Published in inspiringbrilliance

Pradip Hudekar Follow
Oct 12, 2020 · 6 min read · Listen

Save

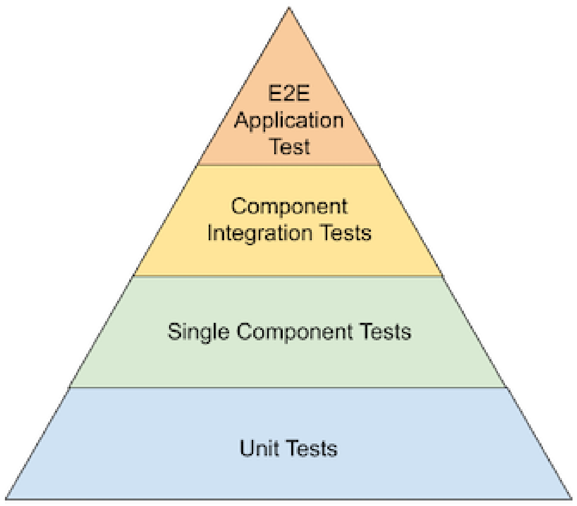
Testing Single Page Applications

There are many ways to test the frontend of a web application. Most widely used methods are to test the application as a user using Selenium Web-driver or similar libraries. This means that you have to write tests for each and every scenario as a user workflow.

This made our testing slow and delayed the whole Test Driven Development cycle. That leads to most UI developers to only write application code and let the UI Automation teams write tests for them. It is not difficult to see then that the dependency on multiple teams delayed the feedback cycle. Thus delaying the continuous delivery process.

But today we are mainly going to focus on testing Single Page Applications (SPA). Modern SPA frameworks like React, Angular and Vue.js use component based approach to build UI by composing smaller components together. This provides us an opportunity to test our application logic at component level as well as the complete app as a black box.

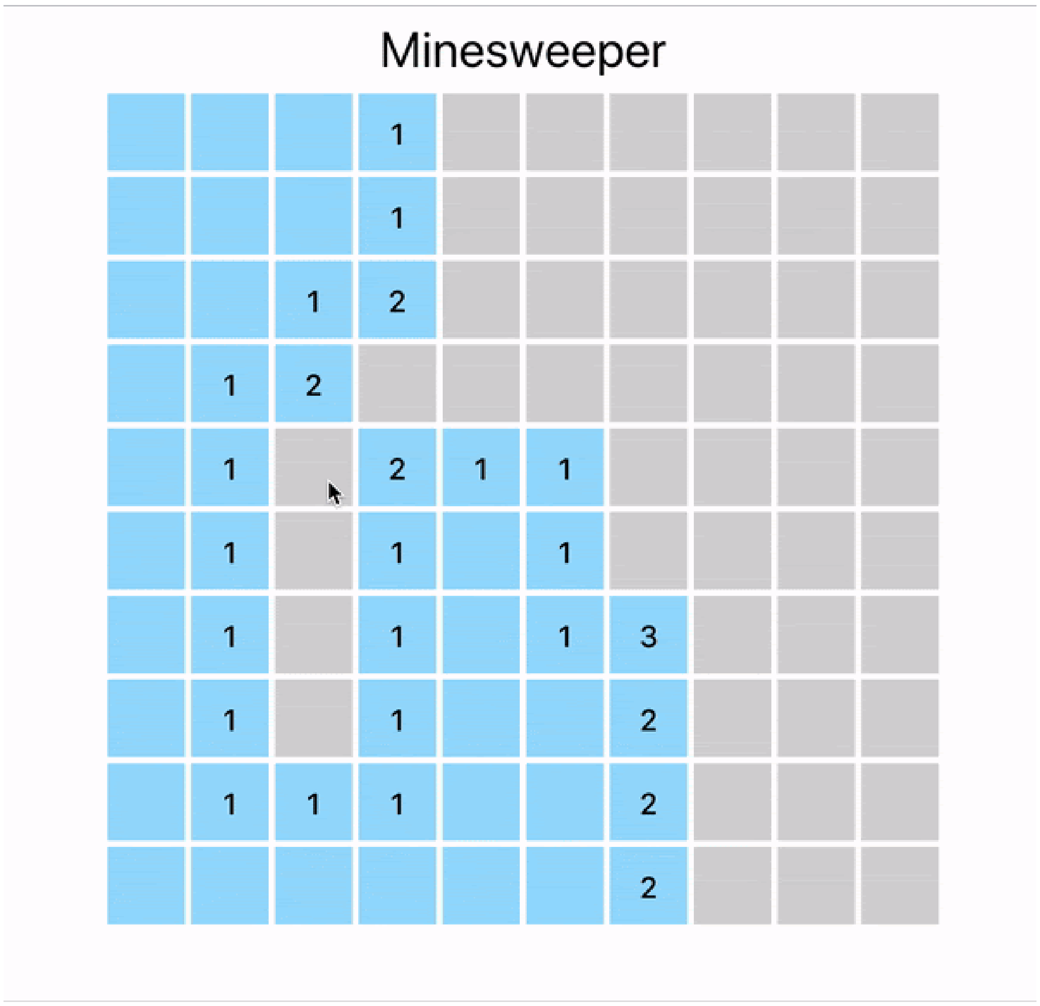
Because of this granularity of modern SPA testing our testing pyramid would usually look like following:



Let's look at each one of them from bottom to top.

To demonstrate various types of tests, I have created a sample implementation of a MineSweeper Game in React. The source code can be found at *https://github.com/phudekar/ui-testing*

> *You can play the game at https://phudekar.github.io/ui-testing/*



1. Unit Tests:

These are your tests for plain old Javascript functions and classes that you are writing as part of the application. These may include any domain classes, functional utilities or more specifically the helper functions of the UI components. Because these tests usually don't depend on any other code or library you should be able to write tests for each and every use case of these functions. There are many frameworks available in Javascript to test this kind of Javascript objects. Most commonly used are Jasmine, Mocha and Jest.

Tests under `src/lib` are an example of Javascript unit tests. These tests are independent of the UI rendering context that is using these library classes.

```
it('should explode if it has mine', () => {
    const mine = true;
    const block = new Block(new Position(1, 1), mine);
    const { mineExploded } = block.reveal([]);
    expect(mineExploded).toBeTruthy();
})

it('should not explode if it does not have mine', () => {
    const block = new Block(new Position(1, 1));
    const { mineExploded } = block.reveal([]);
    expect(mineExploded).toBeFalsy();
})
```

unit-test.js hosted with ❤ by **GitHub** view raw

2. Single Component Tests:

By definition UI components are supposed to be individual building blocks of your applications. These should encapsulate all the logic necessary to render and behave in isolation of other components. If you are writing Pure components, that means that their behaviour is predication and depends only on the properties passed to them, then we can render that component in our test environment without the boilerplate code of the rest of the application. Then we can test all kinds of use cases by changing the properties of this component.

You can use the testing frameworks Testing Library or Enzyme which work with `react-dom` to test individual components.

Tests in `src/game/Block.test.ts` are a good example of testing a single component without any other application context.

```

it('should call onRevealed after left click', () => {
    const block = new Block(position);
    const reveal = jest.fn();
    const { getByTestId } = render(<BlockComponent block={block} onReveal={reveal} />);
    const root = getByTestId(`block-${position.row}-${position.column}`);
    fireEvent.click(root);
    expect(reveal).toHaveBeenCalled();
})

it('should call toggleFlagged after right click', () => {
    const block = new Block(position);
    const toggleFlagged = jest.fn();
    const { getByTestId } = render(<BlockComponent block={block} toggleFlagged=toggleFlagged} />);
    const root = getByTestId(`block-${position.row}-${position.column}`);
    fireEvent.contextMenu(root);
    expect(toggleFlagged).toHaveBeenCalled();
})

it('should show bomb if revealed and exploded', () => {
    const { container } = render(
        <BlockComponent block={{ revealed: true, exploded: true }} />
    );
    expect(container.querySelector('.bomb')).toBeVisible();
})
```

Here you can also use shallow rendering (ie. Just render the component under test without rendering any other components that you may have used as its children). Shallow rendering is beneficial when you want to test the internal logic of a component in isolation.

3. Component Integration Tests

After we test each component in isolation it is time to check if components are being used properly in the context of another component. In the previous step we were rendering and testing only one component, but if we want to make sure that component is passing correct data to the child component and handling events raised by children then we must render our component with all its children without mocking them. This may seem that we are now dependent on the child components but that's precisely what we want to test. This kind of test helps us find any integration issues between dependent components.

In our code example, the `src/game/Board.test.tsx` is an example of Component Integration Test. The Board component uses the Block component.

```
it('should reveal block', () => {
    const rows = 5;
    const columns = 5;
    const board = new Board(rows, columns);
    const { container } = render(<BoardComponent board={board} />)
    const block = container.querySelectorAll('.block')[0];
    fireEvent.click(block);
    expect(board.blocks[0][0].revealed).toBe(true);
})

 it('should call game over on mine explode', () => {
    const rows = 5;
    const columns = 5;
    const board = new Board(rows, columns);
    board.plantMines([new Position(0, 0)]);
    const gameOver = jest.fn();
    const { container } = render(<BoardComponent board={board} onGameOver={gameOver} />)
    const blockWithMine = container.querySelectorAll('.block')[0];
    fireEvent.click(blockWithMine);
    expect(gameOver).toHaveBeenCalled();
})
```

compone-integration.test.js hosted with ❤ by GitHub   view raw

4. End to End Tests:

After testing all our components in isolation of groups we still need to make sure that our app works end to end without stubbing any of its components. The goal of end to end (e2e) tests is to test our app as a user. We can not assume any internals of the applications. We should treat it as a complete black box and assert the behaviour completely on the outcome of user actions.

As part of our previous steps to test various use cases of each individual component in unit or integration tests, e2e tests don't need to cover all the scenarios. Ideally we should write only a few e2e tests to check the overall sanity of the app. Because we are testing our app as an end user we should execute our tests by rendering our SAP in a real browser. It can sometimes cause e2e tests to take more time compared to its counterparts.

There are many ways to execute e2e tests in browser, for example using Selenium Web Driver, Puppeteer, TestCafe or Cypress

In the sample code I have used Cypress to run the e2e scenario in chrome browser. The Cypress tests are located under `cypress/integration/minesweeper/`.

A typical cypress test looks like:

```javascript
it('should toggle flag of a block', () => {
    cy.visit("/");
    cy.get(".flag").should("not.be.visible");
    const block = cy.get(".block").first();
    block.rightclick();
    cy.get(".flag").should("be.visible");
    block.rightclick();
    cy.get(".flag").should("not.be.visible");
})
```

e2e.test.js hosted with ❤ by GitHub view raw

These tests will help test most of the functionality of the application. However if you want to also verify the look and feel of the app, it can get a bit tricky. One simpler approach is to take a snapshot of the app when we are satisfied with styles and then compare it on every run to detect any changes.

These snapshots could be like a screenshot of the browser window or you can capture the rendered DOM tree to compare differences. Both of these approaches have their pros and cons. Former can capture small changes in CSS styles whereas later can point you to the precise DOM location which caused the problem. We can use Jest for DOM based snapshot testing.

In most of the real world applications, you would be making API calls to your backend services while running various scenarios. If the data returned from the server is dynamic you will not be able to verify the functionality consistently in e2e tests. To avoid this, we can either stub the api calls either via some proxy like MockServer, WireMock or use Cypress's `cy.route` and `cy.fixtures` APIs to stub dynamic and fixed responses respectively.

> *But keep in the mind the more apis you stub the chances are there for failure during real integration. I*

Conclusion:

There is a lot of focus on programmatically testing the backend functionality but UI is mostly left to be tested manually. Traditionally few of the reasons to do so are the lack of knowledge and techniques as well not having better tools to test frontend applications efficiently. But constant efforts to improve these areas have now made it possible to test almost all the aspects of a web application. If you architect the application in such a way that the components are well isolated and core business logic is separate from rendering logic it reduces the efforts to test all possible user scenarios and makes the test much faster which in turn also reduces the development feedback cycle.