Running Tests

This section is about testing your changes to React Native as a contributor. If you haven't yet, go through the steps to set up your development environment for building projects with native code.

JavaScript Tests

The simplest way to run the JavaScript test suite is by using the following command at the root of your React Native checkout:

```
npm test
```

This will run tests using Jest.

You should also make sure your code passes Flow and lint tests:

```
npm run flow
npm run lint
```

iOS Tests

Start off by running `pod install` inside the `RNTester/` directory. This will set up your native dependencies and create a `RNTesterPods` Xcode workspace.

Then, go back to the root of your React Native checkout and run `yarn` followed by `yarn start`. This will set up your JavaScript dependencies.

At this point, you can run iOS tests by invoking the following script from the root of your React Native checkout:

```
./scripts/objc-test.sh test
```

You can also use Xcode to run iOS tests. Open `RNTester/RNTesterPods.xcworkspace` and run tests locally by pressing Command + U or selecting `Product` then `Test` from the menubar.

Xcode also allows running individual tests through its Test Navigator (Command + 6).

> *Note: `objc-test.sh` ensures your test environment is set up to run all tests. It also disables tests that are known to be flaky or broken. Keep this in mind when running tests using Xcode. If you see an unexpected failure, see if it's disabled in `objc-test.sh` first.*

Android Tests

The Android unit tests do not run in an emulator. They just use a normal Java installation. The test suite is built using the Buck build tool.

To run the Android unit tests, invoke the following script from the root of your React Native checkout:

```
./scripts/run-android-local-unit-tests.sh
```

The Android integration tests, on the other hand, need additional setup. We recommend going through the instructions to set up your environment for building React Native from source.

Once you've done that, you can start the Android emulator using:

```
./scripts/run-android-emulator.sh
```

Then, run the Android integration tests:

```
./scripts/run-android-local-integration-tests.sh
```

End-to-end Tests

Finally, make sure end-to-end tests run successfully by executing the following script:

```
./scripts/test-manual-e2e.sh
```

End-to-end tests written in Detox confirm that React Native components and APIs function correctly in the context of a running app. They run the RNTester app in the simulator and simulate a user interacting with the app.

You can run Detox end-to-end tests locally by installing the Detox CLI on macOS, then running the following in the command line:

```
npm run build-ios-e2e
npm run test-ios-e2e
```

If you work on a component or API that isn't convered by a Detox test, please consider adding one. Detox tests are stored under `RNTester/e2e/__tests__`.

Writing Tests

Whenever you are fixing a bug or adding new functionality to React Native, it is a good idea to add a test that covers it. Depending on the change you're making, there are different types of tests that may be appropriate.

JavaScript Tests

The JavaScript tests can be found inside `__test__` directories, colocated next to the files that are being tested. See `TouchableHighlight-test.js` for a basic example. You can also follow Jest's Testing React Native Apps tutorial to learn more.

iOS Integration Tests

React Native provides facilities to make it easier to test integrated components that require both native and JS components to communicate across the bridge. The two main components are `RCTTestRunner` and `RCTTestModule`. `RCTTestRunner` sets up the React Native environment and provides facilities to run the tests as `XCTestCase`s in Xcode (`runTest:module` is the simplest method). `RCTTestModule` is exported to JavaScript as `NativeModules.TestModule`.

The tests themselves are written in JS, and must call `TestModule.markTestCompleted()` when they are done, otherwise the test will timeout and fail. Test failures are primarily indicated by throwing a JS exception. It is also

possible to test error conditions with `runTest:module:initialProps:expectErrorRegex:` or `runTest:module:initialProps:expectErrorBlock:` which will expect an error to be thrown and verify the error matches the provided criteria.

See the following for example usage and integration points:

- `IntegrationTestHarnessTest.js`
- `RNTesterIntegrationTests.m`
- `IntegrationTestsApp.js`

iOS Snapshot Tests

A common type of integration test is the snapshot test. These tests render a component, and verify snapshots of the screen against reference images using `TestModule.verifySnapshot()`, using the `FBSnapshotTestCase` library behind the scenes. Reference images are recorded by setting `recordMode = YES` on the `RCTTestRunner`, then running the tests.

Snapshots will differ slightly between 32 and 64 bit, and various OS versions, so it's recommended that you enforce tests are run with the correct configuration. It's also highly recommended that all network data be mocked out, along with other potentially troublesome dependencies. See `SimpleSnapshotTest` for a basic example.

If you make a change that affects a snapshot test in a pull request, such as adding a new example case to one of the examples that is snapshotted, you'll need to re-record the snapshot reference image. To do this, simply change to `_runner.recordMode = YES;` in RNTester/RNTesterSnapshotTests.m, re-run the failing tests, then flip record back to `NO` and submit/update your pull request and wait to see if the Circle build passes.

Android Unit Tests

It's a good idea to add an Android unit test whenever you are working on code that can be tested by Java code alone. The Android unit tests are located in `ReactAndroid/src/tests`. We recommend browsing through these to get an idea of what a good unit test might look like.

Android Integration Tests

It's a good idea to add an Android integration test whenever you are working on code that needs both JavaScript and Java to be tested in conjunction. The Android integration tests can be found in `ReactAndroid/src/androidTest`. We recommend browsing through these to get an idea of what a good integration test might look like.

Continuous Testing

We use Appveyor and Circle CI to automatically run our open source tests. Appveyor and Circle CI will run these tests whenever a commit is added to a pull request, as a way to help maintainers understand whether a code change introduces a regression. The tests also run on commits to the master and `*-stable` branches in order to keep track of the health of these branches.

There's another set of tests that run within Facebook's internal test infrastructure. Some of these tests are integration tests defined by internal consumers of React Native (e.g. unit tests for a React Native surface in the Facebook app). These tests run on every commit to the copy of React Native hosted on Facebook's source control. They also run when a pull request is imported to Facebook's source control.

If one of these tests fail, you'll need someone at Facebook to take a look. Since pull requests can only be imported by Facebook employees, whoever imported the pull request should be able to facilitate any details.

> ***Running CI tests locally***
>
> *Most open source collaborators rely on Circle CI and Appveyor to see the results of these tests. If you'd rather verify your changes locally using the same configuration as Circle CI, Circle CI provides a command line interface with the ability to run jobs locally.*

F.A.Q.

How do I upgrade the Xcode version used in CI tests?

When upgrading to a new version of Xcode, first make sure it is supported by Circle CI. You will also need to update the test environment config to make sure tests run on an iOS Simulator that comes installed in the Circle CI machine. This can also be found in Circle CI's Xcode version reference by clicking the desired version and looking under Runtimes.

You can then edit these two files:

- `.circleci/config.yml` : Edit the `xcode:` line under 'macos: `(search for` _XCODE_VERSION`)
- `scripts/.tests.env` : Edit the `IOS_TARGET_OS` envvar to match the desired iOS Runtime.

If you intend to merge this change on GitHub, please make sure to notify a Facebook employee as they'll need to update the value of `_XCODE_VERSION` used in the internal Sandcastle RN OSS iOS test in `react_native_oss.py` when they import your pull request.