Testing Your Brightlayer UI Applications

Testing is a critical component of successfully building and releasing an application. Despite the value and importance of testing, it is common for teams to overlook or neglect it. Testing should be considered from the very beginning of development, not as an afterthought once development is complete. In general, every time you write a new piece of code or update existing code, you should also write a test that verifies that your code is correct.

There are many varieties of tests, each with their own intended purpose. The most common types of tests are:

- **Unit Tests**: verify individual methods, functions, or components.
- **Integration Tests**: verify that different modules or services in your application work together.
- **Functional Tests**: verify the output of certain actions based on business requirements.
- **End-to-end (E2E) Tests**: verify that user behaviors / workflows function as expected.
- **Acceptance Tests**: verify that the entire system meets the business requirements.
- **Performance Tests**: verify that the system behaves correctly under heavy load.
- **Smoke Tests: quick**: basic checks of the major features of the application.

This document focuses mainly on Unit Tests and E2E tests, as they are the easiest to automate.

Unit Tests

Unit tests target the smallest pieces of your application (such as verifying inputs and outputs of a class method or function). These will likely make up the bulk of your tests because they are quick to write, easy to automate, and can help you achieve significant code coverage.

> *Code Coverage measures the percentage of your code that is verified by your tests.*

A unit test shouldn't have any external dependencies, such as other methods or APIs. By keeping unit tests isolated in this manner, it's easy to identify the cause of a failing test and implement a fix quickly. When you start combining features and methods together, you move into the realm of _integration testing_.

When to Write

As long as you are writing tests and achieving acceptable levels of coverage, the _when_ is not as important.

Some teams elect to follow a Test-Driven Development (TDD) methodology, where unit tests are written prior to any code. In this situation the tests actually define the desired implementation of the application and the code is written to satisfy the tests.

Behavior-Driven Development (BDD) is similar to TDD in that test cases are written prior to writing any applications code. However, these test cases focus more on the desired behavior (inputs and outputs) of the application, and less on the actual implementation. These are usually written in plain language rather than code.

The most important thing to keep in mind about tests is that they should provide confidence that the code and design are working as intended without issues in any scenario. For most teams, it's fine to write tests after features and functions are developed, so long as you write quality tests and achieve good coverage.

How to Write

When unit testing components, the most effective tests are based on what is actually rendered on the screen rather than the internal implementation logic. The test should consider the application from the user's perspective and test things the user views and interacts with.

For example, consider testing an Open Button that should open a modal dialog when clicked. Your test should check that a button with the _Open_ label is rendered on the screen, and when it is clicked, the dialog is rendered on the screen with the correct content. You may be tempted to identify the button by a class or id, or test that a particular function is called when the button is pressed - these tests are not as effective because they are more likely to break if the implementation of the component changes, even though the end behavior is the same.

- Test what the user sees (i.e.,rendered output) and what the user can do (i.e., interactions).
- If there is visual change after an interaction, test for it.
- If there is a value returned after an action, test for it.

Testing Frameworks

Angular, React, and React Native all come with built-in unit testing frameworks.

Angular

The Angular CLI comes pre-configured with Jasmine and Karma for unit testing. When you create a new project, sample tests are created in your project for you (test files are identified by the `.spec.ts` file extension). You can execute the tests by running the following in your terminal:

```sh cd your/project/root ng test ```

This will build your app and launch the browser with the test runner.

React

The Create React App CLI is pre-configured with the Jest testing framework. When creating a new project, a sample test is created in your project for you (test files are identified by the `.test.js` or `test.ts` file extension). You can execute the tests by running the following in your terminal:

```sh cd your/project/root npm run test // or yarn test ```

This will run your tests and show the pass / fail output in the terminal.

There are additional test utilities that complement Jest well. Create React App provides access to React Testing Library through the `react-dom` dependency. The Brightlayer UI team has also made use of Enzyme, a testing framework created by AirBnB, and Test Renderer.

React Native

Like React, the React Native CLI also bundles Jest into your application. Most of the testing strategies and libraries can be shared between the two frameworks. You can also consider additional libraries like React Native Testing Library.

End-to-End Testing

End-to-end tests at a step above unit tests, both in terms of complexity and what they test. They focus primarily on the flow of the application, making sure that a user's journey through the application has the expected outcomes. These tests can generally be categorized as:

- **Acceptance Tests**: verify that various features and flows meet the customer expectations.
- **Regression Tests**: verify that existing functionality is not broken when new features are added.

They should cover user stories that span multiple components and views, such as signing up for an accounts, logging in and out of the application, updating a profile, etc. These tests are not concerned with the underlying state or implementation of the application - just the end results.

End-to-end tests usually run in a browser against a live system (in the case of web, using a test runner that automates the browser). Automated E2E tests can take a long time to run because of all of the different pieces involved and the scope of the tests. You should try to optimize your testing pipeline to run tests in parallel to finish faster.

Functional tests alone will not give you enough test coverage to avoid regressions. Unit tests aim to provide code coverage depth, while functional tests provide coverage over the breadth of user test scenarios.

How to Write

End-to-end tests rely on the ability to find elements on the screen and interact with them in an automated way. This means that you need a reliable mechanism for selecting elements. Depending on your testing framework, there are different ways to select elements, including:

- **CSS Selectors**: find elements by using CSS classes or rules (e.g., `.your-class-name`)
- **Element Selectors**: find elements inside the elements (e.g., `#\@\@blui-drawerlayout-content > div > header > div > button`)
- **xPath Selectors**: find selector within xPath (e.g., `//*[@id="@@blui-drawerlayout-content"]/div/header/div/button`)
- **ID Selectors**: find attribute id within elements (e.g., `#login-button`)

> *You can read more about different selector strategies on Selenium Dev and software testing help.*

Each of these methods has their own advantages and disadvantages. A common pain point in E2E test automation is that modern Javascript platforms are constantly changing, particularly in the open source community. Many of these frameworks automatically (and dynamically) generate IDs and classes for components resulting in a constantly moving target for automation tooling. Identifying elements by these values (i.e., using IDs, CSS, or xPath locators _without_ property attributes) makes your tests brittle because they are subject to change any time a new version comes out (or even any time the page is reloaded).

One way to combat this in your own applications is to use dedicated attributes for testing (e.g., `test-id` or `dev-id`). By adding this test-specific attribute to the element, identifying the correct underlying component should be safer. Because this attribute is test-specific and the application logic does not use it, it will be less likely to change throughout the product lifecycle. Combining this approach with other selectors is a common and reliable approach for element identification.

```html

Login
WebElement click = driver.findElement(By.id("login-button")); ```

## Testing Frameworks

There are many tools available for configuring automated E2E tests. Some of the more popular tools include:

- Selenium
- Cypress
- WebDriverIO

### Selenium

Selenium is a popular automation testing suite which can be used to automate the desktop and mobile web browser interactions. You can write test code in any of the languages supported by Selenium.

Selenium test suite has several test frameworks available and each one can be customized to your project needs.

For more information, check out the browserstack selenium guide or the official selenium documentation.

There are also a number of Selenium tutorials available online to help you get started.
```

Cypress

Cypress is a popular open source testing framework that boasts support for any modern JavaScript framework. It works well for E2E tests in both Angular and React applications and is easier to use than some of the more traditional tools, such as Selenium.

For a more detailed write-up on Cypress, check out the official Cypress Documentation.

WebDriverIO

WebdriverIO is a popular Javascript based test automation framework built on top of node.js. It is an open-source project developed for the automation testing community. WebdriverIO is extendible, compatible and feature-rich.

For more information, check out the official Webdriver.io documentation and their getting started guide.
