**Testing React Native applications**

To start testing code of any kind, the first thing we need is a testing framework, which we can use to run a set of test cases and inspect their results. For testing a JavaScript application, [Jest](https://jestjs.io/) is a popular candidate for such testing framework. For testing an Expo based React Native application with Jest, Expo provides a set of Jest configuration in a form of [jest-expo](https://github.com/expo/expo/tree/master/packages/jest-expo) preset. In order to use ESLint in the Jest's test files, we also need the [eslint-plugin-jest](https://www.npmjs.com/package/eslint-plugin-jest) plugin for ESLint. Let's get started by installing the packages:

npm install --save-dev jest jest-expo eslint-plugin-jest

To use the jest-expo preset in Jest, we need to add the following Jest configuration to the *package.json* file along with the *test* script:

{

// ...

"scripts": {

// other scripts...

"test": "jest" },

"jest": { "preset": "jest-expo", "transform": { "^.+\\.jsx?$": "babel-jest" }, "transformIgnorePatterns": [ "node\_modules/(?!((jest-)?react-native|@react-native(-community)?)|expo(nent)?|@expo(nent)?/.\*|@expo-google-fonts/.\*|react-navigation|@react-navigation/.\*|@unimodules/.\*|unimodules|sentry-expo|native-base|react-native-svg|react-router-native)" ] }, // ...

}

The transform option tells Jest to transform *.js* and *.jsx* files with the [Babel](https://babeljs.io/) compiler. The transformIgnorePatterns option is for ignoring certain directories in the *node\_modules* directory while transforming files. This Jest configuration is almost identical to the one proposed in the Expo's [documentation](https://docs.expo.io/guides/testing-with-jest/).

To use the eslint-plugin-jest plugin in ESLint, we need to include it in the plugins and extensions array in the *.eslintrc* file:

{

"plugins": ["react", "react-native"],

"settings": {

"react": {

"version": "detect"

}

},

"extends": ["eslint:recommended", "plugin:react/recommended", "plugin:jest/recommended"], "parser": "@babel/eslint-parser",

"env": {

"react-native/react-native": true

},

"rules": {

"react/prop-types": "off",

"react/react-in-jsx-scope": "off"

}

}

To see that the setup is working, create a directory *\_\_tests\_\_* in the *src* directory and in the created directory create a file *example.js*. In that file, add this simple test:

describe('Example', () => {

it('works', () => {

expect(1).toBe(1);

});

});

Now, let's run our example test by running npm test. The command's output should indicate that the test located in the *src/\_\_tests\_\_/example.js* file is passed.

**Organizing tests**

Organizing test files in a single *\_\_tests\_\_* directory is one approach in organizing the tests. When choosing this approach, it is recommended to put the test files in their corresponding subdirectories just like the code itself. This means that for example tests related to components are in the *components* directory, tests related to utilities are in the *utils* directory, and so on. This will result in the following structure:

src/

\_\_tests\_\_/

components/

AppBar.js

RepositoryList.js

...

utils/

authStorage.js

...

...

Another approach is to organize the tests near the implementation. This means that for example, the test file containing tests for the AppBar component is in the same directory as the component's code. This will result in the following structure:

src/

components/

AppBar/

AppBar.test.jsx

index.jsx

...

...

In this example, the component's code is in the *index.jsx* file and the test in the *AppBar.test.jsx* file. Note that in order to Jest finding your test files you either have to put them into a *\_\_tests\_\_* directory, use the *.test* or *.spec* suffix, or [manually configure](https://jestjs.io/docs/en/configuration#testmatch-arraystring) the global patterns.

**Testing components**

Now that we have managed to set up Jest and run a very simple test, it is time to find out how to test components. As we know, testing components requires a way to serialize a component's render output and simulate firing different kind of events, such as pressing a button. For these purposes, there is the [Testing Library](https://testing-library.com/docs/intro) family, which provides libraries for testing user interface components in different platforms. All of these libraries share similar API for testing user interface components in a user-centric way.

In [part 5](https://fullstackopen.com/en/part5/testing_react_apps) we got familiar with one of these libraries, the [React Testing Library](https://testing-library.com/docs/react-testing-library/intro). Unfortunately, this library is only suitable for testing React web applications. Luckily, there exists a React Native counterpart for this library, which is the [React Native Testing Library](https://callstack.github.io/react-native-testing-library/). This is the library we will be using while testing our React Native application's components. The good news is, that these libraries share a very similar API, so there aren't too many new concepts to learn. In addition to the React Native Testing Library, we need a set of React Native specific Jest matchers such as toHaveTextContent and toHaveProp. These matchers are provided by the [jest-native](https://github.com/testing-library/jest-native) library. Before getting into the details, let's install these packages:

npm install --save-dev --legacy-peer-deps react-test-renderer@18.2.0 @testing-library/react-native @testing-library/jest-native

**NB:** If you face peer dependency issues, make sure that the react-test-renderer version matches the project's React version in the npm install command above. You can check the React version by running npm list react --depth=0.

If the installation fails due to peer dependency issues, try again using the --legacy-peer-deps flag with the npm install command.

To be able to use these matchers we need to extend the Jest's expect object. This can be done by using a global setup file. Create a file *setupTests.js* in the root directory of your project, that is, the same directory where the *package.json* file is located. In that file add the following line:

import '@testing-library/jest-native/extend-expect';

Next, configure this file as a setup file in the Jest's configuration in the *package.json* file (note that the <rootDir> in the path is intentional and there is no need to replace it):

{

// ...

"jest": {

"preset": "jest-expo",

"transform": {

"^.+\\.jsx?$": "babel-jest"

},

"transformIgnorePatterns": [

"node\_modules/(?!(jest-)?react-native|react-clone-referenced-element|@react-native-community|expo(nent)?|@expo(nent)?/.\*|react-navigation|@react-navigation/.\*|@unimodules/.\*|unimodules|sentry-expo|native-base|@sentry/.\*|react-router-native)"

],

"setupFilesAfterEnv": ["<rootDir>/setupTests.js"] }

// ...

}

The main concepts of the React Native Testing Library are the [queries](https://callstack.github.io/react-native-testing-library/docs/api-queries) and [firing events](https://callstack.github.io/react-native-testing-library/docs/api#fireevent). Queries are used to extract a set of nodes from the component that is rendered using the [render](https://callstack.github.io/react-native-testing-library/docs/api#render) function. Queries are useful in tests where we expect for example some text, such as the name of a repository, to be present in the rendered component. Here's an example how to use the [ByText](https://callstack.github.io/react-native-testing-library/docs/api-queries/#bytext) query to check if the component's Text element has the correct textual content:

import { Text, View } from 'react-native';

import { render } from '@testing-library/react-native';

const Greeting = ({ name }) => {

return (

<View>

<Text>Hello {name}!</Text>

</View>

);

};

describe('Greeting', () => {

it('renders a greeting message based on the name prop', () => {

const { debug, getByText } = render(<Greeting name="Kalle" />);

debug();

expect(getByText('Hello Kalle!')).toBeDefined();

});

});

React Native Testing Library's documentation has some good hints on [how to query different kinds of elements](https://callstack.github.io/react-native-testing-library/docs/how-should-i-query). Another guide worth reading is Kent C. Dodds article [Making your UI tests resilient to change](https://kentcdodds.com/blog/making-your-ui-tests-resilient-to-change).

The render function returns the queries and additional helpers, such as the debug function. The [debug](https://callstack.github.io/react-native-testing-library/docs/api#debug) function prints the rendered React tree in a user-friendly format. Use it if you are unsure what the React tree rendered by the render function looks like. We acquire the Text node containing certain text by using the getByText function. For all available queries, check the React Native Testing Library's [documentation](https://callstack.github.io/react-native-testing-library/docs/api-queries). The toHaveTextContent matcher is used to assert that the node's textual content is correct. The full list of available React Native specific matchers can be found in the [documentation](https://github.com/testing-library/jest-native#matchers) of the jest-native library. Jest's [documentation](https://jestjs.io/docs/en/expect) contains every universal Jest matcher.

The second very important React Native Testing Library concept is firing events. We can fire an event in a provided node by using the [fireEvent](https://callstack.github.io/react-native-testing-library/docs/api#fireevent) object's methods. This is useful for example typing text into a text field or pressing a button. Here is an example of how to test submitting a simple form:

import { useState } from 'react';

import { Text, TextInput, Pressable, View } from 'react-native';

import { render, fireEvent } from '@testing-library/react-native';

const Form = ({ onSubmit }) => {

const [username, setUsername] = useState('');

const [password, setPassword] = useState('');

const handleSubmit = () => {

onSubmit({ username, password });

};

return (

<View>

<View>

<TextInput

value={username}

onChangeText={(text) => setUsername(text)}

placeholder="Username"

/>

</View>

<View>

<TextInput

value={password}

onChangeText={(text) => setPassword(text)}

placeholder="Password"

/>

</View>

<View>

<Pressable onPress={handleSubmit}>

<Text>Submit</Text>

</Pressable>

</View>

</View>

);

};

describe('Form', () => {

it('calls function provided by onSubmit prop after pressing the submit button', () => {

const onSubmit = jest.fn();

const { getByPlaceholderText, getByText } = render(<Form onSubmit={onSubmit} />);

fireEvent.changeText(getByPlaceholderText('Username'), 'kalle');

fireEvent.changeText(getByPlaceholderText('Password'), 'password');

fireEvent.press(getByText('Submit'));

expect(onSubmit).toHaveBeenCalledTimes(1);

// onSubmit.mock.calls[0][0] contains the first argument of the first call

expect(onSubmit.mock.calls[0][0]).toEqual({

username: 'kalle',

password: 'password',

});

});

});

In this test, we want to test that after filling the form's fields using the fireEvent.changeText method and pressing the submit button using the fireEvent.press method, the onSubmit callback function is called correctly. To inspect whether the onSubmit function is called and with which arguments, we can use a [mock function](https://jestjs.io/docs/en/mock-function-api). Mock functions are functions with preprogrammed behavior such as a specific return value. In addition, we can create expectations for the mock functions such as "expect the mock function to have been called once". The full list of available expectations can be found in the Jest's [expect documentation](https://jestjs.io/docs/en/expect).

Before heading further into the world of testing React Native applications, play around with these examples by adding a test file in the *\_\_tests\_\_* directory we created earlier.

**Handling dependencies in tests**

Components in the previous examples are quite easy to test because they are more or less *pure*. Pure components don't depend on *side effects* such as network requests or using some native API such as the AsyncStorage. The Form component is much less pure than the Greeting component because its state changes can be counted as a side effect. Nevertheless, testing it isn't too difficult.

Next, let's have a look at a strategy for testing components with side effects. Let's pick the RepositoryList component from our application as an example. At the moment the component has one side effect, which is a GraphQL query for fetching the reviewed repositories. The current implementation of the RepositoryList component looks something like this:

const RepositoryList = () => {

const { repositories } = useRepositories();

const repositoryNodes = repositories

? repositories.edges.map((edge) => edge.node)

: [];

return (

<FlatList

data={repositoryNodes}

// ...

/>

);

};

export default RepositoryList;

The only side effect is the use of the useRepositories hook, which sends a GraphQL query. There are a few ways to test this component. One way is to mock the Apollo Client's responses as instructed in the Apollo Client's [documentation](https://www.apollographql.com/docs/react/development-testing/testing/). A more simple way is to assume that the useRepositories hook works as intended (preferably through testing it) and extract the components "pure" code into another component, such as the RepositoryListContainer component:

export const RepositoryListContainer = ({ repositories }) => {

const repositoryNodes = repositories

? repositories.edges.map((edge) => edge.node)

: [];

return (

<FlatList

data={repositoryNodes}

// ...

/>

);

};

const RepositoryList = () => {

const { repositories } = useRepositories();

return <RepositoryListContainer repositories={repositories} />;

};

export default RepositoryList;

Now, the RepositoryList component contains only the side effects and its implementation is quite simple. We can test the RepositoryListContainer component by providing it with paginated repository data through the repositories prop and checking that the rendered content has the correct information.