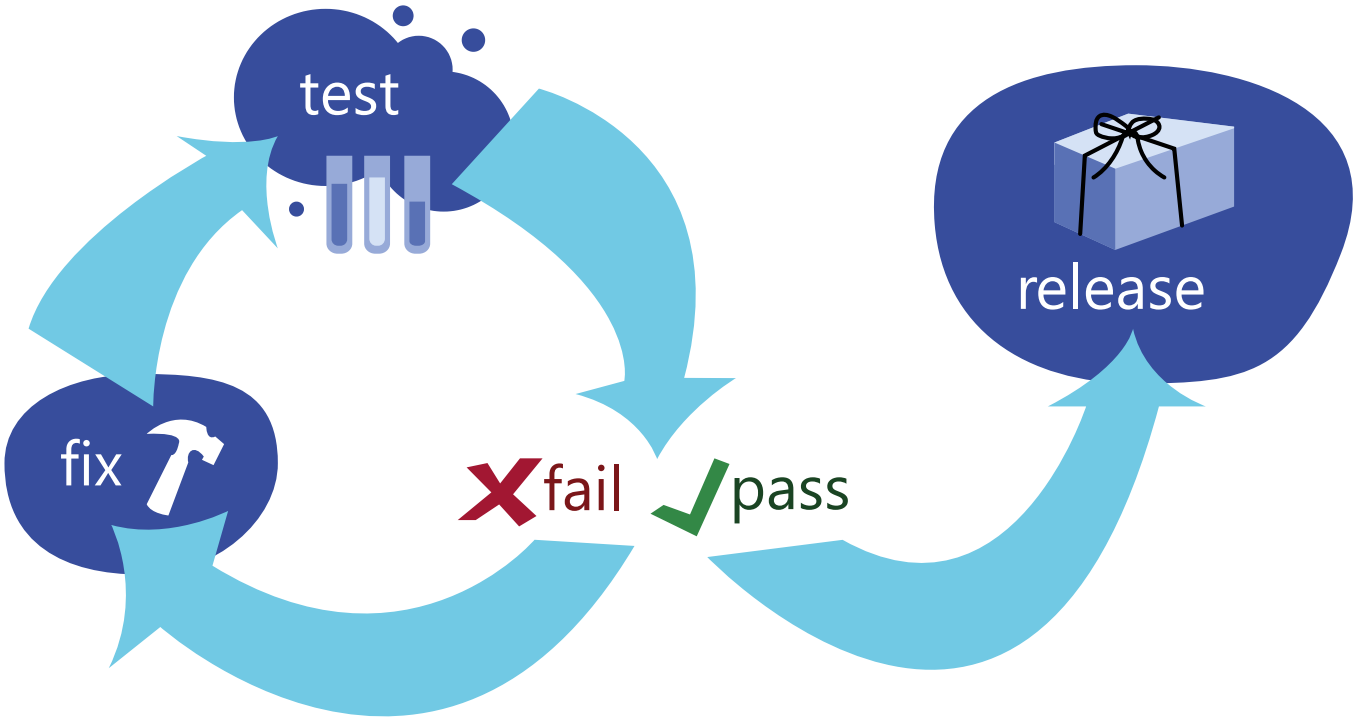
Testing

As your codebase expands, small errors and edge cases you don't expect can cascade into larger failures. Bugs lead to bad user experience and ultimately, business losses. One way to prevent fragile programming is to test your code before releasing it into the wild.

In this guide, we will cover different, automated ways to ensure your app works as expected, ranging from static analysis to end-to-end tests.



Why Test

We're humans, and humans make mistakes. Testing is important because it helps you uncover these mistakes and verifies that your code is working. Perhaps even more

importantly, testing ensures that your code continues to work in the future as you add new features, refactor the existing ones, or upgrade major dependencies of your project.

There is more value in testing than you might realize. One of the best ways to fix a bug in your code is to write a failing test that exposes it. Then when you fix the bug and re-run the test, if it passes it means the bug is fixed, never reintroduced into the code base.

Tests can also serve as documentation for new people joining your team. For people who have never seen a codebase before, reading tests can help them understand how the existing code works.

Last but not least, more automated testing means less time spent with manual QA, freeing up valuable time.

Static Analysis

The first step to improve your code quality is to start using static analysis tools. Static analysis checks your code for errors as you write it, but without running any of that code.

- **Linters** analyze code to catch common errors such as unused code and to help avoid pitfalls, to flag style guide no-nos like using tabs instead of spaces (or vice versa, depending on your configuration).
- **Type checking** ensures that the construct you're passing to a function matches what the function was designed to accept, preventing passing a string to a counting function that expects a number, for instance.

React Native comes with two such tools configured out of the box: ESLint for linting and TypeScript for type checking.

Writing Testable Code

To start with tests, you first need to write code that is testable. Consider an aircraft manufacturing process - before any model first takes off to show that all of its complex systems work well together, individual parts are tested to guarantee they are safe and function correctly. For example, wings are tested by bending them under extreme load;

engine parts are tested for their durability; the windshield is tested against simulated bird impact.

Software is similar. Instead of writing your entire program in one huge file with many lines of code, you write your code in multiple small modules that you can test more thoroughly than if you tested the assembled whole. In this way, writing testable code is intertwined with writing clean, modular code.

To make your app more testable, start by separating the view part of your app—your React components—from your business logic and app state (regardless of whether you use Redux, MobX or other solutions). This way, you can keep your business logic testing—which shouldn't rely on your React components—independent of the components themselves, whose job is primarily rendering your app's UI!

Theoretically, you could go so far as to move all logic and data fetching out of your components. This way your components would be solely dedicated to rendering. Your state would be entirely independent of your components. Your app's logic would work without any React components at all!

> We encourage you to further explore the topic of testable code in other learning resources.

Writing Tests

After writing testable code, it's time to write some actual tests! The default template of React Native ships with Jest testing framework. It includes a preset that's tailored to this environment so you can get productive without tweaking the configuration and mocks straight away—more on mocks shortly. You can use Jest to write all types of tests featured in this guide.

> If you do test-driven development, you actually write tests first! That way, testability of your code is given.

Structuring Tests

Your tests should be short and ideally test only one thing. Let's start with an example unit test written with Jest:

```
it('given a date in the past, colorForDueDate() returns red', () => {
  expect(colorForDueDate('2000-10-20')).toBe('red');
});
```

The test is described by the string passed to the `it` function. Take good care writing the description so that it's clear what is being tested. Do your best to cover the following:

1. **Given** - some precondition
2. **When** - some action executed by the function that you're testing
3. **Then** - the expected outcome

This is also known as AAA (Arrange, Act, Assert).

Jest offers `describe` function to help structure your tests. Use `describe` to group together all tests that belong to one functionality. Describes can be nested, if you need that. Other functions you'll commonly use are `beforeEach` or `beforeAll` that you can use for setting up the objects you're testing. Read more in the Jest api reference.

If your test has many steps or many expectations, you probably want to split it into multiple smaller ones. Also, ensure that your tests are completely independent of one another. Each test in your suite must be executable on its own without first running some other test. Conversely, if you run all your tests together, the first test must not influence the output of the second one.

Lastly, as developers we like when our code works great and doesn't crash. With tests, this is often the opposite. Think of a failed test as of a *good thing!* When a test fails, it often means something is not right. This gives you an opportunity to fix the problem before it impacts the users.

Unit Tests

Unit tests cover the smallest parts of code, like individual functions or classes.

When the object being tested has any dependencies, you'll often need to mock them out, as described in the next paragraph.

The great thing about unit tests is that they are quick to write and run. Therefore, as you work, you get fast feedback about whether your tests are passing. Jest even has an option to continuously run tests that are related to code you're editing: Watch mode.

Mocking

Sometimes, when your tested objects have external dependencies, you'll want to "mock them out." "Mocking" is when you replace some dependency of your code with your own implementation.

> Generally, using real objects in your tests is better than using mocks but there are situations where this is not possible. For example: when your JS unit test relies on a native module written in Java or Objective-C.

Imagine you're writing an app that shows the current weather in your city and you're using some external service or other dependency that provides you with the weather information. If the service tells you that it's raining, you want to show an image with a rainy cloud. You don't want to call that service in your tests, because:

- It could make the tests slow and unstable (because of the network requests involved)
- The service may return different data every time you run the test
- Third party services can go offline when you really need to run tests!

Therefore, you can provide a mock implementation of the service, effectively replacing thousands of lines of code and some internet-connected thermometers!

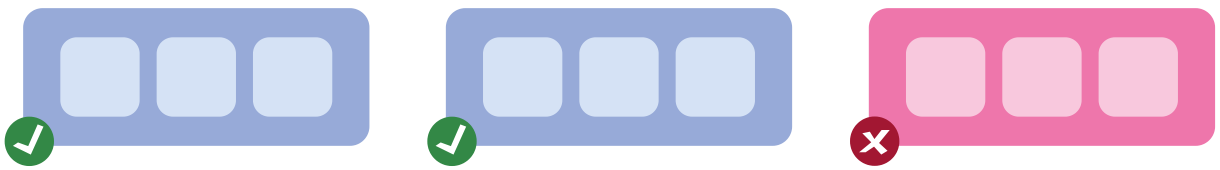
> Jest comes with support for mocking from function level all the way to module level mocking.

Integration Tests

When writing larger software systems, individual pieces of it need to interact with each other. In unit testing, if your unit depends on another one, you'll sometimes end up mocking the dependency, replacing it with a fake one.

In integration testing, real individual units are combined (same as in your app) and tested together to ensure that their cooperation works as expected. This is not to say that mocking does not happen here: you'll still need mocks (for example, to mock communication with a weather service), but you'll need them much less than in unit testing.

> Please note that the terminology around what integration testing means is not always consistent. Also, the line between what is a unit test and what is an integration test may not always be clear. For this guide, your test falls into "integration testing" if it:
>
> - Combines several modules of your app as described above
> - Uses an external system
> - Makes a network call to other application (such as the weather service API)
> - Does any kind of file or database I/O



Component Tests

React components are responsible for rendering your app, and users will directly interact with their output. Even if your app's business logic has high testing coverage and is correct, without component tests you may still deliver a broken UI to your users. Component tests could fall into both unit and integration testing, but because they are such a core part of React Native, we'll cover them separately.

For testing React components, there are two things you may want to test:

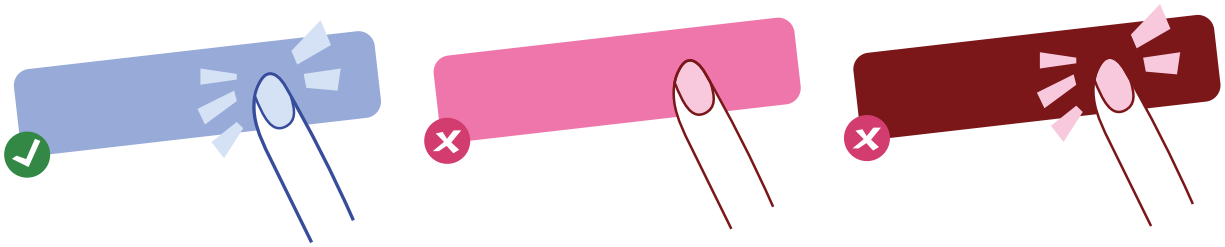
- Interaction: to ensure the component behaves correctly when interacted with by a user (eg. when user presses a button)
- Rendering: to ensure the component render output used by React is correct (eg. the button's appearance and placement in the UI)

For example, if you have a button that has an `onPress` listener, you want to test that the button both appears correctly and that tapping the button is correctly handled by the component.

There are several libraries that can help you testing these:

- React's Test Renderer, developed alongside its core, provides a React renderer that can be used to render React components to pure JavaScript objects, without depending on the DOM or a native mobile environment.
- React Native Testing Library builds on top of React's test renderer and adds `fireEvent` and `query` APIs described in the next paragraph.

> Component tests are only JavaScript tests running in Node.js environment. They do *not* take into account any iOS, Android, or other platform code which is backing the React Native components. It follows that they cannot give you a 100% confidence that everything works for the user. If there is a bug in the iOS or Android code, they will not find it.



Testing User Interactions

Aside from rendering some UI, your components handle events like `onChangeText` for `TextInput` or `onPress` for `Button`. They may also contain other functions and event callbacks. Consider the following example:

```
function GroceryShoppingList() {
  const [groceryItem, setGroceryItem] = useState('');
  const [items, setItems] = useState<string[]>([]);

  const addNewItemToShoppingList = useCallback(() => {
    setItems([groceryItem, ...items]);
    setGroceryItem('');
  }, [groceryItem, items]);

  return (
    <>
      <TextInput
        value={groceryItem}
        placeholder="Enter grocery item"
        onChangeText={text => setGroceryItem(text)}
      />
      <Button
        title="Add the item to list"
        onPress={addNewItemToShoppingList}
      />
      {items.map(item => (
        <Text key={item}>{item}</Text>
      ))}
    </>
  );
}
```

When testing user interactions, test the component from the user perspective—what's on the page? What changes when interacted with?

As a rule of thumb, prefer using things users can see or hear:

- make assertions using rendered text or accessibility helpers

Conversely, you should avoid:

- making assertions on component props or state
- testID queries

Avoid testing implementation details like props or state—while such tests work, they are not oriented toward how users will interact with the component and tend to break by refactoring (for example when you'd like to rename some things or rewrite class component using hooks).

> React class components are especially prone to testing their implementation details such as internal state, props or event handlers. To avoid testing implementation details, prefer using function components with Hooks, which make relying on component internals *harder*.

Component testing libraries such as React Native Testing Library facilitate writing user-centric tests by careful choice of provided APIs. The following example uses `fireEvent` methods `changeText` and `press` that simulate a user interacting with the component and a query function `getAllByText` that finds matching `Text` nodes in the rendered output.

```
test('given empty GroceryShoppingList, user can add an item to it', () => {
  const {getByPlaceholderText, getByText, getAllByText} = render(
    <GroceryShoppingList />,
  );

  fireEvent.changeText(
    getByPlaceholderText('Enter grocery item'),
    'banana',
  );
  fireEvent.press(getByText('Add the item to list'));
```

```
  const bananaElements = getAllByText('banana');
  expect(bananaElements).toHaveLength(1); // expect 'banana' to be on the list
});
```

This example is not testing how some state changes when you call a function. It tests what happens when a user changes text in the `TextInput` and presses the `Button`!

Testing Rendered Output

Snapshot testing is an advanced kind of testing enabled by Jest. It is a very powerful and low-level tool, so extra attention is advised when using it.

A "component snapshot" is a JSX-like string created by a custom React serializer built into Jest. This serializer lets Jest translate React component trees to string that's human-readable. Put another way: a component snapshot is a textual representation of your component's render output *generated* during a test run. It may look like this:

```
<Text
  style={
    Object {
      "fontSize": 20,
      "textAlign": "center",
    }
  }>
  Welcome to React Native!
</Text>
```

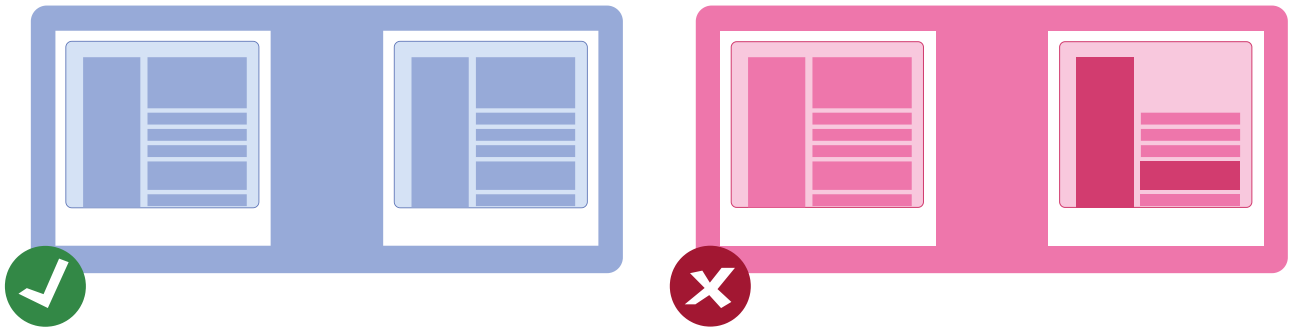
With snapshot testing, you typically first implement your component and then run the snapshot test. The snapshot test then creates a snapshot and saves it to a file in your repo as a reference snapshot. **The file is then committed and checked during code review**. Any future changes to the component render output will change its snapshot, which will cause the test to fail. You then need to update the stored reference snapshot for the test to pass. That change again needs to be committed and reviewed.

Snapshots have several weak points:

- For you as a developer or reviewer, it can be hard to tell whether a change in snapshot is intended or whether it's evidence of a bug. Especially large snapshots can quickly become hard to understand and their added value becomes low.
- When snapshot is created, at that point it is considered to be correct-even in the case when the rendered output is actually wrong.
- When a snapshot fails, it's tempting to update it using the `--updateSnapshot` jest option without taking proper care to investigate whether the change is expected. Certain developer discipline is thus needed.

Snapshots themselves do not ensure that your component render logic is correct, they are merely good at guarding against unexpected changes and for checking that the components in the React tree under test receive the expected props (styles and etc.).

We recommend that you only use small snapshots (see `no-large-snapshots` rule). If you want to test a *change* between two React component states, use `snapshot-diff`. When in doubt, prefer explicit expectations as described in the previous paragraph.



End-to-End Tests

In end-to-end (E2E) tests, you verify your app is working as expected on a device (or a simulator / emulator) from the user perspective.

This is done by building your app in the release configuration and running the tests against it. In E2E tests, you no longer think about React components, React Native APIs, Redux stores or any business logic. That is not the purpose of E2E tests and those are not even accessible to you during E2E testing.

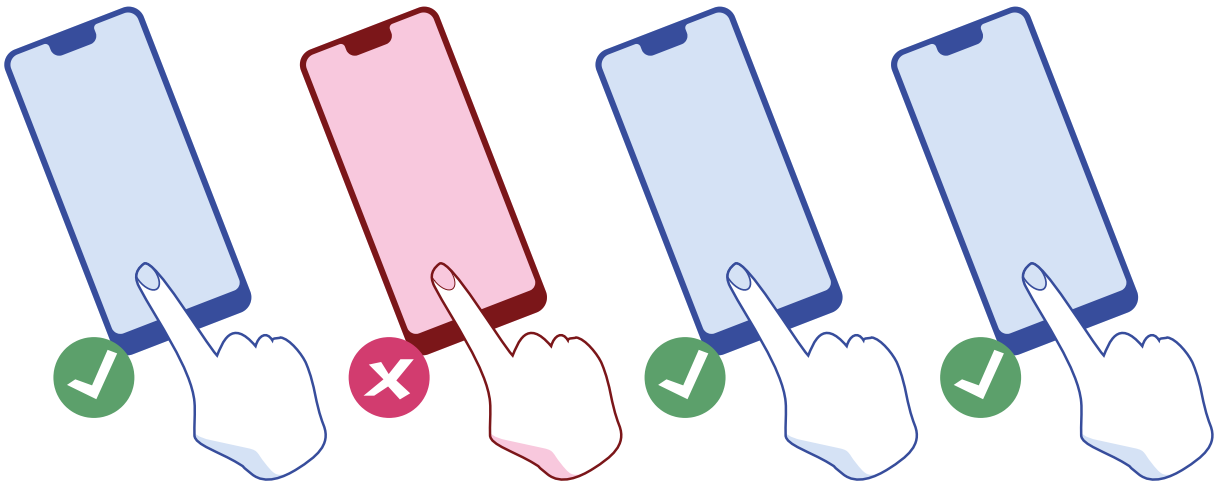
Instead, E2E testing libraries allow you to find and control elements in the screen of your app: for example, you can *actually* tap buttons or insert text into `TextInputs` the same way a real user would. Then you can make assertions about whether or not a certain element exists in the app's screen, whether or not it's visible, what text it contains, and so on.

E2E tests give you the highest possible confidence that part of your app is working. The tradeoffs include:

- writing them is more time consuming compared to the other types of tests
- they are slower to run
- they are more prone to flakiness (a "flaky" test is a test which randomly passes and fails without any change to code)

Try to cover the vital parts of your app with E2E tests: authentication flow, core functionalities, payments, etc. Use faster JS tests for the non-vital parts of your app. The more tests you add, the higher your confidence, but also, the more time you'll spend maintaining and running them. Consider the tradeoffs and decide what's best for you.

There are several E2E testing tools available: in the React Native community, Detox is a popular framework because it's tailored for React Native apps. Another popular library in the space of iOS and Android apps is Appium or Maestro.



Summary

We hope you enjoyed reading and learned something from this guide. There are many ways you can test your apps. It may be hard to decide what to use at first. However, we believe it all will make sense once you start adding tests to your awesome React Native app. So what are you waiting for? Get your coverage up!

Links

- React testing overview
- React Native Testing Library
- Jest docs
- Detox
- Appium
- Maestro

This guide originally authored and contributed in full by Vojtech Novak.

Is this page useful? 

Edit this page

*Last updated on **Jun 21, 2023***