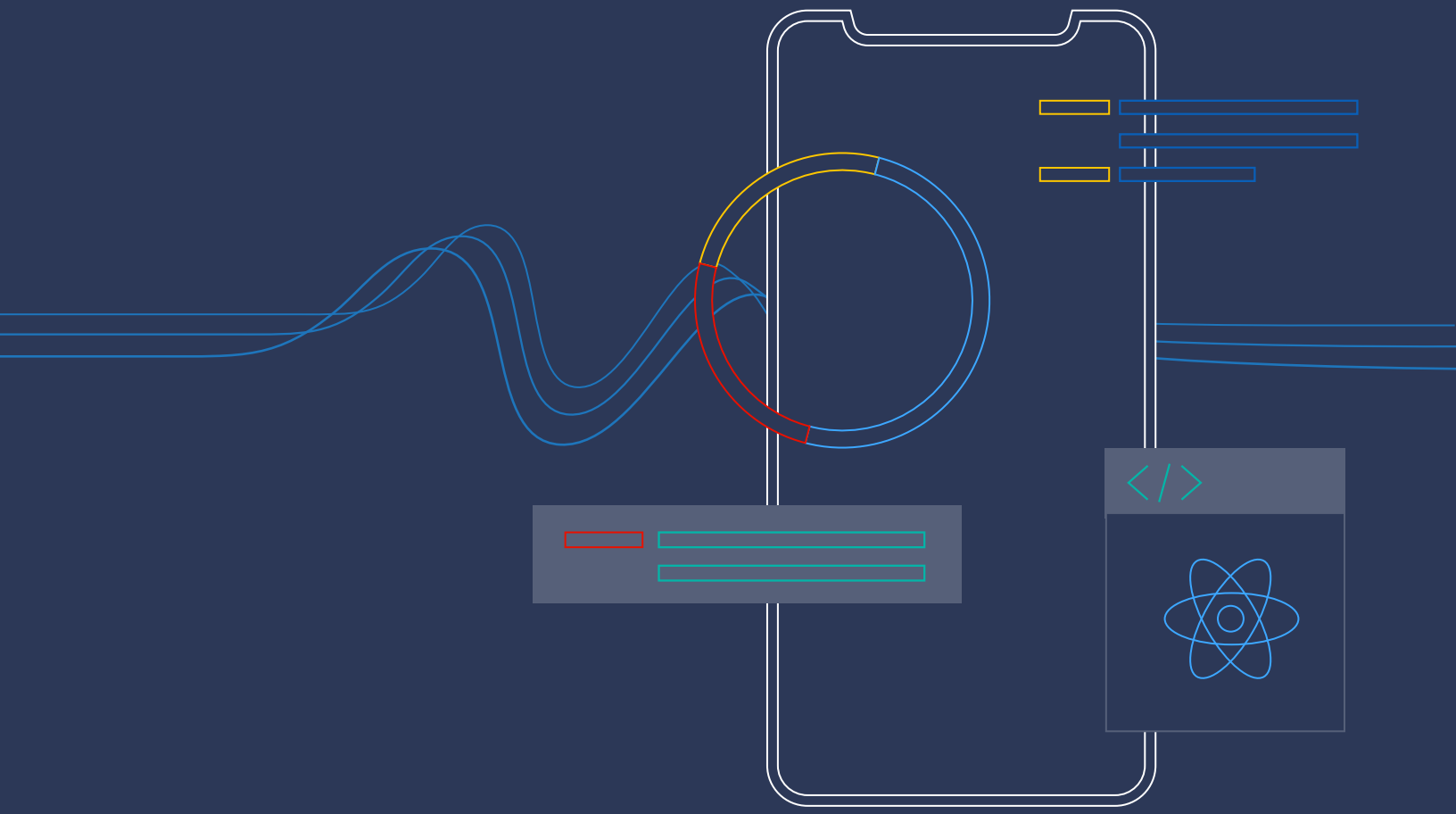
G2i

So You Want To Build A Maintainable React Native App

A step-by-step handbook for
React Native teams



About the Author

Adrian Carolli is an experienced React Native expert with a focus on helping companies build high-quality React Native apps at G2i. For the past 4 years, he has helped startups from Y Combinator, Google Ventures and Stanford University ship high-quality React Native apps. He loves to help companies find the right solution, not just the fastest solution for both technology and process.

• • • •

Follow Adrian Carolli on Twitter for more React Native advice. He works as a React Native developer at G2i, building maintainable apps for our clients.

Overview

Introduction 4

Inconsistent Code Is Hard To Understand 6

Automate Your Testing Efforts To Avoid Software Regression 9

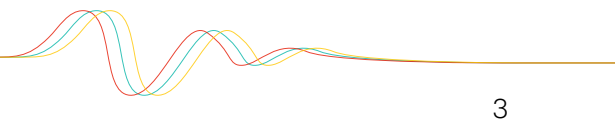
Automate Your Workflow To Avoid Repetitive Work 12

Type Check Everything 14

Keep Your Libraries Updated 17

Test Your Components 19

Conclusion 21



Introduction

Let's say you're building a React Native application for a company or organization and it's about to go into production. It is often overlooked but maintainability is the single most important factor in any software product. It's something you should be thinking about as early as possible. Therefore in this handbook, I will discuss how to make sure you build a maintainable React Native app.

Why should you care about maintainability?

The simple answer is that our goal as engineers is to build always working software. In fact, I once had an engineering manager who said that to me and it completely changed how I thought about software.

A lot of us forget to think about the always working part of software and instead focus on the building part of software— but that's bad for the future of any company. Building software that doesn't work is bad for hiring. It's bad because good engineers don't want to work for a company where the codebase is a big hairy mess. It's bad for users because users will become frustrated with broken software. And ultimately it's bad for the company because users will churn and good engineers will turnover. In fact, some companies have died because of unmaintainable code (more on that later).

So, why is always working software often overlooked? It's because people don't speak in terms of maintainability. They speak in terms of building features but as engineering managers it's our responsibility and our duty to think and act differently. We need to make maintainability a priority so engineers can write working and clean code in the future as well as the present. Such that we don't have to throw out the entire codebase and start over. And so that users don't find bugs that we could have caught and fixed. Most importantly by making the codebase easier to maintain we build high-quality working software.

> *Our goal as engineers is to build always working software.*

What's the problem with unmaintained software?

Now that we covered why we should care about maintainability, let's talk about the problem with unmaintained software. There is an old saying that says:

> *The single worst strategic mistake that any software company can make is to rewrite the code from scratch.*

Joel Spolsky talked about this mistake in his essay Things You Should Never Do. Joel spoke about Netscape taking three years to rewrite their codebase from scratch. As a result, Netscape lost their market leadership to Microsoft. Eventually, Microsoft's Internet Explorer became king.

It took Netscape nearly three years to ship a new version of their software! In addition, during those three years, Netscape barely supported their old software version — a cardinal sin. And that is the problem with writing unmaintainable software. It can kill a company.

> *Unmaintained software can kill any company.*

In fact, I went through the same problem when I was working on a React Native app for a new client at G2i. I joined the project before G2i got involved and it was in development for over a year. Like most developers who join a new project I had the urge to rewrite it, but I'm glad we didn't. Rewriting the app would have likely killed the company.

> *The reason that developers think old code is a mess is because of a cardinal, fundamental law of programming. It's harder to read code than to write it.*

Instead of rewriting the codebase, over a period of two months, the team focused on shipping the app and making the project more maintainable. As a result, we shipped the project in two months and we continue to make it more maintainable.

You can do the same thing. If you can improve maintainability by one percent each day, after one hundred days you will have a great codebase.

How can we make software maintainable?

So that begs the key question, how do you build a maintainable React Native app? Well, buckle up because I'm going to teach you.

Firstly, we must understand that unmaintainable code comes from the following key problems:

- Inconsistency
- A lack of automation
- Outdated code
- Readability
- Poor code structure
- Poor architecture
- A lack of documentation

In this book, I will mainly focus on improving consistency, readability, automation and keeping your code up to date. Let's get right to it.

Inconsistent Code Is Hard To Understand

Problem: Inconsistent code is harder to understand and leads to more bugs

Every engineer has a different coding style but ultimately this is bad for the overall consistency of the codebase. If every component is written differently, it's harder for a team of engineers to understand each component.

Imagine reading a book where each chapter was written by a different author. I bet it would be hard to read that book because the book would be really inconsistent — it's the same thing with writing code. The codebase needs to be consistent and cohesive because it will be easier to read. Joel Spolsky explained this concept:

> *The reason that developers think old code is a mess is because of a cardinal, fundamental law of programming. It's harder to read code than to write it.*

Reading code that you didn't write can give off the feeling that the codebase is unmaintainable. This leads engineers down the wrong path which is to rewrite the codebase. However, the contrarian thing to do is to make the codebase more consistent. Let's do that.

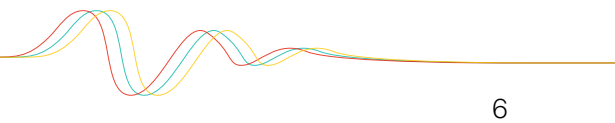
Solution: Use ESLint, Prettier, and Husky

The easiest and the best thing you can do to make code style consistent is to setup ESLint and Prettier.

ESLint is a static code analysis tool for identifying problematic patterns found in JavaScript code. ESLint covers both code quality and coding style issues. Whereas Prettier is an opinionated code formatter. It enforces a consistent style by parsing your code and re-printing it with its own rules.

In addition, we are going to setup something a bit more novel. We are going to install ESLint, Prettier and Husky. Husky improves your commits to make sure the entire team can't commit sh*t to the codebase. You can use it to lint your commit messages, run tests, lint code, etc... when someone commits or pushes code. We are simply going to use Husky to run ESLint and Prettier.

I've considered using Husky for other things like running unit tests but that can take a long time to run. Consequently, we don't want our tools to degrade the developer experience because then developers won't use them. If every time we commit we have to wait for tests to run, that can slow us down. Therefore, we will only run ESLint and Prettier with Husky.



We will also use a library called `lint-staged`. The neat thing about using lint-staged is that it only runs on the code you are about to commit. This way we can avoid lint-ing the entire codebase, which would be really slow to lint and would format the entire codebase.

Setup ESLint, Husky and Prettier

Firstly, install ESLint and Prettier (run this in the working directory of your source code):

```
npm install prettier eslint --save-dev
```

To install Husky follow the instructions found in the Husky repository like so:

```
npm install husky --save-dev
```

And to install lint-staged follow the instructions found in the lint-staged repository:

```
npx mrm lint-staged
```

Once you have that setup, inside your `package.json` you should have the following:

```
“husky”: {
   “hooks”: {
      “pre-commit”: “lint-staged”
    }
},
“lint-staged”: {
   “*.{js,jsx,ts,tsx}”: [
      “eslint --cache --fix”,
      “prettier --write”
   ]
}
```

Now, every time an engineer commits code it will run ESLint and Prettier. Always keeping the codebase consistent. Awesome!

You can test it like so:

```
git add .
git commit -m “Adds ESLint, Prettier, Husky and lint-staged”
```

The output will look something like this after you run `git commit.`

```
husky > pre-commit (node v10.20.0)
✔ Preparing...
✔ Running tasks...
✔ Applying modifications...
✔ Cleaning up...
[adrian/test-1 bee83072] Adds ESLint, Prettier, Husky and lint-staged
 1 file changed, 1 insertion(+)
```

Automate Your Testing Efforts To Avoid Software Regression

Problem: Without automated testing code, changes lead to software regression

If you ever played a game of whack-a-mole you know what software regression feels like. You make a code change (whack) and you break something unexpectedly (mole). It's a constant and never-ending battle between you and the codebase. No software engineer should go through this game of whack a mole.

Solution: Use Detox to automate your testing efforts

It's hard to continuously release software if it's always breaking. So, how do you combat this? Well, it's as simple as introducing quality assurance (QA) automation to your React Native app.

A good form of quality assurance automation for React Native is Detox. Detox is a gray box end-to-end testing and automation library for mobile apps. Unlike black box testing which involves no knowledge of the system. And white box testing, which tests the internals of a system. Gray box testing has limited knowledge of the internals of the system. Gray box testing uses the internals of the system to wait for network requests to finish, for example. However, gray box testing does not have any more knowledge of the system.

Detox is a gray box end-to-end testing library

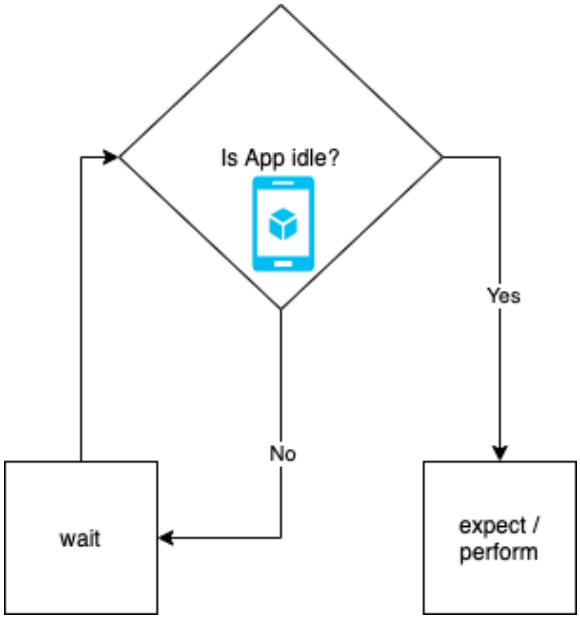
Advantages and Disadvantages

The disadvantage of white box testing such as unit testing is that exhaustive testing is expensive and complex. You can quickly get a lot more coverage with gray box testing than you can with unit testing. Nonetheless, unit testing is important, but a few simple Detox tests will test your app much more thoroughly than hundreds of unit tests could.

Furthermore, the disadvantage of black box testing (for example Appium) is that the tests tend to be flaky and unreliable. With gray box testing, you don't experience flakiness because with Detox you can wait for network requests to finish before testing further. You don't need a bunch of arbitrary `Thread.sleep` everywhere.

How does Detox work?

Unlike most QA automation frameworks Detox is different. It synchronizes with your app's activity. For each line of test code Detox waits for your app to be completely idle before proceeding to the next line of code. Here is what that looks like in a diagram:



Detox wait and then expect / perform

The operations that Detox waits for include:

- Network requests
- Main thread (native code)
- UI changes
- Timers
- Animations
- React Native JavaScript thread
- React Native bridge

After all these operations are complete Detox will run the next line of code. This makes for a pretty robust and reliable test suit in contrast to other QA automation frameworks.

Setup and Testing

To setup Detox you must follow the getting started instructions. I suggest you setup the testing for a single platform (for example iOS), especially if you are strapped for time. It's a lot of work to maintain two platforms as opposed to one.

Once you have setup Detox, you can write your first test. Create a folder called `e2e` at the root of your project. This `e2e` folder stands for end-to-end (tests).

For example, this is a test for a login screen, it runs on a device/simulator like an actual user:

```
describe('Login flow', () => {
  it('should login successfully', async () => {
    await device.reloadReactNative();

    await element(by.id('email')).typeText('john@example.com');
    await element(by.id('password')).typeText('123456');
    await element(by.text('Login')).tap();

    await expect(element(by.text('Welcome'))).toBeVisible();
    await expect(element(by.id('email'))).toNotExist();
  });

});
```

With just ten lines of code this test will test the login flow and prevent software regression. Awesome!

Automate Your Workflow To Avoid Repetitive Work

Problem: It's time-consuming to run tests and release manually

One of the biggest challenges of mobile app development is building and releasing your app. It can be a painful process without automation. Therefore it is important we automate the build and release workflow to avoid these time consuming issues.

Furthermore, automation will make it easier and faster for teams to continuously deploy code every day. The amount of time you will save by setting up automation is well-worth the setup time.

Solution: Automate your workflow with Bitrise CI / CD

Some people like Fastlane or App Center for automating mobile workflows, but I recommend using Bitrise. Bitrise is a Continuous Integration and Delivery (CI/CD) Platform as a Service (PaaS) with a main focus on mobile app development (iOS, Android, React Native, and so on). It is a collection of tools and services to help you with the development and automation of your software projects.

I mainly like Bitrise because it has over 300+ automated steps and integrations that make setup a breeze. It's definitely the fastest way I have found to deploy, test and build mobile apps.

Setup Bitrise

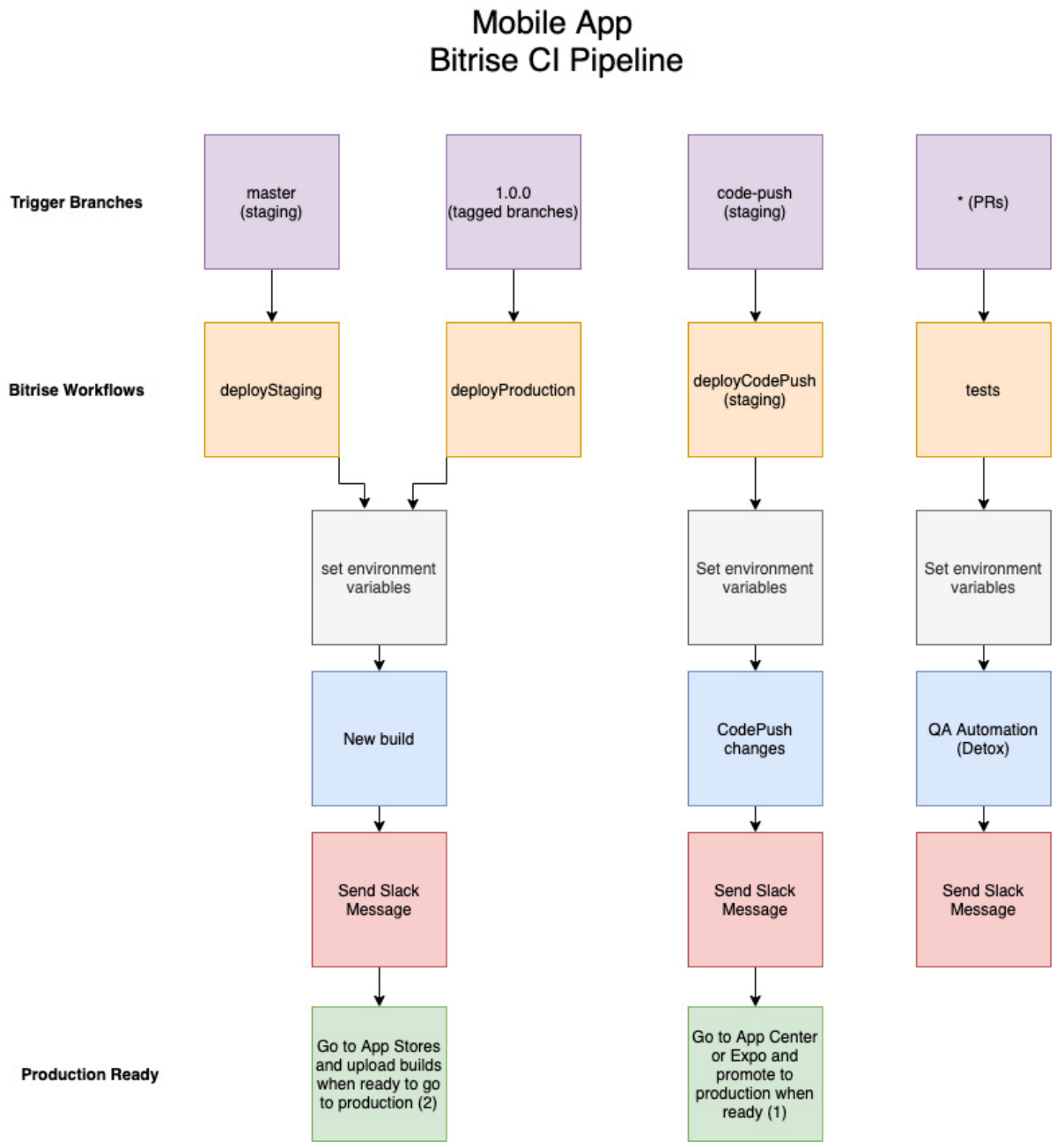
To set up Bitrise follow the getting started instructions. They have a free hobby tier that I recommend as a starting point before committing. However, if you're setting this up for a company or organization the "Org Standard" tier is ultimately the best for most companies. It allows you to run up to ten builds in parallel whereas the free tier only allows you to run one.

After the initial setup, you will want to create a few workflows. The workflows that you should setup are:

- **deployStaging —** This workflow will deploy a QA build to TestFlight (iOS) and App Tester (Android). In Github, when code changes on the `master` branch this workflow will run on a nightly schedule or immediately if you prefer it to be instant.
- **deployProduction —** This workflow will deploy a build that's ready to release on the iOS and Android stores. In Github, when code get's pushed to a tagged branch this workflow will run. For example, when you push a new git tag called 1.0.0

- **tests —** This workflow will run your Detox end to end tests. In Github, when a pull request is made this workflow will run.
- **deployCodePush —** This workflow will allow you to hot-fix changes to production quickly using CodePush or Expo over the air updates (OTA). We often run this workflow manually from Bitrise on specific tagged branches.

Here is what those workflows look like in a simple to understand diagram:



Type Check Everything

Problem: Without data types it's hard to understand data structures and data flow

React Native is written in JavaScript which is a loosely typed language, meaning you don't have to specify what type of information will be stored in a variable. The problem is that in the long-term the codebase is filled with variables in which developers don't understand what type of information they store.

Imagine if you asked your friend to go get your jacket from inside but you didn't specify what it looked like — that would be a loosely typed language. Now, if you tell your friend exactly what your jacket looks like that becomes a strongly typed language. Strongly typed languages allow developers to understand what the data looks like.

Let's see an example, here is a loosely typed snippet of code in JavaScript:

```
function add(x, y) {
    return x + y
}
```

In this code, it is unclear what the type of x and y is supposed to be. In fact, all of these function invocations could be valid:

```
add(1,2)
add("Adrian", "Caro")
add(undefined, undefined)
```

Yet the developer only expected numbers to be valid — not strings.

Solution: Use a strongly typed language like TypeScript to avoid type ambiguities

With a strongly typed language like TypeScript developers can inform future developers on what variable types are permitted for a given variable. Here is the same code but in TypeScript:

```
function add(x: number, y: number): number {
    return x + y
}
```

This new code informs any future developer that `x` and `y` should be numbers and the return value is also a number. Awesome!

Setup

To setup TypeScript for an existing project please follow these instructions*.*

1. Add TypeScript and the types for React Native and Jest to your project:

```
yarn add -D typescript @types/jest @types/react @types
/react-native @types/react-test-renderer
```

2. Add a TypeScript config file. Create a `tsconfig.json` in the root of your project:

```
{
  “compilerOptions”: {
    “allowJs”: true,
    “allowSyntheticDefaultImports”: true,
    “esModuleInterop”: true,
    “isolatedModules”: true,
    “jsx”: “react”,
    “lib”: [“es6”],
    “moduleResolution”: “node”,
        “noEmit”: true,
        “strict”: true,
        “target”: “esnext”
  },
  “exclude”: [
    “node_modules”,
    “babel.config.js”,
    “metro.config.js”,
    “jest.config.js”
  ]
}
```

3. Create a `jest.config.js` file to configure Jest to use TypeScript

```
module.exports = {
  preset: 'react-native',
  moduleFileExtensions: ['ts', 'tsx', 'js', 'jsx', 'json', 'node']
};
```

4. Rename a JavaScript file to be `*.tsx`

5. Run `yarn tsc` to type-check your new TypeScript files.

A few important points:

- The `any` type in TypeScript allows developers to assign anything to a variable — it's as ambiguous as using vanilla JavaScript. Try to avoid typing variables as any as it defeats the purpose of using a strongly typed language.
- It's a good idea to take a course on TypeScript — I recommend this course by Stephen Grider.
- Enabling strict mode is important because it enforces developers to specify the type of every variable in the codebase. If you miss typing a variable you leave the codebase vulnerable to unexpected types. It's like wearing armor that covers your entire body but not your eyes (see the picture below).

Strictly type your code or else...

Keep Your Libraries Updated

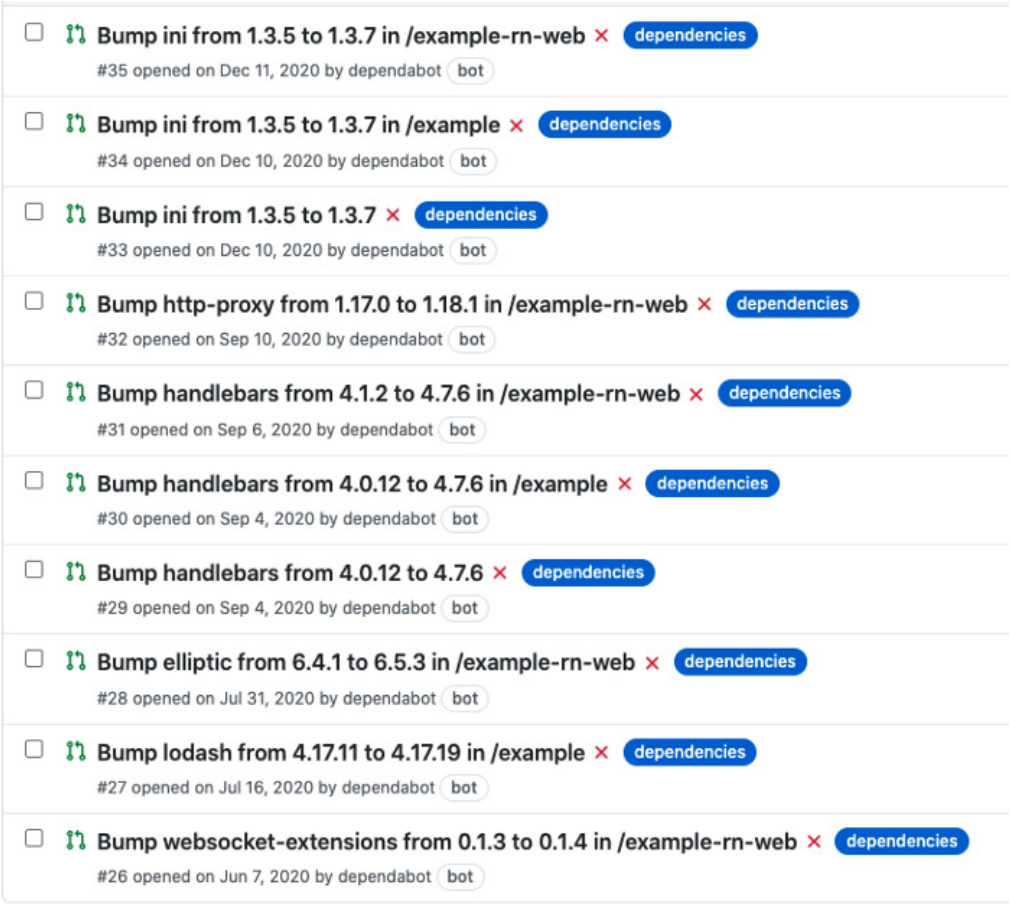
Problem: Outdated libraries or packages leave the codebase vulnerable to security issues and bugs

Using an outdated version of React Native or any library is much like running around with your head on fire. It is wildly foolish to use outdated libraries in production. It can lead to security attacks or bugs that could easily be avoided by constantly updating your libraries.

One option is that you could run `yarn outdated` to find all outdated packages. However, that's manual work and time-consuming.

Solution: Install dependabot to keep your libraries updated

Dependabot automatically creates pull requests to keep your dependencies secure and up-to-date. It integrates with Github and works seamlessly in the background constantly informing developers that packages are outdated. For example, a Dependabot pull request to upgrade a package looks like this:



Being on the latest version of a package usually (not always) means less bugs for your users. Dependabot acts as a constant reminder to update your packages. Awesome!

Setup

To setup Dependabot follow these instructions:

1. Create a dependabot.yml file in the root directory of the codebase
2. Configure the file accordingly (see below)
3. Commit the file to the repository

Here is an example of a dependabot.yml configuration.

```
# Setup for dependabot

version: 2
updates:
  # Maintain dependencies for npm
  - package-ecosystem: 'npm'
    # Look for `package.json` and `lock` files in the `root`
directory
    directory: '/'
    # Check the yarn registry for updates every month
    schedule:
      interval: 'monthly'
    # Allow up to 8 open pull requests for npm dependencies
    open-pull-requests-limit: 8
```

This will check that your npm packages are updated every month and submit up to eight pull requests. Awesome!

Test Your Components

Problem: Unit Testing is time-consuming and difficult to maintain

Before thinking about unit testing every function, know that it will likely increase development time. In fact, this might not be suitable for every company (such as a fast-paced startup).

For a bigger company and established product, unit testing makes a lot of sense. On the other hand if the product is in flux you may want to opt for another approach (more on that soon).

Since unit tests are so closely tied to the implementation details your tests will need to change as frequently as your implementation changes — this is a big price to pay. However, in the long term, it tends to pay off. Overall it is a tradeoff between having short term speed vs long term maintainability.

Truthfully, only you can decide if unit testing is the right decision. Nonetheless, here are some simple questions to ask yourself to help you with your decision:

- How fast is the product moving? If the product is frequently changing consider dialling down your unit testing until the product is more established. Opt for QA automation instead or component testing.
- What is the project timeline? If the product launch date is in six months to over a year consider adding unit tests along the way. Opt for less unit testing if the project due date is right around the corner.
- How big is your team? The bigger the team the more you need unit testing. Not every engineer will understand the code without proper unit testing. Furthermore, it may lead to future bugs if you don't have tests.

Ultimately, in the long term unit testing is important but in the short term unit testing can impede progress. So, what should you do?

Solution: Test your components using react-native-testing-library

Instead of unit testing functions in React Native there is a middle ground. It is called component testing. Instead you write tests for React Native components.

As a part of this goal, you want your tests to avoid including implementation details of your components. As part of this, you want your test-base to be maintainable in the long run so refactors of your components (changes to implementation but not functionality) don't break your tests which slow you and your team down.

The React Native Testing Library is a great solution for testing React Native components. It provides light utility functions on top of `react-test-renderer`, in a way that encourages better testing practices. Its primary guiding principle is:

> *The more your tests resemble the way your software is used, the more confidence they can give you.*

This project is inspired by React Testing Library. It is tested to work with Jest, but it should work with other test runners as well.

Setup

In order to setup React Native Testing Library follow these instructions. Assuming you have Jest installed it is as simple as running the following command in your terminal:

```
yarn add --dev @testing-library/react-native
```

Now, let's see an example of a React Native component test:

```
import React from 'react'
import Text from './Text'
import { render } from '@testing-library/react-native'

describe('Text tests', () => {
  it('should render with text foo', () => {
    const { getByText } = render(<Text>foo</Text>)
    const el = getByText('foo')
    expect(el).not.toBeNull()
  })
})
```

The above test is testing a `<Text>` component. It uses `getByText` to see if the react test renderer has rendered the words "foo". The test is completely unaware of the implementation details and only tests the functionality of the component—this is the power of component testing!

Conclusion

In conclusion, I hope this handbook helps you build a maintainable React Native app. Additionally, I hope that it helps you build a codebase that will last for decades because no company should rewrite their codebase (if possible).

Of course, not everyone has the time or resources to build a maintainable React Native app. That's where G2i comes in.

G2i has developed a keen sense of developer skill. We've evaluated thousands of React, React Native, and Node.js developers, carefully honing our interview skills and constantly improving our vetting process to ensure our clients are working solely with the best developers.

G2i pairs incredible companies with high-quality developers

G2i is the only marketplace specifically for JavaScript developers. We focus on finding and vetting developers that have deep domain knowledge and understand how to build a maintainable codebase. When you hire a React Native developer via G2i, they will have passed a test specifically designed to vet their intricate knowledge of the ins and outs of React Native. Due to their deep knowledge base, our developers can hit the ground running immediately implementing the best practices mentioned in this book.

Learn more about how G2i can pair you with the perfect developer for your project and budget.


G2i