usability testing**

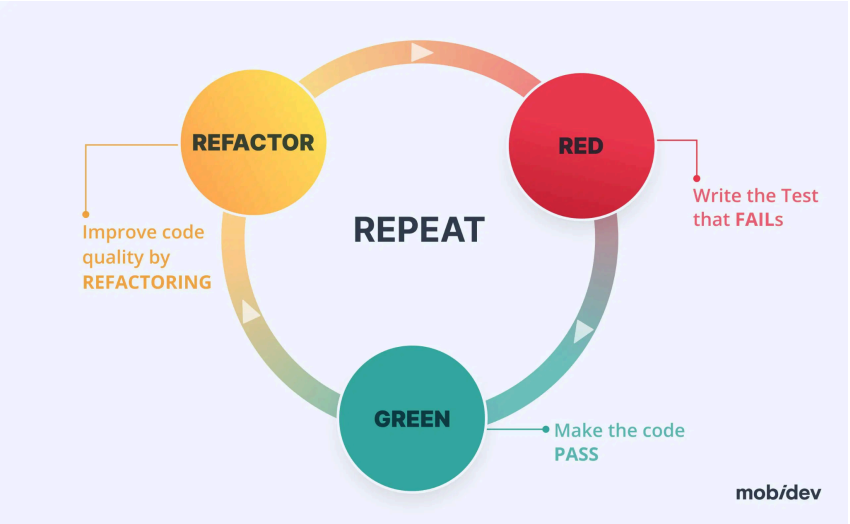
  Tests the way a user interacts with the system. It doesn't only determine the functionality of the code, but also its simplicity and intuitiveness.

- **Security testing**

  This involves tests determining any potential security flaws within the system.

It is considered good practice to create tests early on in the design or development process. Test-driven development (TDD) is an approach to software development where you are encouraged to write tests before you write the actual code.

In TDD, the Red-Green-Refactor cycle is an iterative process that promotes code maintainability:



The Red-Green-Refactor cycle in TDD (Koba, 2024)

- **Red**

  Write a test for a new feature. This test will initially fail since the new feature is not implemented yet.

- **Green**

  Add the minimal code necessary to make the failing test pass. At this stage, the main focus is functionality rather than code optimisation.

- **Refactor**

  Optimise the code without altering its behaviour while ensuring that all tests continue to pass.

- **Repeat**

  Once the refactoring is done, repeat the cycle for each new feature in the app.

Vitest: A testing framework for React

Testing frameworks play a crucial role in ensuring the reliability and quality of your code. **Vitest** is a next-generation testing framework built on top of Vite. As such, it is meant to provide a testing experience that integrates seamlessly with projects powered by Vite. One of its standout features is its speed, achieved by leveraging Vite's fast bundling capabilities.

In short, Vitest positions itself as the test runner of choice for projects using Vite, combining speed and ease of use to streamline the testing process.

Extra resource

Other compelling features of Vitest include: Hot Module Replacement, instant watch mode, and out-of-the-box support for modern syntax (such as ES Modules, TypeScript, and JSX). Please see the **documentation** for a comprehensive overview of all its features.

Recap: Bootstrapping your React app with Vite

In order to create a new React app (using the JavaScript variant) with Vite, run the following from the command line:

```
npm create vite@latest my-react-app -- --template react
```

After this has been done, navigate into your React project and install the necessary dependencies:

```
cd my-react-app
npm install
```

You should see that `src/App.jsx` is populated as follows:

```
import { useState } from 'react'
import reactLogo from './assets/react.svg'
import viteLogo from '/vite.svg'
import './App.css'
function App() {
  const [count, setCount] = useState(0)
  return (
    <>
      <div>
        <a href="https://vite.dev" target="_blank">
          <img src={viteLogo} className="logo" alt="Vite logo" />
        </a>
        <a href="https://react.dev" target="_blank">
          <img src={reactLogo} className="logo react" alt="React logo" />
        </a>
      </div>
      <h1>Vite + React</h1>
      <div className="card">
        <button onClick={() => setCount((count) => count + 1)}>
          count is {count}
        </button>
        <p>
          Edit <code>src/App.jsx</code> and save to test HMR
```

```
        </p>
      </div>
      <p classNameName="read-the-docs">
        Click on the Vite and React logos to learn more
      </p>
    </>
  )
}
export default App
```

Configuring your app for testing

In the following sections, you'll find instructions for installing packages, adding a test script, and configuring Vite.

Package installations

To set up the testing environment, install the following packages as development dependencies:

```
npm install --save-dev vitest @testing-library/react jsdom
@testing-library/jest-dom
```

Here is a small breakdown of all the packages:

- **`vitest`**

  A modern testing framework designed to integrate seamlessly with Vite-powered projects (in this case, a React App).

- **`@testing-library/react`**

  It offers utilities to test React components by simulating user interactions. According to **ReactTutorial**, it encourages testing components in a way that closely mirrors how users interact with the application, leading to more meaningful and robust tests.

- **`@testing-library/jest-dom`**

  Provides custom 'matchers' that extend the built-in assertions. This allows for more expressive and readable tests when interacting with the DOM. For example:

  - `expect(element).toBeInTheDocument();`

    Asserts that the element is present in the document

  - `expect(element).toHaveTextContent('Hello, World!');`

    Verifies that an element contains the specified text content.

  For a comprehensive list of matchers, please refer to the `jest-dom` **documentation** on GitHub.

- **`jsdom`**

- According to the **documentation**, this is a pure JavaScript implementation of the DOM and HTML standards. It allows us to run tests involving DOM manipulation without requiring a real browser, which makes it valuable for testing React components in a Node.js environment.

At this stage, the `"devDependencies"` key of the `package.json` file should look something like the following (note the bold highlights):

```
"devDependencies": {
    "@eslint/js": "^9.17.0",
    "@testing-library/jest-dom": "6.6.3",
    "@testing-library/react": "16.1.0",
    "@types/react": "^18.3.18",
    "@types/react-dom": "^18.3.5",
    "@vitejs/plugin-react": "^4.3.4",
    "eslint": "^9.17.0",
    "eslint-plugin-react": "^7.37.2",
    "eslint-plugin-react-hooks": "^5.0.0",
    "eslint-plugin-react-refresh": "^0.4.16",
    "globals": "^15.14.0",
    "jsdom": "26.0.0",
    "vite": "^6.0.5",
    "vitest": "2.1.8"
  }
```

Add a test script

To integrate Vitest into your project, add a **`"test"`** script to your **`package.json`** file:

```
"scripts": {
    "dev": "vite",
    "build": "vite build",
    "lint": "eslint .",
    "preview": "vite preview",
    "test": "vitest"
  }
```

This addition allows you to execute tests using the following command in your terminal:

```
npm run test
```

When you run this command, Vitest will search for all the test files in your project and execute all defined test cases.

Take note

As per the **Vitest documentation**, tests must contain **`".test."`** or **`".spec."`** in their file name. Because of the limited scope of this task, we will stick to the **`".test."`** format.

Configure `vite.config.js` for Vitest integration

Modify the `vite.config.js` file to the following:

```
import { defineConfig } from 'vite';
import react from '@vitejs/plugin-react';
// https://vite.dev/config/
export default defineConfig({
  plugins: [react()],
  test: {
    globals: true,
    environment: 'jsdom',
    setupFiles: './src/setupTests.js',
  },
});
```

- **`test`**:

  Contains fields that define the behaviour of the testing environment. Please see the **Vitest Configuration** page for more exhaustive descriptions.

  - **`globals: true`**

    When set to true, it makes test functions like **`describe`** and **`expect`** globally available. This means you don't need to import these functions explicitly in each test file, simplifying your test setup.

  - **`environment: 'jsdom'`**

    This allows tests to simulate a browser-like setting within Node.js using JSDOM. This is essential for testing React components that interact with the DOM, as it provides the necessary browser APIs.

  - **`setupFiles: './src/setupTests.js'`**

    Specifies the path to the setup file that runs before each test suite. It is typically used for configuring the testing environment.

Extra resource

Vitest reads the `vite.config.js` file to align with your Vite application's setup, including plugins and aliases. However, if you prefer a separate configuration for testing, you can create a `vitest.config.js` file, which will override the settings in `vite.config.js`.

The `setupTests.js` file

Inside the src folder, create a `setupTests.js` file and add the following code:

```
import '@testing-library/jest-dom'; // Extend the jest matchers.
```

Here, we import the `jest-dom` package to provide custom 'matchers' specifically designed for testing DOM elements. By doing so, we don't need to include this import statement in each individual `.test.` (or `.spec.`) file, therefore streamlining the test setup process.

Take note

Vitest offers compatibility with most of the Jest API libraries. As such, it's a suitable drop-in replacement for Jest. This is preferable, because the seamless integration between Vite and Vitest provides a unified configuration for both development and testing environments. This reduces the complexity compared to using, for example, Jest.

Writing tests

There are two types of tests we're going to look at: **snapshot tests** and **unit tests**.

Snapshot tests

A snapshot is a serialised representation of the DOM structure of a component at the time the test is first executed. Snapshot testing is commonly used for front-end user interfaces to detect unintended changes to the UI.

The process involves capturing a snapshot of the interface's appearance at a specific stage. This snapshot is then stored and used for comparisons during future test runs. If the current DOM structure deviates from the stored snapshot, the test will fail, indicating that the interface has changed.

Follow along the code below to create a simple example of a snapshot test:

Inside the src folder, create a file named `App.snapshot.test.jsx` and add the following code:

```
import { render } from '@testing-library/react';
import { describe, test, expect } from 'vitest';
import App from './App';
describe('App Component', () => {
  // Test to ensure the rendered App component matches the saved snapshot:
  test('matches the snapshot', () => {
    const { asFragment } = render(<App />); // Render the App component.
    expect(asFragment()).toMatchSnapshot(); // Compare with stored snapshot.
  });
});
```

Important points from the example above:

- **How snapshots work**:
    - `render(<App />)` renders the app component in a virtual DOM.
    - `asFragment()` generates a lightweight representation of the DOM structure.
    - `expect(asFragment()).toMatchSnapshot()` compares the current DOM output to the saved snapshot.
- **First versus subsequent runs**:
    - In the first run, a snapshot is created and stored inside the `__snapshot__` folder. In this example, `App.snapshot.test.jsx.snap` is created. Have a look at this file to get an idea of how the UI component was serialised.
    - In subsequent runs, the current output is compared to the saved snapshot. If there's a difference, the test fails, indicating a change in the component's DOM structure.

To run the snapshot test, execute the following command in your terminal:

```
npm run test
```

After running the command, the terminal output should look similar to the following:

```
✓ src/App.snapshot.test.jsx (2)
   ✓ App Component (1)
     ✓ matches the snapshot
```

Making an intentional UI change

Inside `App.jsx`, let us make a small change to the `<button>` element:

```
<button onClick={() => setCount((count) => count + 1)}>
    The count is {count}
</button>
```

Now, let's run the snapshot test again with `npm run test App.test.jsx`, and inspect the terminal output:

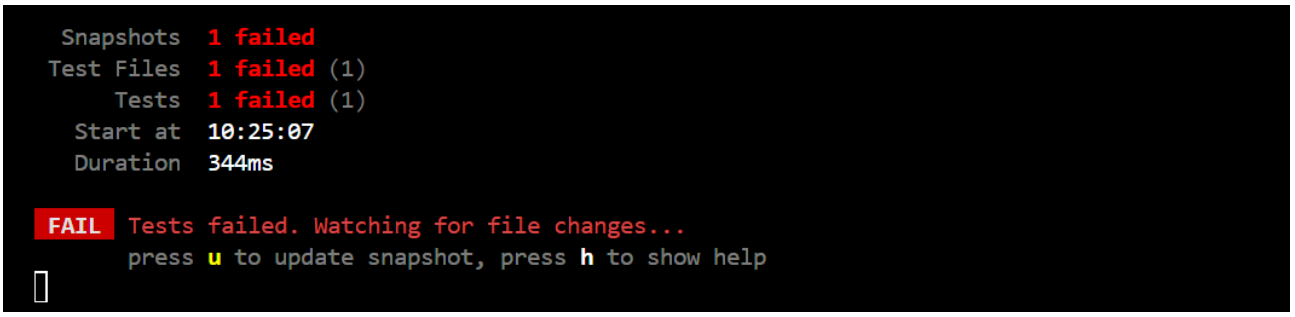
```
❯ src/App.snapshot.test.jsx (1)
   ❯ App Component (1)
     × matches the snapshot
_____________________________ Failed Tests 1 _____________________________
FAIL  src/App.snapshot.test.jsx > App Component > matches the snapshot
Error: Snapshot `App Component > matches the snapshot 1` mismatched
- Expected
+ Received
@@ -26,11 +26,11 @@
    </h1>
    <div
      class="card"
    >
      <button>
-       count is 0
+       The count is 0
      </button>
      <p>
        Edit
        <code>
          src/App.jsx
 ❯ src/App.snapshot.test.jsx:9:26
      7|   test('matches the snapshot', () => {
      8|     const { asFragment } = render(<App />);
      9|     expect(asFragment()).toMatchSnapshot();
       |                          ^
     10|   });
     11| });
```

Notice that the test **failed** because the stored snapshot no longer matches the current-rendered output. This error occurred at Line 9 in `src/App.snapshot.text.jsx` during the snapshot comparison.

The *expected* and *received* output (causing the mismatch) are respectively as follows:

- count is 0
- The count is 0

Since the change to the `<button>` element was **intentional**, the snapshot needs to be updated. As indicated in the screenshot below, press "u" to update the snapshot.



Note that you need to be in **watch mode** when selecting this option, which means the test runner must be actively running in interactive mode. Since Vitest operates in watch mode by default, it will automatically rerun tests whenever you save changes to the test file.

Alternatively, you could try using the following command to regenerate the snapshot:

```
npm test src/App.snapshot.test.jsx -- -u
```

This reruns the tests for the file and updates the snapshot to match the current output. After running this, the terminal output should confirm that the test now passes:

```
✓ src/App.snapshot.test.jsx (1)
  ✓ App Component (1)
    ✓ matches the snapshot
```

In the `src/__snapshots__/App.snapshot.test.jsx.snap` file, the following change should appear in Lines 33–35 of the serialised output:

```
<button>
    The count is 0
</button>
```

This updated snapshot reflects the intentional change made to the `<button>` element in `App.jsx`, and ensures the snapshot test now aligns with the modified UI.

Take note

If a UI change was **not intentional**, then we should **not update the snapshot**. Instead, we should fix any unintended changes in the **component code** itself, and re-run the tests with `npm run test`.

To summarise: Always validate whether a UI change is intentional before deciding how to proceed with your snapshots. This practice is crucial for maintaining test reliability and avoiding hidden issues in your code.

Unit tests

Unit testing involves testing individual components of software to verify that they function as intended. These components, or "units", are the building blocks of an application, such as functions or React components. By validating each unit, we can identify bugs early in the development cycle, improving the quality and reliability of the application.

In Vitest, the `test()` function and `expect()` object are used to write unit tests. In the following example, we will create unit tests to check whether the "count" button (see picture below) increments correctly. This UI can be found by visiting **http:localhost:5173/** (or whatever port Vite is serving your app on) after executing `npm run dev` in the terminal.

Create a file named `App.test.jsx` inside the src folder and add the following code:

```jsx
import { render, screen, fireEvent } from '@testing-library/react';
import { describe, test, expect } from 'vitest';
import App from './App';
describe('App Component', () => {
  // Test to verify the App renders the initial count correctly.
  test('renders the initial count correctly', () => {
    render(<App />); // Render the App component.
    const button = screen.getByRole('button', { name: /count is 0/i });
    expect(button).toBeInTheDocument(); // Assert that the button is present
  });
  describe('increments count on button click', () => {
    test('increments count to 1 after the FIRST click', () => {
      render(<App />); // Render the App component.
      const button = screen.getByRole('button', { name: /count is 0/i });
      fireEvent.click(button); // Simulate the FIRST button click.
      expect(button).toHaveTextContent('count is 1');
    });
    test('increments count to 2 after the SECOND click', () => {
      render(<App />); // Render the App component.
      const button = screen.getByRole('button', { name: /count is 0/i });
      fireEvent.click(button); // Simulate the FIRST button click.
      fireEvent.click(button); // Simulate the SECOND button click.
      // Check if the button's text updates to reflect the new count.
      expect(button).toHaveTextContent('count is 2');
    });
  });
});
```

Take note

In the above code, the trailing `i` in `{ name: /count is 0/i }` is a flag that makes the match in the `getByRole()` call case-insensitive. This ensures the test does not fail due to differences in letter casing, making it more robust. If you don't want case insensitivity, you can simply remove the `i` flag: `const button = screen.getByRole('button', { name: /count is 0/ });`

Key concepts explained:

Let's review key testing concepts for React to ensure clear and accurate checks:

- **React-specific testing**
    - Unlike plain vanilla JavaScript functions, testing a React component requires utilities from **@testing-library/react**. In our simple example, we imported three utility functions: `render`, `screen`, and `fireEvent`:
        - **`render`**: Renders a React component into a virtual DOM.
        - **`screen`**: Allows you to access and interact with elements rendered by the render function.
        - **`fireEvent`**: Simulates user interactions such as clicks, keypresses, etc.
- **Grouping tests with `describe`**
    - The outer **`describe`** groups tests for the App component, making the test suite organised.
    - The inner `describe` focuses on button-click behaviour, keeping related tests together.
- **`test`**
    - **`test`** is used to define an individual test case in which we check a specific functionality or behaviour.
- **Assertions with `expect()`**
    - **`.toBeInTheDocument()`** verifies an element is present in the DOM.
    - **`.toHaveTextContent()`** checks if an element contains specific text. In this case, we expect the button to have the text "count is 1" after the first click, and "count is 2" after the second click.
    - **`expect`** is the core of assertions, where you define what you expect to happen.

As usual, run `npm run test` to execute the tests. Here is the expected output:

```
✓ src/App.test.jsx (3)
  ✓ App Component (3)
    ✓ renders the initial count correctly
    ✓ increments count on button click (2)
      ✓ increments count to 1 after the FIRST click
      ✓ increments count to 2 after the SECOND click
```

Code hack

Vitest states that `it()` is an alias for `test()`. This means you can use `it()` interchangeably with `test()`. However, maintaining consistency across your codebase is crucial. In this task, and the provided code snippets, we use `test()`, as it is semantically more explicit. However, you are free to use `it()` if you prefer.

Spot check

1. Based on the section "**Configuring your app for testing**", what steps are necessary to set up and run tests for a React app using Vitest (assuming you already bootstrapped your React app with Vite)?
2. According to the callouts on Pages 4 and 9, what are the key advantages of using Vitest over other testing frameworks like Jest?
3. How does the `environment: 'jsdom'` setting help when testing React components?
4. What is a snapshot in the context of testing, and where is it typically stored?
5. Name, and briefly describe, three commonly used utility functions of the `@testing-library/react` package used for simulating user interactions?

Instructions

Please inspect the **example** folder provided with this task. This **example** project focuses on testing a React app using `expect` assertions in both unit and snapshot tests. It also briefly touches on generating coverage reports.

In the following, we briefly specify the purpose of each `.test.` file:

- **`src/api/api.test.js`**

  Verify that the `fetchData` function retrieves information accurately from the PokéAPI.

- **`components/Child/Child.test.jsx`**

  Test the Child React component to ensure its rendered output matches a previously saved snapshot.

- **`utils/multiply.test.js`**

  Test the multiply function to confirm it correctly multiplies input numbers and returns NaN for invalid inputs.

Before executing the following tests, remember to run `npm install` to have all the package dependencies sorted out.

After running the `npm run test`, the terminal output should look similar to the following:

```
✓ src/api/api.test.js (1) 1196ms
✓ src/utils/multiply.test.js (2)
✓ src/components/Child/Child.test.jsx (1)
Test Files  3 passed (3)
     Tests  4 passed (4)
```

In summary, all three test files passed successfully, therefore validating the functionality of the `fetchData` function, the consistency of the Child component's UI, and the behaviour of the multiply function. In total, we performed four tests: one in `api.test.js`, two in `multiply.test.js`, and one in `Child.test.jsx`. The output below (from the terminal) provides detailed information for each test case.

Code hack

Since Vitest operates in watch mode by default, tests are automatically rerun whenever you save changes to the test file. If you want to run tests for a specific file manually, you can specify the name of the file as in the following instance:

```
npm run test filename.test.js
```

This is useful if you want to debug a particular test file without running the entire test suite.

For `src/api/api.test.js`:

```
 ✓ src/api/api.test.js (1) 1182ms
   ✓ PokeAPI data check (1) 1181ms
     ✓ the first ability of Squirtle is "torrent" 647ms
```

For `src/components/Child/Child.test.jsx`:

```
 ✓ src/components/Child/Child.test.jsx (1)
   ✓ React Child.jsx Component (1)
     ✓ matches the snapshot
```

For `src/utils/multiply.test.js`:

```
✓ src/utils/multiply.test.js (2)
   ✓ multiply function (2)
     ✓ correctly multiplies three numbers: 2 x 3 x 4 = 24
     ✓ returns NaN when invalid input is passed
```

Coverage reports – from code to confidence

Coverage reports provide an in-depth analysis of how effectively your tests cover your codebase, offering a critical tool for identifying gaps and ensuring code quality. In general, coverage reports provide developers insights to:

- **Assess the test suite quality**

  Confirm that critical areas of the codebase are well-tested.

- **Identify coverage gaps**

  Highlight untested or undertested code areas to reduce hidden bugs.

- **Track testing progress**

  Use metrics to measure and improve test coverage over time, fostering a culture of high-code quality.

The following metrics are typically included in coverage reports:

- **Line coverage**

  The percentage of code lines executed during testing.

- **Branch coverage**

  The percentage of decision points (e.g., if or switch statements) tested.

- **Function coverage**

  The percentage of functions or methods executed.

- **Statement coverage**

  The percentage of individual executable statements tested.

Extra resource

In the industry, it's a common practice to use integration servers to automatically run tests on the entire codebase whenever a commit is pushed to a version control system like GitHub. This process provides clear insights into the stability of specific Git branches, helping teams identify potential issues early. As highlighted in an **Atlassian article**, enforcing a coverage threshold – typically between 80% and 90% – can further ensure high code quality by preventing the integration of inadequately tested code.

The **Vitest coverage guide** gives detailed steps on configuring and customising your coverage reports. In the subsections below, simple instructions are given to create a coverage report for our **example** project.

Add a "coverage" script

In the `package.json` file, under the `"scripts"` section, add the following `"coverage"` field:

```
"coverage": "vitest run --coverage"
```

Install a coverage provider

A coverage provider is a tool responsible for calculating and generating the test coverage reports. Vitest supports two main **coverage providers**: **v8** and **Istanbul**. By default, v8 is typically used and is a popular choice.

Before you install the coverage provider, it's important to ensure that the Vitest version matches the required version for `@vitest/coverage-v8`. Version conflicts could occur if they don't align.

You can now proceed by installing it as a development dependency as follows:

```
npm install --save-dev @vitest/coverage-v8
```

Generate a coverage report

From the command line, run the following:

```
npm run coverage
```

The terminal output should look similar to the following:

```
 % Coverage report from v8
----------------------|---------|----------|---------|---------|-------------------
File                  | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
----------------------|---------|----------|---------|---------|-------------------
All files             |   45.45 |    57.14 |      60 |   45.45 |
 src                  |       0 |        0 |       0 |       0 |
  App.jsx             |       0 |        0 |       0 |       0 | 1-31
  main.jsx            |       0 |        0 |       0 |       0 | 1-10
 src/api              |      75 |       50 |     100 |      75 |
  api.js              |      75 |       50 |     100 |      75 | 18-23
 src/components/Child |     100 |      100 |     100 |     100 |
  Child.jsx           |     100 |      100 |     100 |     100 |
 src/utils            |     100 |      100 |     100 |     100 |
  multiply.js         |     100 |      100 |     100 |     100 |
----------------------|---------|----------|---------|---------|-------------------
```

Notice that for the **`api.js`** file, we have the following stats:

- **75%** of **Statements** covered.
- **50%** of **Branches** covered.
- **100%** of **Functions** covered.
- **75%** of **Lines** covered (Lines 18-23 in `api.js` are *not* covered)

At this stage, a `coverage` folder should have been automatically generated in our **example** project. If we launch the `coverage/src/api/api.js.html` file and view it in a web browser (`http://127.0.0.1:5500/coverage/src/api/api.js.html`), we should see the following:

Notice how uncovered lines are highlighted in red, while uncovered branches are marked in yellow. The yellow highlight indicates that our `api.test.js` does not cover the `catch` block in which an error is thrown if data cannot be retrieved for a specific `character` name.

To add coverage to our `catch` clause, we can expand our `api.test.js` to also contain a test case in which we check whether garbage input (such as when `"foobar"` is passed to `fetchData`) throws an error:

```
import fetchData from './api';
import { test, expect, describe } from 'vitest';
// Investigate Line 5 of the `pretty-print` version of
// https://pokeapi.co/api/v2/pokemon/squirtle/
describe('PokeAPI data check', () => {
  test('the first ability of Squirtle is "torrent"', async () => {
    const ability = await fetchData('squirtle');
    expect(ability).toBe('torrent');
  });
  test('garbage input throws an error', async () => {
    await expect(fetchData('foobar')).rejects.toThrow();
  });
});
```

With our `api.test.js` code expanded, let's see what changes there are in our `api.js` coverage report after rerunning `npm run coverage` and reopening `coverage/src/api/api.js.html`:

Notice now that, with the addition of the `test('garbage input throws an error', ...)` clause, our entire codebase (that is, 100%) for `api.js` is now covered.

Conclusion

While we should aim for comprehensive test coverage in our code, it should be noted that 100% coverage does not guarantee bug-free code. Nevertheless, one should focus on covering critical paths and edge cases in your code to maximise the effectiveness of your test suite.

Spot-check answers

1. Based on the section "**Configuring your app for testing**", what steps are necessary to set up and run tests for a React app using Vitest (assuming you already bootstrapped your React app with Vite)?
    - Install the necessary testing packages, which includes the following: `@testing-library/react @testing-library/jest-dom jsdom vitest`
    - Add the test configuration block in `vite.config.js` (or `vitest.config.js` if you prefer).
    - Create a `setupTests.js` file to streamline the test setup.
    - Add a test script to the `"scripts"` section of the `package.json`
2. According to the callouts on Pages 4 and 9, what are the key advantages of using Vitest over other testing frameworks like Jest?
    - Seamless integration with Vite, therefore allowing for a single configuration pipeline for both development and testing.
    - Built-in support for ES modules, TypeScript, and JSX without requiring additional configuration.
    - Offers Jest-compatible APIs, reducing the need for external libraries.
3. How does the `environment: 'jsdom'` setting help when testing React components?
    - It simulates a browser-like environment in Node.js. This eliminates the need for a real browser, making it faster to test React components that rely on DOM manipulation.

4. What is a snapshot in the context of testing, and where is it typically stored?
    - A snapshot is a serialised representation of a component's DOM structure at a specific point in time.
    - It is stored in a `__snapshots__` folder alongside the test file, and has a `.snap` extension (for example: `App.snapshot.test.jsx.snap`).
5. Name and briefly describe three commonly used utility functions of the `@testing-library/react` package used for simulating user interactions?
    - `render`: Renders a React component into a virtual DOM.
    - `screen`: Allows you to access and interact with elements rendered by the render function.
    - `fireEvent`: Simulates user interactions such as clicks, keypresses, etc.

Practical task

Familiarising yourself with the app and setting up the test environment

1. Open the accompanying **task** folder, and inspect the `src/App.jsx` file. Pay attention to the following details:
    - Line 7 initialises the count state to (not 0).
    - In Lines 9-12, we have a decrement function that 'decreases' the count with each button click, but prevents the count from going below 0.
    - In Line 26, the decrement function is bound to the button's `onClick` event.
2. Setup your testing environment by:
    - Installing the necessary packages.
    - Adding a test clause to the `vite.config.js` file.
    - Adding a test script in the `package.json` file.
    - Checking if Vitest executes after running `npm run test`.

Unit testing

3. Create a new file named `App.test.jsx` inside the **src** folder. Write unit tests to confirm that the count does not go below zero after three button clicks. (**TIP**: Simulate at least three button clicks in your test).

Snapshot testing

4. Inside the `src/components/` folder, create a new file named `Greetings.jsx`. This component should:
    - Accept a name prop, and render a personalised greeting, for example: "Hello, Alice!"
    - Default to "Hello, World!" if no name prop is provided.
    - **TIP**: You can insert `/* eslint-disable react/prop-types */` at the top of `Greetings.jsx` to disable the validation of React props.
5. Inside the same folder, create a test file named `Greetings.test.jsx` and write a snapshot test to:
    - Test the Greetings component with two different name prop values, e.g., "Mpho" and "Lydia".
    - Verify that the component handles the default behaviour correctly when no name prop is provided.

Generate a coverage report

6. Inside `package.json`, configure the `"coverage"` script as follows: `"coverage": "vitest run --coverage"`.
7. Install the **v8** coverage provider as a development dependency.
8. Once the unit and snapshot tests are coded, execute `npm run coverage`. Note that for this task, you are required to have 100% test-coverage specifically for `App.jsx` and `Greetings.jsx`.
9. Take a screenshot of the coverage report generated in the console. Save it in the parent folder of this task and name it either `coverage_screenshot.png` or `coverage_screenshot.jpg`.

Summary of files that need to be completed and included in the submission

- `src/App.test.jsx`
- `src/components/Greetings.jsx`
- `src/components/Greetings.test.jsx`
- `coverage_screenshot.png` or `coverage_screenshot.jpg`

Important: Before submitting your task, delete the `node_modules` folder. This asset can be easily regenerated by the reviewer by running `npm install` from the command line. Once this step is complete, click the "Request review" button on your dashboard to submit your task for review.

Share your thoughts

Please take some time to complete this short feedback **form** to help us ensure we provide you with the best possible learning experience.

Reference list

Koba, S. (2024). *Reclaim your app's potential: All you need to know about application refactoring*. MobiDev. **https://mobidev.biz/blog/application-refactoring-strategy-techniques-best-practices**