{() => fs}

Fullstack > Part 10 > Introduction to React Native

a Introduction to React Native

Note: This course material was updated in Feb 2024. Some updates are not compatible anymore with older material. We recommend a fresh start with this new Part 10 material. However, if you´re returning to this course after a break, and you want to continue the exercises in your older project, please use Part 10 material before the update.

Traditionally, developing native iOS and Android applications has required the developer to use platform-specific programming languages and development environments. For iOS development, this means using Objective C or Swift and for Android development using JVM-based languages such as Java, Scala or Kotlin. Releasing an application for both these platforms technically requires developing two separate applications with different programming languages. This requires lots of development resources.

One of the popular approaches to unify the platform-specific development has been to utilize the browser as the rendering engine. Cordova is one of the most popular platforms for building cross-platform applications. It allows for developing multi-platform applications using standard web technologies - HTML5, CSS3, and JavaScript. However, Cordova applications are running within an embedded browser window in the user's device. That is why these applications can not achieve the performance nor the look-and-feel of native applications that utilize actual native user interface components.

React Native is a framework for developing native Android and iOS applications using JavaScript and React. It provides a set of cross-platform components that behind the scenes utilize the platform's native components. Using React Native allows us to bring all the familiar features of React such as JSX, components, props, state, and hooks into native application development. On top of that, we can utilize many familiar libraries in the React ecosystem such as React Redux, Apollo, React Router and many more.

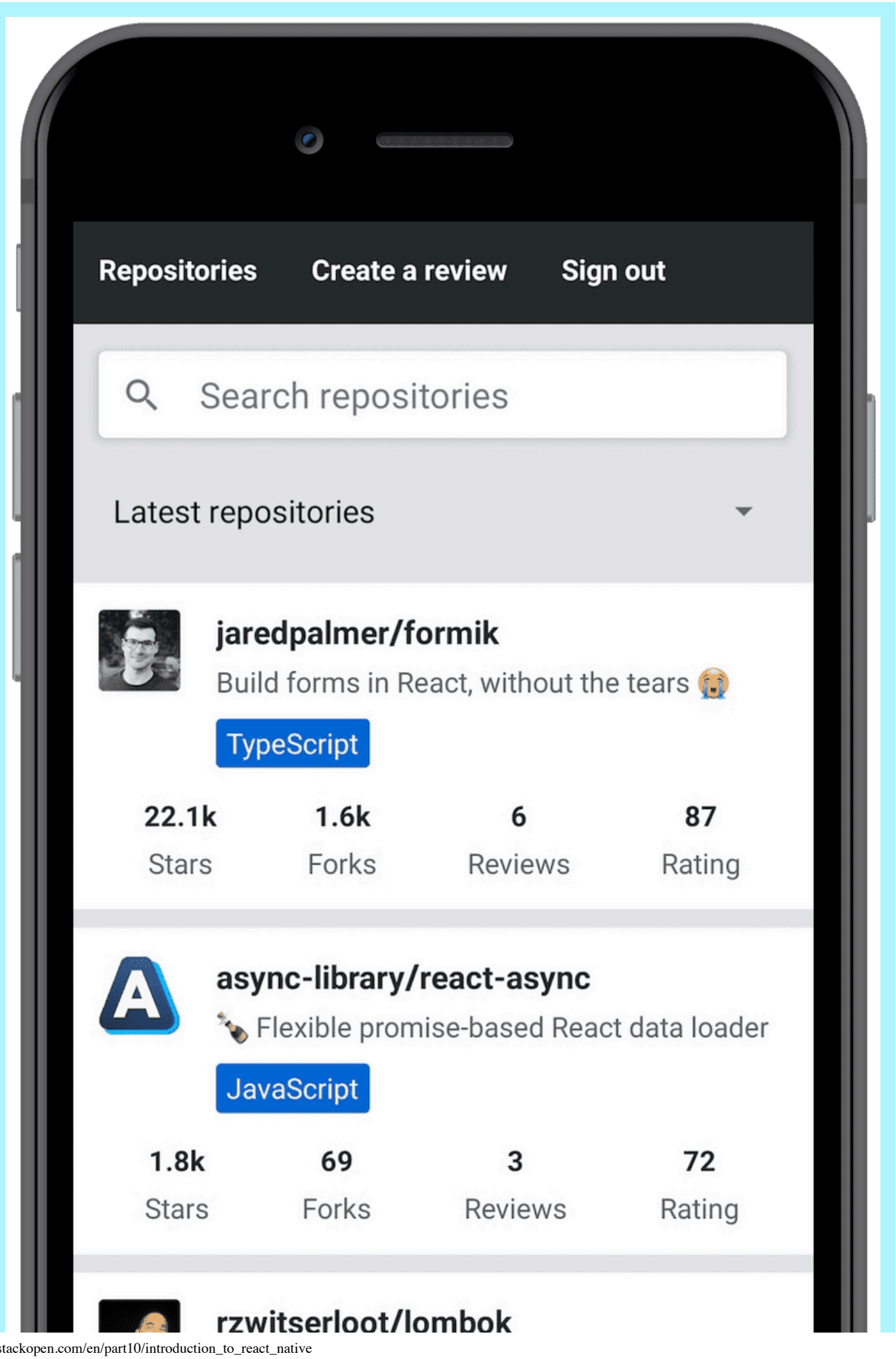
The speed of development and gentle learning curve for developers familiar with React is one of the most important benefits of React Native. Here's a motivational quote from Coinbase's article Onboarding thousands of users with React Native on the benefits of React Native:

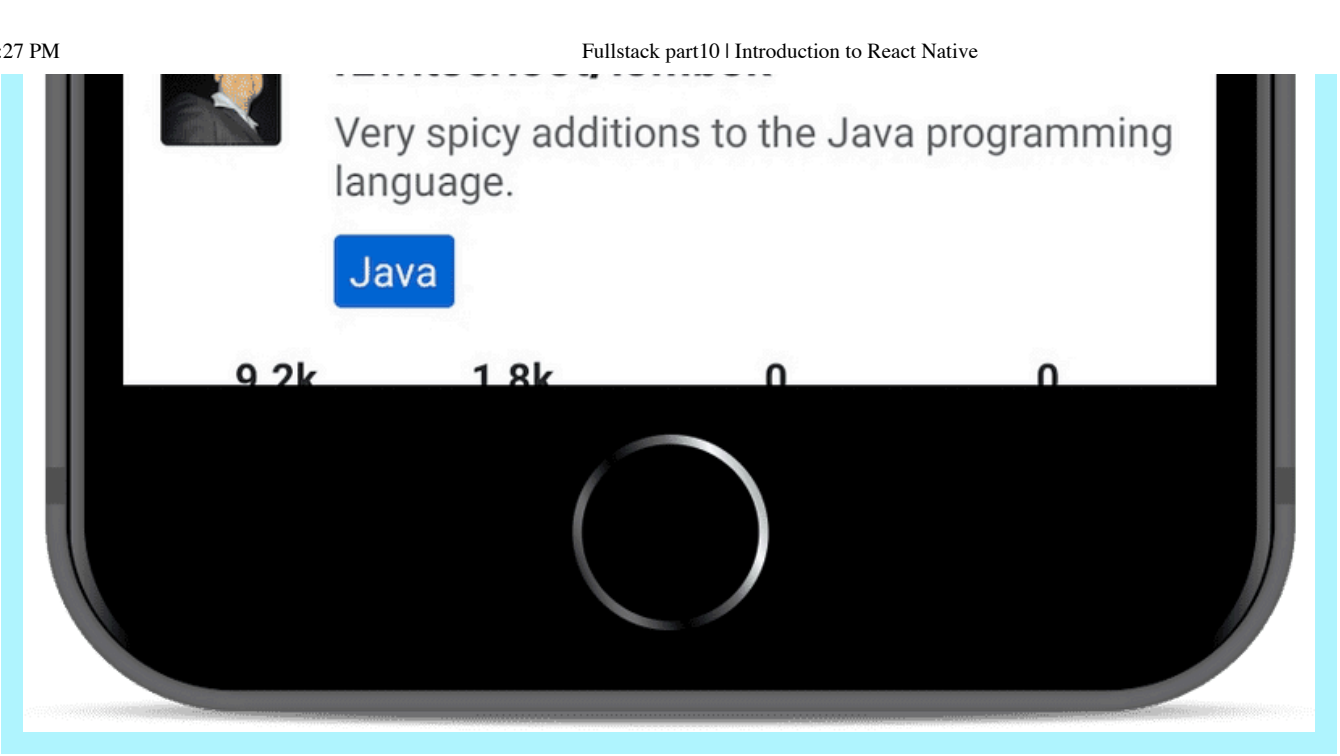
> *If we were to reduce the benefits of React Native to a single word, it would be "velocity". On average, our team was able to onboard engineers in less time, share more code (which we expect will lead to future productivity boosts), and ultimately deliver features faster than if we had taken a purely native approach.*

About this part

During this part, we will learn how to build an actual React Native application from the bottom up. We will learn concepts such as what are React Native's core components, how to create beautiful user interfaces, how to communicate with a server and how to test a React Native application.

We will be developing an application for rating GitHub repositories. Our application will have features such as, sorting and filtering reviewed repositories, registering a user, logging in and creating a review for a repository. The back end for the application will be provided for us so that we can solely focus on the React Native development. The final version of our application will look something like this:


Repositories
Create a review
Sign out
Search repositories
Latest repositories
jaredpalmer/formik
Build forms in React, without the tears 😭
TypeScript
22.1k
Stars
1.6k
Forks
6
Reviews
87
Rating
async-library/react-async
🍾 Flexible promise-based React data loader
JavaScript
1.8k
Stars
69
Forks
3
Reviews
72
Rating
rzwitserloot/lombok



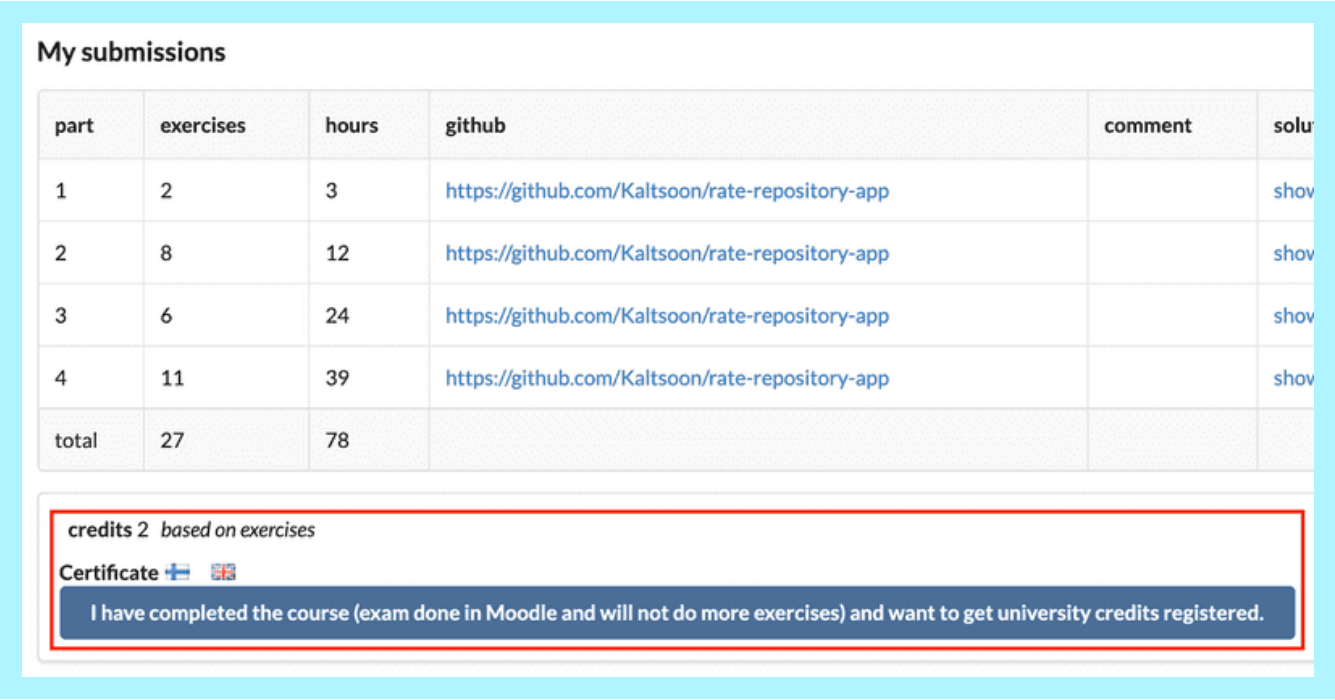
All the exercises in this part have to be submitted into *a single GitHub repository* which will eventually contain the entire source code of your application. There will be model solutions available for each section of this part which you can use to fill in incomplete submissions. This part is structured based on the idea that you develop your application as you progress in the material. So *do not* wait until the exercises to start the development. Instead, develop your application at the same pace as the material progresses.

This part will heavily rely on concepts covered in the previous parts. Before starting this part you will need basic knowledge of JavaScript, React and GraphQL. Deep knowledge of server-side development is not required and all the server-side code is provided for you. However, we will be making network requests from your React Native applications, for example, using GraphQL queries. The recommended parts to complete before this part are part 1, part 2, part 5, part 7 and part 8.

Submitting exercises and earning credits

Exercises are submitted via the submissions system just like in the previous parts. Note that, exercises in this part are submitted *to a different course instance* than in parts 0-9. Parts 1-4 in the submission system refer to sections a-d in this part. This means that you will be submitting exercises a single section at a time starting with this section, "Introduction to React Native", which is part 1 in the submission system.

During this part, you will earn credits based on the number of exercises you complete. Completing *at least 25 exercises* in this part will earn you *2 credits*. Once you have completed the exercises and want to get the credits, let us know through the exercise submission system that you have completed the course:



Note that you need a registration to the corresponding course part for getting the credits registered, see here for more information.

You can download the certificate for completing this part by clicking one of the flag icons. The flag icon corresponds to the certificate's language. Note that you must have completed at least one credit worth of exercises before you can download the certificate.

Initializing the application

To get started with our application we need to set up our development environment. We have learned from previous parts that there are useful tools for setting up React applications quickly such as Create React App. Luckily React Native has these kinds of tools as well.

For the development of our application, we will be using Expo. Expo is a platform that eases the setup, development, building, and deployment of React Native applications. Let's get started with Expo by initializing our project with *create-expo-app*:

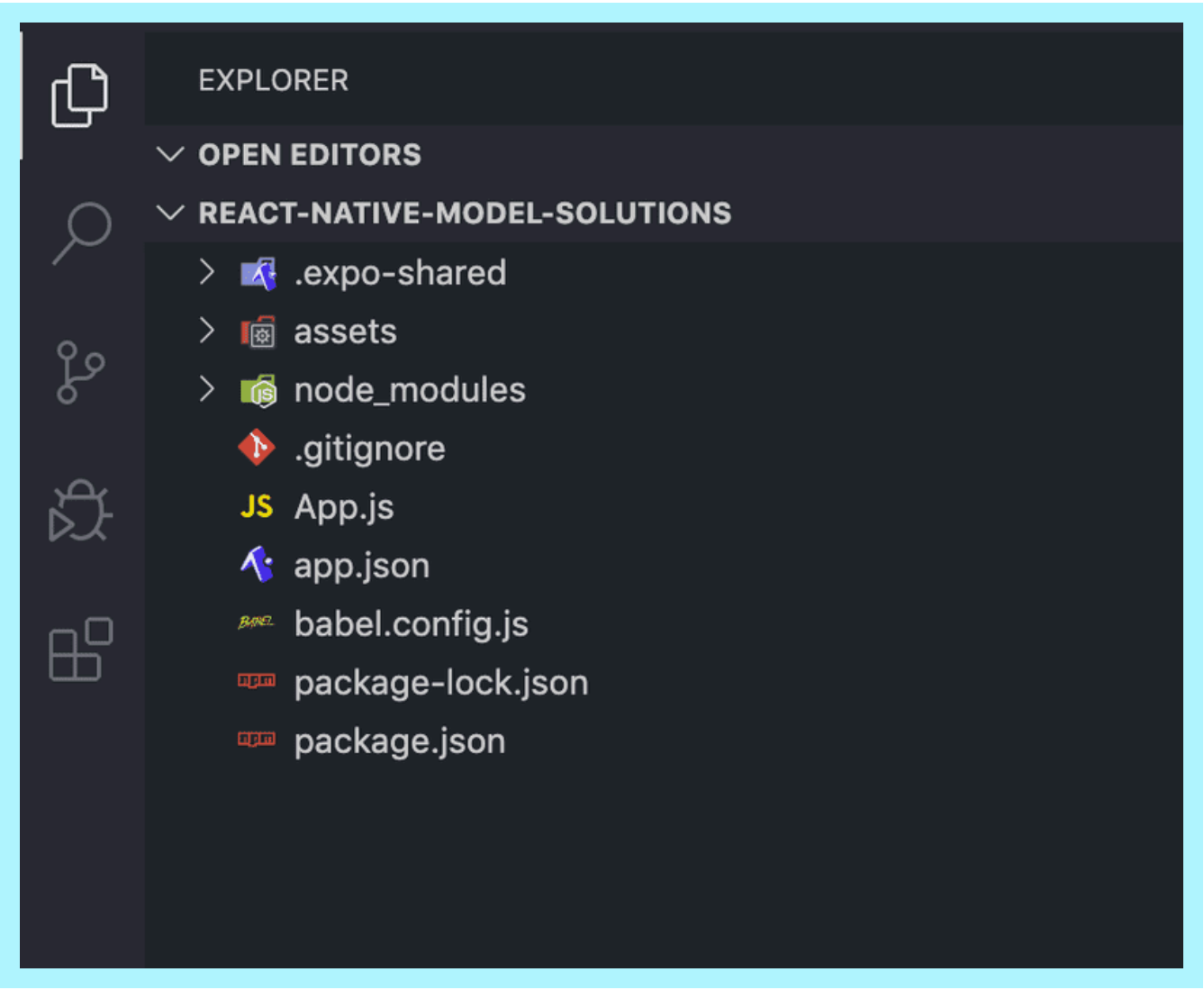
```
npx create-expo-app rate-repository-app --template expo-template-blank@sdk-50
```

Note, that the `@sdk-50` sets the project's *Expo SDK version to 50*, which supports *React Native version 0.73*. Using other Expo SDK versions might cause you trouble while following this material. Also, Expo has a few limitations when compared to plain React Native CLI. However, these limitations do not affect the application implemented in the material.

Next, let's navigate to the created *rate-repository-app* directory with the terminal and install a few dependencies we'll be needing soon:

```
npx expo install react-native-web@~0.19.6 react-dom@18.2.0 @expo/metro-runtime@~3.1.1
```
copy

Now that our application has been initialized, open the created *rate-repository-app* directory with an editor such as Visual Studio Code. The structure should be more or less the following:



We might spot some familiar files and directories such as *package.json* and *node_modules*. On top of those, the most relevant files are the *app.json* file which contains Expo-related configuration and *App.js* which is the root component of our application. *Do not* rename or move the *App.js* file because by default Expo imports it to register the root component.

Let's look at the *scripts* section of the *package.json* file which has the following scripts:

```
{
  // ...
  "scripts": {
    "start": "expo start",
    "android": "expo start --android",
    "ios": "expo start --ios",
```
copy

```
    "web": "expo start --web"
  },
  // ...
}
```

Let us now run the script `npm start`

> *If the script fails with error the problem is most likely your Node version. In case of problems, switch to version* `20`.

The script starts the Metro bundler which is a JavaScript bundler for React Native. In addition to the Metro bundler, the Expo command-line interface should be open in the terminal window. The command-line interface has a useful set of commands for viewing the application logs and starting the application in an emulator or in Expo's mobile application. We will get to emulators and Expo's mobile application soon, but first, let's open our application.

Expo command-line interface suggests a few ways to open our application. Let's press the "w" key in the terminal window to open the application in a browser. We should soon see the text defined in the *App.js* file in a browser window. Open the *App.js* file with an editor and make a small change to the text in the `Text` component. After saving the file you should be able to see that the changes you have made in the code are visible in the browser window.

Setting up the development environment

We have had the first glance of our application using the Expo's browser view. Although the browser view is quite usable, it is still a quite poor simulation of the native environment. Let's have a look at the alternatives we have regarding the development environment.

Android and iOS devices such as tablets and phones can be emulated in computers using specific *emulators*. This is very useful for developing native applications. macOS users can use both Android and iOS emulators with their computers. Users of other operating systems such as Linux or Windows have to settle for Android emulators. Next, depending on your operating system follow one of these instructions on setting up an emulator:

- Set up the Android emulator with Android Studio (any operating system)
- Set up the iOS simulator with Xcode (macOS operating system)

After you have set up the emulator and it is running, start the Expo development tools as we did before, by running `npm start`. Depending on the emulator you are running either press the corresponding key for the "open Android" or "open iOS simulator". After pressing the key, Expo should connect to the emulator and you should eventually see the application in your emulator. Be patient, this might take a while.

In addition to emulators, there is one extremely useful way to develop React Native applications with Expo, the Expo mobile app. With the Expo mobile app, you can preview your application using your actual mobile device, which provides a bit more concrete development experience compared to emulators. To get started, install the Expo mobile app by following the instructions in the Expo's documentation. Note that the Expo mobile app can only open your application if your mobile device is connected to *the same local network* (e.g. connected to the same Wi-Fi network) as the computer you are using for development.

When the Expo mobile app has finished installing, open it up. Next, if the Expo development tools are not already running, start them by running `npm start`. You should be able to see a QR code at the beginning of the command output. Open the app by scanning the QR code, in Android with Expo app or in iOS with the Camera app. The Expo mobile app should start building the JavaScript bundle and after it is finished you should be able to see your application. Now, every time you want to reopen your application in the Expo mobile app, you should be able to access the application without scanning the QR code by pressing it in the *Recently opened* list in the *Projects* view.

Exercise 10.1

Exercise 10.1: initializing the application

Initialize your application with Expo command-line interface and set up the development environment either using an emulator or Expo's mobile app. It is recommended to try both and find out which

development environment is the most suitable for you. The name of the application is not that relevant. You can, for example, go with *rate-repository-app*.

To submit this exercise and all future exercises you need to create a new GitHub repository. The name of the repository can be for example the name of the application you initialized with `expo init`. If you decide to create a private repository, add GitHub user mluukkai as a repository collaborator. The collaborator status is only used for verifying your submissions.

Now that the repository is created, run `git init` within your application's root directory to make sure that the directory is initialized as a Git repository. Next, to add the created repository as the remote run `git remote add origin git@github.com:<YOUR GITHUB USERNAME>/<NAME OF YOUR_REPOSITORY>.git` (remember to replace the placeholder values in the command). Finally, just commit and push your changes into the repository and you are all done.

ESLint

Now that we are somewhat familiar with the development environment let's enhance our development experience even further by configuring a linter. We will be using ESLint which is already familiar to us from the previous parts. Let's get started by installing the dependencies:

```
npm install --save-dev eslint @babel/eslint-parser eslint-plugin-react eslint-plugin-react-native
```

copy

Next, let's add the ESLint configuration into a *.eslintrc.json* file in the *rate-repository-app* directory with the following content:

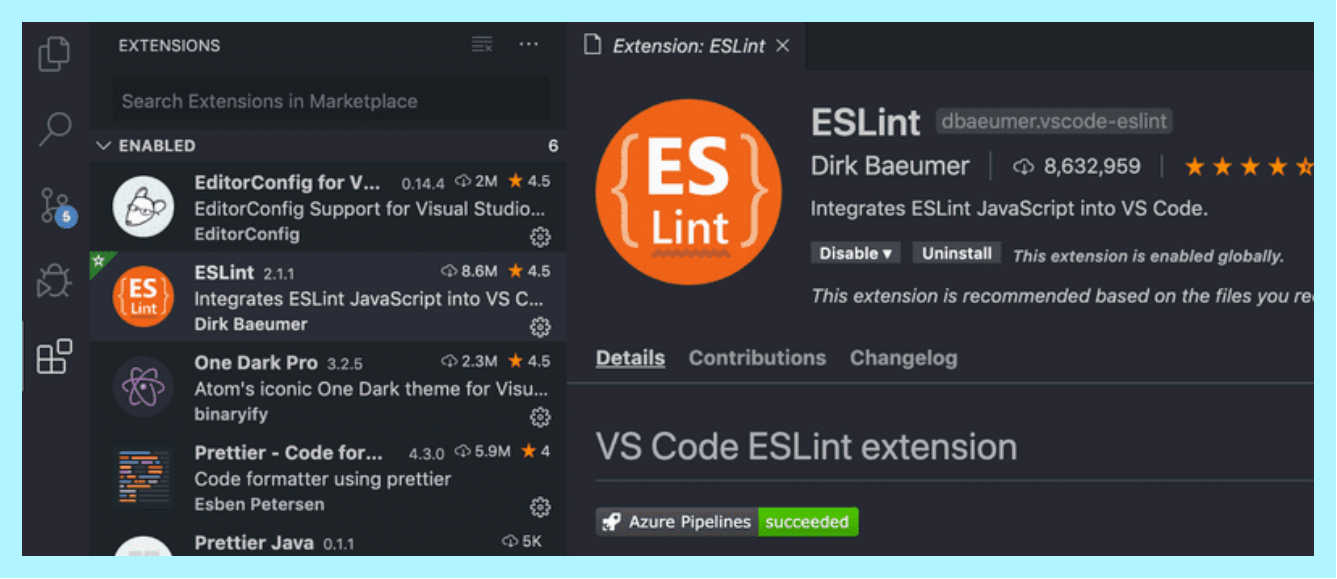
```
{
  "plugins": ["react", "react-native"],
  "settings": {
    "react": {
      "version": "detect"
    }
  },
  "extends": ["eslint:recommended", "plugin:react/recommended"],
  "parser": "@babel/eslint-parser",
  "env": {
    "react-native/react-native": true
  },
  "rules": {
    "react/prop-types": "off",
    "react/react-in-jsx-scope": "off"
  }
}
```

copy

And finally, let's add a `lint` script to the *package.json* file to check the linting rules in specific files:

```
{
  // ...
  "scripts": {
    "start": "expo start",
    "android": "expo start --android",
    "ios": "expo start --ios",
    "web": "expo start --web",
    "lint": "eslint ./src/**/*.{js,jsx} App.js --no-error-on-unmatched-pattern"
  },
  // ...
}
```

Now we can check that the linting rules are obeyed in JavaScript files in the *src* directory and in the *App.js* file by running `npm run lint`. We will be adding our future code to the *src* directory but because we haven't added any files there yet, we need the `no-error-on-unmatched-pattern` flag. Also if possible integrate ESLint with your editor. If you are using Visual Studio Code you can do that by, going to the extensions section and checking that the ESLint extension is installed and enabled:



The provided ESLint configuration contains only the basis for the configuration. Feel free to improve the configuration and add new plugins if you feel like it.

Exercise 10.2

Exercise 10.2: setting up the ESLint

Set up ESLint in your project so that you can perform linter checks by running `npm run lint`. To get most of linting it is also recommended to integrate ESLint with your editor.

This was the last exercise in this section. It's time to push your code to GitHub and mark all of your finished exercises to the exercise submission system. Note that exercises in this section should be submitted to part 1 in the exercise submission system.

Debugging

When our application doesn't work as intended, we should immediately start *debugging* it. In practice, this means that we'll need to reproduce the erroneous behavior and monitor the code execution to find out which part of the code behaves incorrectly. During the course, we have already done a bunch of debugging by logging messages, inspecting network traffic, and using specific development tools, such as *React Development Tools*. In general, debugging isn't that different in React Native, we'll just need the right tools for the job.

The good old console.log messages appear in the Expo development tools command line:

That might actually be enough in most cases, but sometimes we need more. React Native provides an in-app developer menu which offers several debugging options. Read more about debugging react native applications.

To inspect the React element tree, props, and state you can install React DevTools.

```
npx react-devtools
```

copy

Read here about React DevTools. For more useful React Native application debugging tools, also head out to the Expo's debugging documentation.

Propose changes to material

Part 9
Previous part

Part 10b
Next part

About course

Course contents

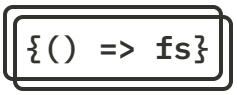
FAQ

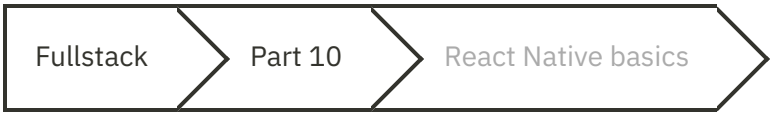
Partners

Challenge


UNIVERSITY OF HELSINKI


HOUSTON





b React Native basics

Note: This course material was updated in Feb 2024. Some updates are not compatible anymore with older material. We recommend a fresh start with this new Part 10 material. However, if you´re returning to this course after a break, and you want to continue the exercises in your older project, please use Part 10 material before the upgrade.

Now that we have set up our development environment we can get into React Native basics and get started with the development of our application. In this section, we will learn how to build user interfaces with React Native's core components, how to add style properties to these core components, how to transition between views, and how to manage the form's state efficiently.

Core components

In the previous parts, we have learned that we can use React to define components as functions, which receive props as an argument and returns a tree of React elements. This tree is usually represented with JSX syntax. In the browser environment, we have used the ReactDOM library to turn these components into a DOM tree that can be rendered by a browser. Here is a concrete example of a very simple component:

```
const HelloWorld = props => {
  return <div>Hello world!</div>;
};
```

The `HelloWorld` component returns a single *div* element which is created using the JSX syntax. We might remember that this JSX syntax is compiled into `React.createElement` method calls, such as this:

```
React.createElement('div', null, 'Hello world!');
```

copy

This line of code creates a *div* element without any props and with a single child element which is a string *"Hello world"*. When we render this component into a root DOM element using the `ReactDOM.render` method the *div* element will be rendered as the corresponding DOM element.

As we can see, React is not bound to a certain environment, such as the browser environment. Instead, there are libraries such as ReactDOM that can render *a set of predefined components*, such as DOM elements, in a specific environment. In React Native these predefined components are called *core components*.

Core components are a set of components provided by React Native, which behind the scenes utilize the platform's native components. Let's implement the previous example using React Native:

```
import { Text } from 'react-native';

const HelloWorld = props => {
  return <Text>Hello world!</Text>;
};
```

copy

So we import the Text component from React Native and replace the `div` element with a `Text` element. Many familiar DOM elements have their React Native "counterparts". Here are some examples picked from React Native's Core Components documentation:

- Text component is *the only* React Native component that can have textual children. It is similar to for example the `<strong>` and the `<h1>` elements.
- View component is the basic user interface building block similar to the `<div>` element.
- TextInput component is a text field component similar to the `<input>` element.
- Pressable component is for capturing different press events. It is similar to for example the `<button>` element.

There are a few notable differences between core components and DOM elements. The first difference is that the `Text` component is *the only* React Native component that can have textual children. This means that you can't, for example, replace the `Text` component with the `View` component in the previous example.

The second notable difference is related to the event handlers. While working with the DOM elements we are used to adding event handlers such as `onClick` to basically any element such as `<div>` and `<button>`. In React Native we have to carefully read the API documentation to know what event handlers (as well as other props) a component accepts. For example, the Pressable component provides props for listening to different kinds of press events. We can for example use the component's onPress prop for listening to press events:

```
import { Text, Pressable, Alert } from 'react-native';

const PressableText = props => {
  return (
    <Pressable
      onPress={() => Alert.alert('You pressed the text!')}
    >
      <Text>You can press me</Text>
    </Pressable>
  );
};
```

Now that we have a basic understanding of the core components, let's start to give our project some structure. Create a *src* directory in the root directory of your project and in the *src* directory create a *components* directory. In the *components* directory create a file *Main.jsx* with the following content:

```
import Constants from 'expo-constants';
import { Text, StyleSheet, View } from 'react-native';

const styles = StyleSheet.create({
  container: {
    marginTop: Constants.statusBarHeight,
    flexGrow: 1,
    flexShrink: 1,
  },
});

const Main = () => {
  return (
    <View style={styles.container}>
      <Text>Rate Repository Application</Text>
    </View>
  );
};

export default Main;
```

Next, let's use the `Main` component in the `App` component in the *App.js* file which is located in our project's root directory. Replace the current content of the file with this:

```
import Main from './src/components/Main';

const App = () => {
  return <Main />;
};

export default App;
```

Manually reloading the application

As we have seen, Expo will automatically reload the application when we make changes to the code. However, there might be times when automatic reload isn't working and the application has to be reloaded manually. This can be achieved through the in-app developer menu.

You can access the developer menu by shaking your device or by selecting "Shake Gesture" inside the Hardware menu in the iOS Simulator. You can also use the `⌘D` keyboard shortcut when your app is running in the iOS Simulator, or `⌘M` when running in an Android emulator on Mac OS and `Ctrl+M` on Windows and Linux.

Once the developer menu is open, simply press "Reload" to reload the application. After the application has been reloaded, automatic reloads should work without the need for a manual reload.

Exercise 10.3

Exercise 10.3: the reviewed repositories list

In this exercise, we will implement the first version of the reviewed repositories list. The list should contain the repository's full name, description, language, number of forks, number of stars, rating average and number of reviews. Luckily React Native provides a handy component for displaying a list of data, which is the FlatList component.

Implement components `RepositoryList` and `RepositoryItem` in the *components* directory's files *RepositoryList.jsx* and *RepositoryItem.jsx*. The `RepositoryList` component should render the `FlatList` component and `RepositoryItem` a single item on the list (hint: use the `FlatList` component's renderItem prop). Use this as the basis for the *RepositoryList.jsx* file:

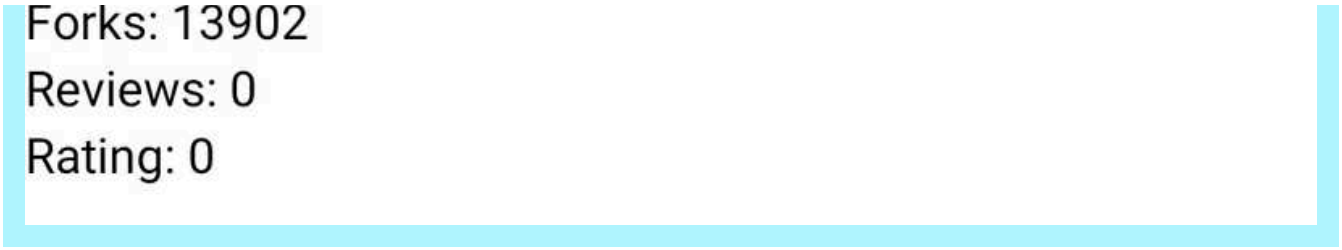
```
import { FlatList, View, StyleSheet } from 'react-native';

const styles = StyleSheet.create({
  separator: {
    height: 10,
  },
});

const repositories = [
  {
    id: 'jaredpalmer.formik',
    fullName: 'jaredpalmer/formik',
    description: 'Build forms in React, without the tears',
    language: 'TypeScript',
    forksCount: 1589,
    stargazersCount: 21553,
    ratingAverage: 88,
```

```
    reviewCount: 4,
    ownerAvatarUrl: 'https://avatars2.githubusercontent.com/u/4060187?v=4',
  },
  {
    id: 'rails.rails',
    fullName: 'rails/rails',
    description: 'Ruby on Rails',
    language: 'Ruby',
    forksCount: 18349,
    stargazersCount: 45377,
    ratingAverage: 100,
    reviewCount: 2,
    ownerAvatarUrl: 'https://avatars1.githubusercontent.com/u/4223?v=4',
  },
  {
    id: 'django.django',
    fullName: 'django/django',
    description: 'The Web framework for perfectionists with deadlines.',
    language: 'Python',
    forksCount: 21015,
    stargazersCount: 48496,
    ratingAverage: 73,
    reviewCount: 5,
    ownerAvatarUrl: 'https://avatars2.githubusercontent.com/u/27804?v=4',
  },
  {
    id: 'reduxjs.redux',
    fullName: 'reduxjs/redux',
    description: 'Predictable state container for JavaScript apps',
    language: 'TypeScript',
    forksCount: 13902,
    stargazersCount: 52869,
    ratingAverage: 0,
    reviewCount: 0,
    ownerAvatarUrl: 'https://avatars3.githubusercontent.com/u/13142323?v=4',
  },
];

const ItemSeparator = () => <View style={styles.separator} />;

const RepositoryList = () => {
  return (
    <FlatList
      data={repositories}
      ItemSeparatorComponent={ItemSeparator}
      // other props
    />
  );
};

export default RepositoryList;
```

Do not alter the contents of the `repositories` variable, it should contain everything you need to complete this exercise. Render the `RepositoryList` component in the `Main` component which we

previously added to the *Main.jsx* file. The reviewed repository list should roughly look something like this:


15.19
Full name: jaredpalmer/formik
Description: Build forms in React, without the tears
Language: TypeScript
Stars: 21553
Forks: 1589
Reviews: 4
Rating: 88
Full name: rails/rails
Description: Ruby on Rails
Language: Ruby
Stars: 45377
Forks: 18349
Reviews: 2
Rating: 100
Full name: django/django
Description: The Web framework for perfectionists with deadlines.
Language: Python
Stars: 48496
Forks: 21015
Reviews: 5
Rating: 73
Full name: reduxjs/redux
Description: Predictable state container for JavaScript apps
Language: TypeScript
Stars: 52869



Style

Now that we have a basic understanding of how core components work and we can use them to build a simple user interface it is time to add some style. In part 2 we learned that in the browser environment we can define React component's style properties using CSS. We had the option to either define these styles inline using the `style` prop or in a CSS file with a suitable selector.

There are many similarities in the way style properties are attached to React Native's core components and the way they are attached to DOM elements. In React Native most of the core components accept a prop called `style`. The `style` prop accepts an object with style properties and their values. These style properties are in most cases the same as in CSS, however, property names are in *camelCase*. This means that CSS properties such as `padding-top` and `font-size` are written as `paddingTop` and `fontSize`. Here is a simple example of how to use the `style` prop:

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

const BigBlueText = () => {
  return (
    <View style={{ padding: 20 }}>
      <Text style={{ color: 'blue', fontSize: 24, fontWeight: '700' }}>
        Big blue text
      </Text>
    </View>
  );
};
```

On top of the property names, you might have noticed another difference in the example. In CSS numerical property values commonly have a unit such as *px*, *%*, *em* or *rem*. In React Native all dimension-related property values such as `width`, `height`, `padding`, and `margin` as well as font sizes are *unitless*. These unitless numeric values represent *density-independent pixels*. In case you are wondering what are the available style properties for certain core components, check the React Native Styling Cheat Sheet.

In general, defining styles directly in the `style` prop is not considered such a great idea, because it makes components bloated and unclear. Instead, we should define styles outside the component's render function using the StyleSheet.create method. The `StyleSheet.create` method accepts a single argument which is an object consisting of named style objects and it creates a StyleSheet style

reference from the given object. Here is an example of how to refactor the previous example using the `StyleSheet.create` method:

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

const styles = StyleSheet.create({
  container: {
    padding: 20,
  },
  text: {
    color: 'blue',
    fontSize: 24,
    fontWeight: '700',
  },
});

const BigBlueText = () => {
  return (
    <View style={styles.container}>
      <Text style={styles.text}>
        Big blue text
      </Text>
    </View>
  );
};
```

We create two named style objects, `styles.container` and `styles.text`. Inside the component, we can access specific style objects the same way we would access any key in a plain object.

In addition to an object, the `style` prop also accepts an array of objects. In the case of an array, the objects are merged from left to right so that latter-style properties take precedence. This works recursively, so we can have for example an array containing an array of styles and so forth. If an array contains values that evaluate to false, such as `null` or `undefined`, these values are ignored. This makes it easy to define *conditional styles* for example, based on the value of a prop. Here is an example of conditional styles:

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

const styles = StyleSheet.create({
  text: {
    color: 'grey',
    fontSize: 14,
  },
  blueText: {
    color: 'blue',
  },
  bigText: {
    fontSize: 24,
    fontWeight: '700',
```

```
  },
});

const FancyText = ({ isBlue, isBig, children }) => {
  const textStyles = [
    styles.text,
    isBlue && styles.blueText,
    isBig && styles.bigText,
  ];

  return <Text style={textStyles}>{children}</Text>;
};

const Main = () => {
  return (
    <>
      <FancyText>Simple text</FancyText>
      <FancyText isBlue>Blue text</FancyText>
      <FancyText isBig>Big text</FancyText>
      <FancyText isBig isBlue>
        Big blue text
      </FancyText>
    </>
  );
};
```

In the example, we use the `&&` operator with the expression `condition && exprIfTrue`. This expression yields `exprIfTrue` if the `condition` evaluates to true, otherwise it will yield `condition`, which in that case is a value that evaluates to false. This is an extremely widely used and handy shorthand. Another option would be to use the conditional operator like this:

```
condition ? exprIfTrue : exprIfFalse
```

copy

Consistent user interface with theming

Let's stick with the concept of styling but with a bit wider perspective. Most of us have used a multitude of different applications and might agree that one trait that makes a good user interface is *consistency*. This means that the appearance of user interface components such as their font size, font family and color follows a consistent pattern. To achieve this we have to somehow *parametrize* the values of different style properties. This method is commonly known as *theming*.

Users of popular user interface libraries such as Bootstrap and Material UI might already be quite familiar with theming. Even though the theming implementations differ, the main idea is always to use variables such as `colors.primary` instead of "magic numbers" such as `#0366d6` when defining styles. This leads to increased consistency and flexibility.

Let's see how theming could work in practice in our application. We will be using a lot of text with different variations, such as different font sizes and colors. Because React Native does not support

global styles, we should create our own `Text` component to keep the textual content consistent. Let's get started by adding the following theme configuration object in a *theme.js* file in the *src* directory:

```
const theme = {
  colors: {
    textPrimary: '#24292e',
    textSecondary: '#586069',
    primary: '#0366d6',
  },
  fontSizes: {
    body: 14,
    subheading: 16,
  },
  fonts: {
    main: 'System',
  },
  fontWeights: {
    normal: '400',
    bold: '700',
  },
};

export default theme;
```

Next, we should create the actual `Text` component which uses this theme configuration. Create a *Text.jsx* file in the *components* directory where we already have our other components. Add the following content to the *Text.jsx* file:

```
import { Text as NativeText, StyleSheet } from 'react-native';

import theme from '../theme';

const styles = StyleSheet.create({
  text: {
    color: theme.colors.textPrimary,
    fontSize: theme.fontSizes.body,
    fontFamily: theme.fonts.main,
    fontWeight: theme.fontWeights.normal,
  },
  colorTextSecondary: {
    color: theme.colors.textSecondary,
  },
  colorPrimary: {
    color: theme.colors.primary,
  },
  fontSizeSubheading: {
    fontSize: theme.fontSizes.subheading,
  },
  fontWeightBold: {
```

```
    fontWeight: theme.fontWeights.bold,
  },
});

const Text = ({ color, fontSize, fontWeight, style, ...props }) => {
  const textStyle = [
    styles.text,
    color === 'textSecondary' && styles.colorTextSecondary,
    color === 'primary' && styles.colorPrimary,
    fontSize === 'subheading' && styles.fontSizeSubheading,
    fontWeight === 'bold' && styles.fontWeightBold,
    style,
  ];

  return <NativeText style={textStyle} {...props} />;
};

export default Text;
```

Now we have implemented our text component. This text component has consistent color, font size and font weight variants that we can use anywhere in our application. We can get different text variations using different props like this:

```
import Text from './Text';

const Main = () => {
  return (
    <>
      <Text>Simple text</Text>
      <Text style={{ paddingBottom: 10 }}>Text with custom style</Text>
      <Text fontWeight="bold" fontSize="subheading">
        Bold subheading
      </Text>
      <Text color="textSecondary">Text with secondary color</Text>
    </>
  );
};

export default Main;
```

Feel free to extend or modify this component if you feel like it. It might also be a good idea to create reusable text components such as `Subheading` which use the `Text` component. Also, keep on extending and modifying the theme configuration as your application progresses.

Using flexbox for layout

The last concept we will cover related to styling is implementing layouts with flexbox. Those who are more familiar with CSS know that flexbox is not related only to React Native, it has many use cases in

web development as well. Those who know how flexbox works in web development won't probably learn that much from this section. Nevertheless, let's learn or revise the basics of flexbox.

Flexbox is a layout entity consisting of two separate components: a *flex container* and inside it a set of *flex items*. A Flex container has a set of properties that control the flow of its items. To make a component a flex container it must have the style property `display` set as `flex` which is the default value for the `display` property. Here is an example of a flex container:

```
import { View, StyleSheet } from 'react-native';

const styles = StyleSheet.create({
  flexContainer: {
    flexDirection: 'row',
  },
});

const FlexboxExample = () => {
  return <View style={styles.flexContainer}>{/* ... */}</View>;
};
```

Perhaps the most important properties of a flex container are the following:

- flexDirection property controls the direction in which the flex items are laid out within the container. Possible values for this property are `row`, `row-reverse`, `column` (default value) and `column-reverse`. Flex direction `row` will lay out the flex items from left to right, whereas `column` from top to bottom. `*-reverse` directions will just reverse the order of the flex items.
- justifyContent property controls the alignment of flex items along the main axis (defined by the `flexDirection` property). Possible values for this property are `flex-start` (default value), `flex-end`, `center`, `space-between`, `space-around` and `space-evenly`.
- alignItems property does the same as `justifyContent` but for the opposite axis. Possible values for this property are `flex-start`, `flex-end`, `center`, `baseline` and `stretch` (default value).

Let's move on to flex items. As mentioned, a flex container can contain one or many flex items. Flex items have properties that control how they behave in respect of other flex items in the same flex container. To make a component a flex item all you have to do is to set it as an immediate child of a flex container:

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

const styles = StyleSheet.create({
  flexContainer: {
    display: 'flex',
  },
  flexItemA: {
```

```
    flexGrow: 0,
    backgroundColor: 'green',
  },
  flexItemB: {
    flexGrow: 1,
    backgroundColor: 'blue',
  },
});

const FlexboxExample = () => {
  return (
    <View style={styles.flexContainer}>
      <View style={styles.flexItemA}>
        <Text>Flex item A</Text>
      </View>
      <View style={styles.flexItemB}>
        <Text>Flex item B</Text>
      </View>
    </View>
  );
};
```

One of the most commonly used properties of flex items is the flexGrow property. It accepts a unitless value which defines the ability for a flex item to grow if necessary. If all flex items have a `flexGrow` of `1`, they will share all the available space evenly. If a flex item has a `flexGrow` of `0`, it will only use the space its content requires and leave the rest of the space for other flex items.

Here you can find how to simplify layouts with Flexbox gap: Flexbox gap.

Next, read the article A Complete Guide to Flexbox which has comprehensive visual examples of flexbox. It is also a good idea to play around with the flexbox properties in the Flexbox Playground to see how different flexbox properties affect the layout. Remember that in React Native the property names are the same as the ones in CSS except for the *camelCase* naming. However, the *property values* such as `flex-start` and `space-between` are exactly the same.

NB: React Native and CSS has some differences regarding the flexbox. The most important difference is that in React Native the default value for the `flexDirection` property is `column`. It is also worth noting that the `flex` shorthand doesn't accept multiple values in React Native. More on React Native's flexbox implementation can be read in the documentation.

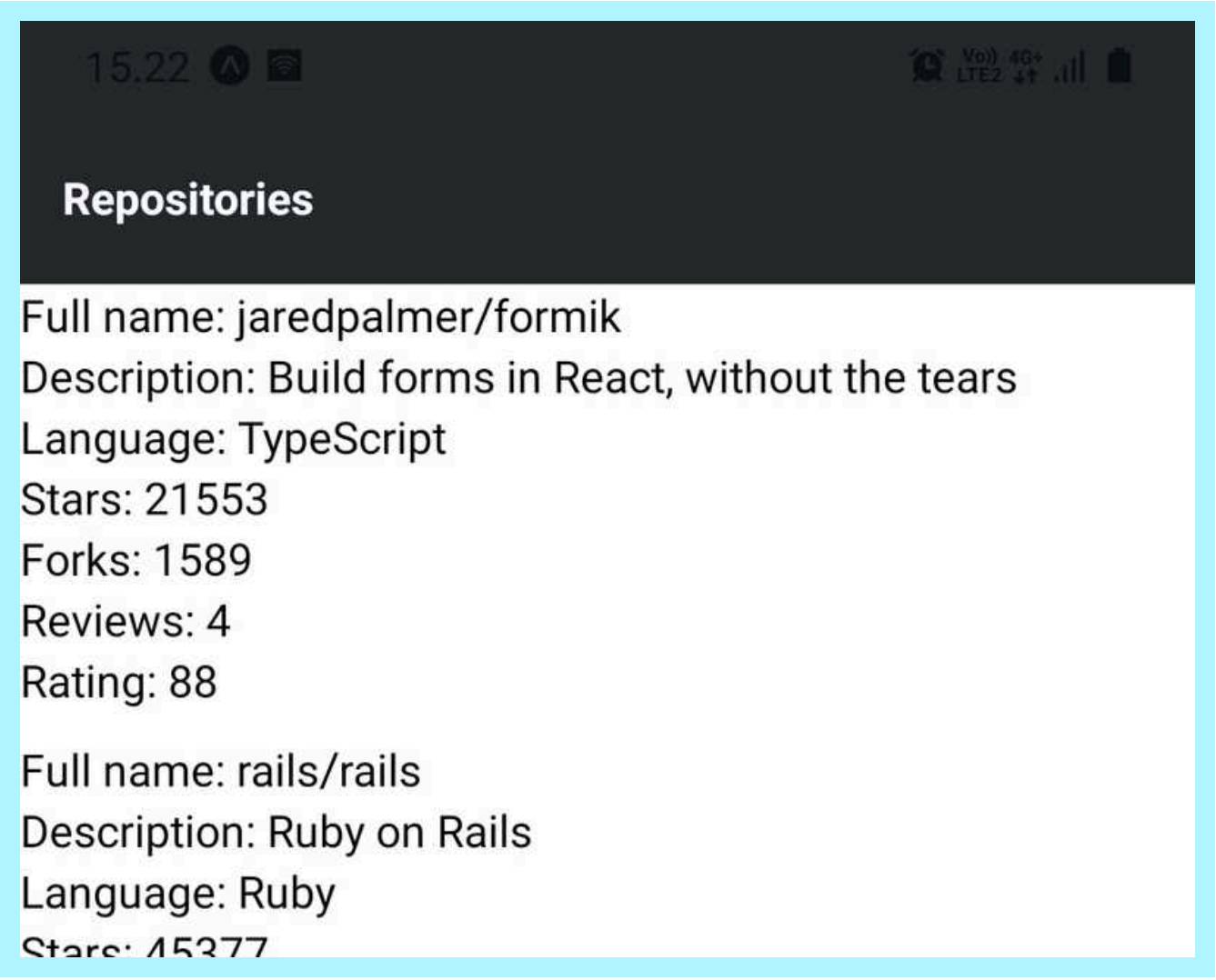
Exercises 10.4-10.5

Exercise 10.4: the app bar

We will soon need to navigate between different views in our application. That is why we need an app bar to display tabs for switching between different views. Create a file *AppBar.jsx* in the *components* folder with the following content:

```
import { View, StyleSheet } from 'react-native';
import Constants from 'expo-constants';

const styles = StyleSheet.create({
  container: {
    paddingTop: Constants.statusBarHeight,
    // ...
  },
  // ...
});

const AppBar = () => {
  return <View style={styles.container}>{/* ... */}</View>;
};

export default AppBar;
```

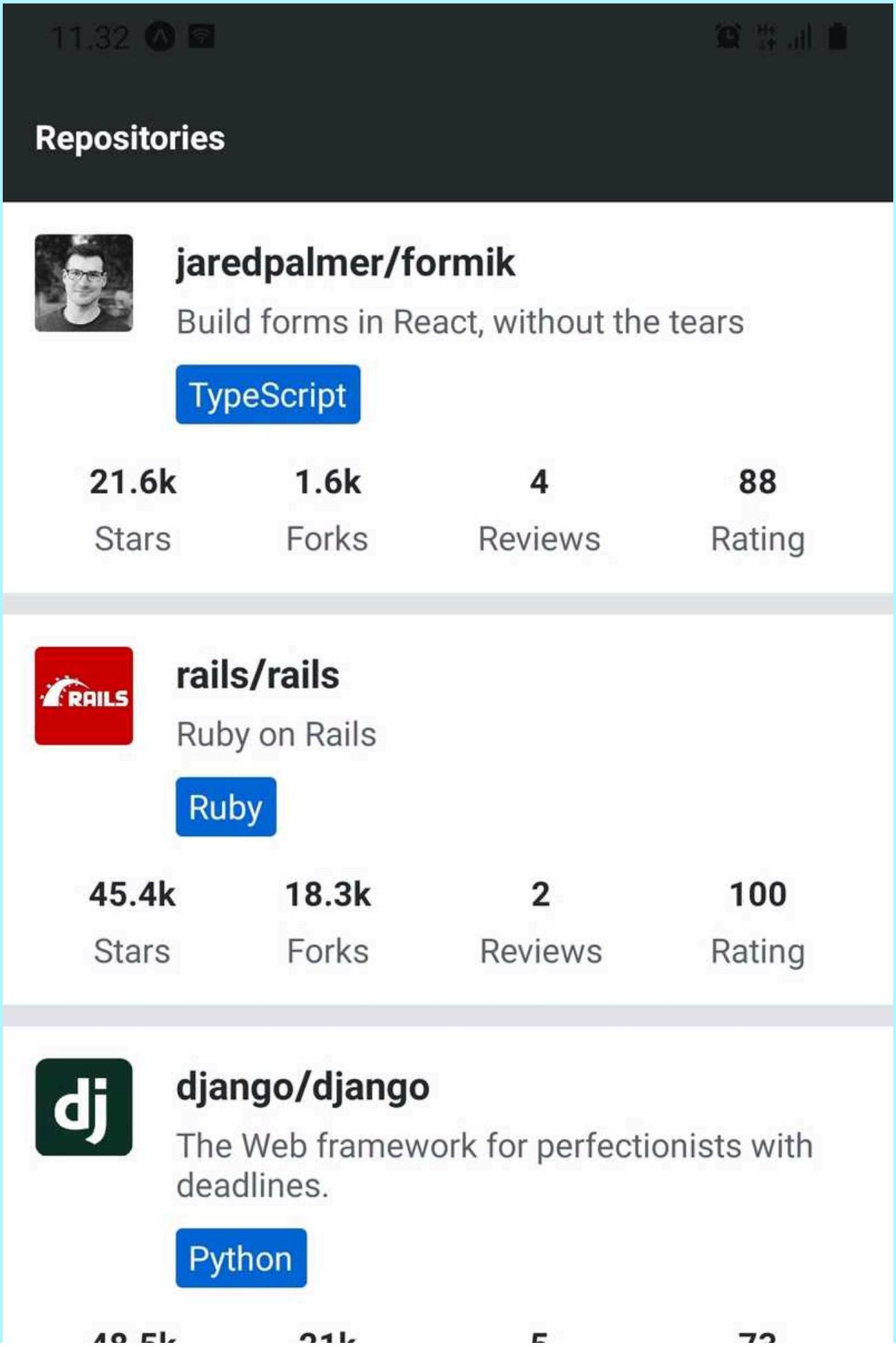
Now that the `AppBar` component will prevent the status bar from overlapping the content, you can remove the `marginTop` style we added for the `Main` component earlier in the *Main.jsx* file. The `AppBar` component should currently contain a tab with the text `"Repositories"`. Make the tab pressable by using the Pressable component but you don't have to handle the `onPress` event in any way. Add the `AppBar` component to the `Main` component so that it is the uppermost component on the screen. The `AppBar` component should look something like this:

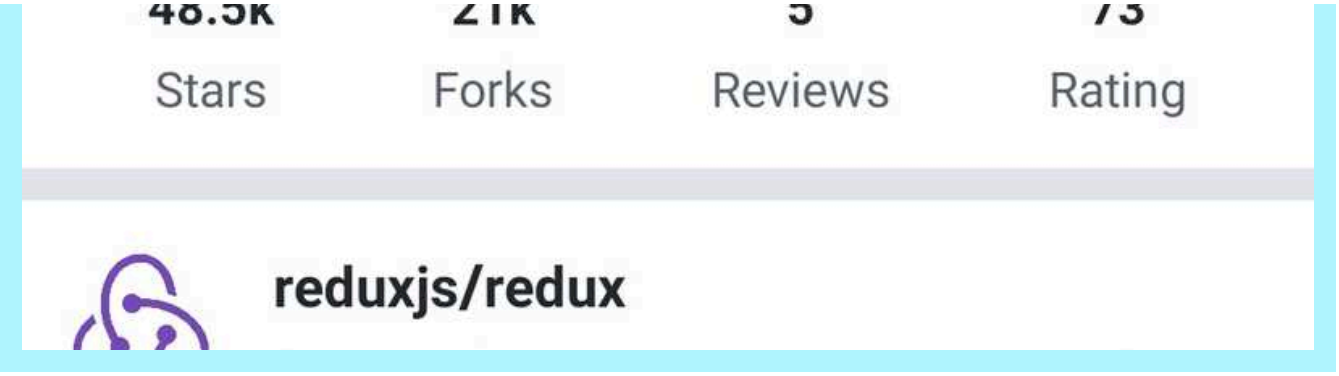


The background color of the app bar in the image is `#24292e` but you can use any other color as well. It might be a good idea to add the app bar's background color into the theme configuration so that it is easy to change it if needed. Another good idea might be to separate the app bar's tab into a component like `AppBarTab` so that it is easy to add new tabs in the future.

Exercise 10.5: polished reviewed repositories list

The current version of the reviewed repositories list looks quite grim. Modify the *RepositoryItem* component so that it also displays the repository author's avatar image. You can implement this by using the `Image` component. Counts, such as the number of stars and forks, larger than or equal to 1000 should be displayed in thousands with the precision of one decimal and with a "k" suffix. This means that for example fork count of 8439 should be displayed as "8.4k". Also, polish the overall look of the component so that the reviewed repositories list looks something like this:


11.32
Repositories
jaredpalmer/formik
Build forms in React, without the tears
TypeScript
21.6k
Stars
1.6k
Forks
4
Reviews
88
Rating
rails/rails
Ruby on Rails
Ruby
45.4k
Stars
18.3k
Forks
2
Reviews
100
Rating
django/django
The Web framework for perfectionists with deadlines.
Python



In the image, the `Main` component's background color is set to `#e1e4e8` whereas `RepositoryItem` component's background color is set to `white`. The language tag's background color is `#0366d6` which is the value of the `colors.primary` variable in the theme configuration. Remember to exploit the `Text` component we implemented earlier. Also when needed, split the `RepositoryItem` component into smaller components.

Routing

When we start to expand our application we will need a way to transition between different views such as the repositories view and the sign-in view. In part 7 we got familiar with React router library and learned how to use it to implement routing in a web application.

Routing in a React Native application is a bit different from routing in a web application. The main difference is that we can't reference pages with URLs, which we type into the browser's address bar, and can't navigate back and forth through the user's history using the browser's history API. However, this is just a matter of the router interface we are using.

With React Native we can use the entire React router's core, including the hooks and components. The only difference to the browser environment is that we must replace the `BrowserRouter` with React Native compatible NativeRouter, provided by the react-router-native library. Let's get started by installing the *react-router-native* library:

```
npm install react-router-native
```

copy

Next, open the *App.js* file and add the `NativeRouter` component to the `App` component:

```
import { StatusBar } from 'expo-status-bar';
import { NativeRouter } from 'react-router-native';

import Main from './src/components/Main';

const App = () => {
  return (
    <>
```

copy

```
      <NativeRouter>
        <Main />
      </NativeRouter>
      <StatusBar style="auto" />
    </>
  );
};

export default App;
```

Once the router is in place, let's add our first route to the Main component in the *Main.jsx* file:

```
import { StyleSheet, View } from 'react-native';
import { Route, Routes, Navigate } from 'react-router-native';

import RepositoryList from './RepositoryList';
import AppBar from './AppBar';
import theme from '../theme';

const styles = StyleSheet.create({
  container: {
    backgroundColor: theme.colors.mainBackground,
    flexGrow: 1,
    flexShrink: 1,
  },
});

const Main = () => {
  return (
    <View style={styles.container}>
      <AppBar />
      <Routes>
        <Route path="/" element={<RepositoryList />} />
        <Route path="*" element={<Navigate to="/" replace />} />
      </Routes>
    </View>
  );
};

export default Main;
```

That's it! The last `Route` inside the `Routes` is for catching paths that don't match any previously defined path. In this case, we want to navigate to the home view.

Exercises 10.6-10.7

Exercise 10.6: the sign-in view

We will soon implement a form, that a user can use to *sign in* to our application. Before that, we must implement a view that can be accessed from the app bar. Create a file *SignIn.jsx* in the *components* directory with the following content:

```
import Text from './Text';

const SignIn = () => {
  return <Text>The sign-in view</Text>;
};

export default SignIn;
```

Set up a route for this `SignIn` component in the `Main` component. Also, add a tab with the text "Sign in" to the app bar next to the "Repositories" tab. Users should be able to navigate between the two views by pressing the tabs (hint: you can use the React router's Link component).

Exercise 10.7: scrollable app bar

As we are adding more tabs to our app bar, it is a good idea to allow horizontal scrolling once the tabs won't fit the screen. The ScrollView component is just the right component for the job.

Wrap the tabs in the `AppBar` component's tabs with a `ScrollView` component:

```
const AppBar = () => {
  return (
    <View style={styles.container}>
      <ScrollView horizontal>{/* ... */}</ScrollView>
    </View>
  );
};
```

Setting the horizontal prop `true` will cause the `ScrollView` component to scroll horizontally once the content won't fit the screen. Note that, you will need to add suitable style properties to the `ScrollView` component so that the tabs will be laid in a *row* inside the flex container. You can make sure that the app bar can be scrolled horizontally by adding tabs until the last tab won't fit the screen. Just remember to remove the extra tabs once the app bar is working as intended.

Form state management

Now that we have a placeholder for the sign-in view the next step would be to implement the sign-in form. Before we get to that let's talk about forms from a wider perspective.

Implementation of forms relies heavily on state management. Using React's `useState` hook for state management might get the job done for smaller forms. However, it will quickly make state management for more complex forms quite tedious. Luckily there are many good libraries in the React ecosystem that ease the state management of forms. One of these libraries is Formik.

The main concepts of Formik are the *context* and the *field*. However, the easiest way to do a simple form submit is by using useFormik(). It is a custom React hook that will return all Formik state and helpers directly.

There are some restrictions concerning the use of UseFormik(). Read this to become familiar with useFormik()

Let's see how this works by creating a form for calculating the body mass index:

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

const initialValues = {
  mass: '',
  height: '',
};

const getBodyMassIndex = (mass, height) => {
  return Math.round(mass / Math.pow(height, 2));
};

const BodyMassIndexForm = ({ onSubmit }) => {
  const formik = useFormik({
    initialValues,
    onSubmit,
  });

  return (
    <View>
      <TextInput
        placeholder="Weight (kg)"
        value={formik.values.mass}
        onChangeText={formik.handleChange('mass')}
      />
      <TextInput
        placeholder="Height (m)"
        value={formik.values.height}
        onChangeText={formik.handleChange('height')}
      />
      <Pressable onPress={formik.handleSubmit}>
        <Text>Calculate</Text>
      </Pressable>
    </View>
  );
};

const BodyMassIndexCalculator = () => {
```

```
  const onSubmit = values => {
    const mass = parseFloat(values.mass);
    const height = parseFloat(values.height);

    if (!isNaN(mass) && !isNaN(height) && height !== 0) {
      console.log(`Your body mass index is: ${getBodyMassIndex(mass, height)}`);
    }
  };

  return <BodyMassIndexForm onSubmit={onSubmit} />;
};

export default BodyMassIndexCalculator;
```

This example is not part of our application, so you don't need to add this code to the application. You can however try it out for example in Expo Snack. Expo Snack is an online editor for React Native, similar to JSFiddle and CodePen. It is a useful platform for quickly trying out code. You can share Expo Snacks with others using a link or embedding them as a *Snack Player* on a website. You might have bumped into Snack Players for example in this material and React Native documentation.

Exercise 10.8

Exercise 10.8: the sign-in form

Implement a sign-in form to the `SignIn` component we added earlier in the *SignIn.jsx* file. The sign-in form should include two text fields, one for the username and one for the password. There should also be a button for submitting the form. You don't need to implement an `onSubmit` callback function, it is enough that the form values are logged using `console.log` when the form is submitted:

```
const onSubmit = (values) => {
  console.log(values);
};
```

copy

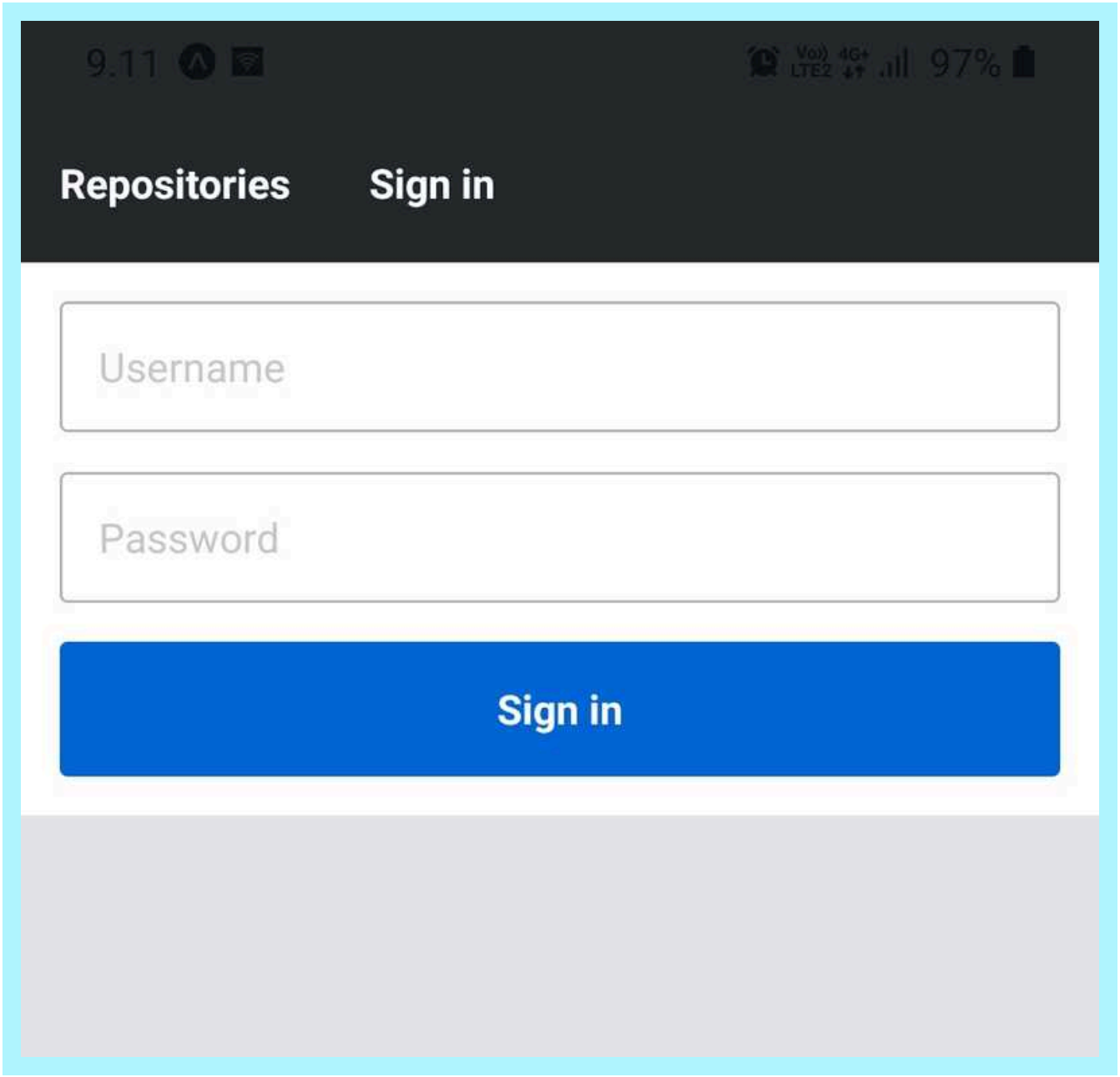
The first step is to install Formik:

```
npm install formik
```

copy

You can use the secureTextEntry prop in the `TextInput` component to obscure the password input.

The sign-in form should look something like this:



Form validation

Formik offers two approaches to form validation: a validation function or a validation schema. A validation function is a function provided for the `Formik` component as the value of the validate prop. It receives the form's values as an argument and returns an object containing possible field-specific error messages.

The second approach is the validation schema which is provided for the `Formik` component as the value of the validationSchema prop. This validation schema can be created with a validation library called Yup. Let's get started by installing Yup:

```
npm install yup
```

Next, as an example, let's create a validation schema for the body mass index form we implemented earlier. We want to validate that both `mass` and `height` fields are present and they are numeric. Also, the value of `mass` should be greater or equal to 1 and the value of `height` should be greater or equal to 0.5. Here is how we define the schema:

```
import * as yup from 'yup';

// ...

const validationSchema = yup.object().shape({
  mass: yup
    .number()
    .min(1, 'Weight must be greater or equal to 1')
    .required('Weight is required'),
  height: yup
    .number()
    .min(0.5, 'Height must be greater or equal to 0.5')
    .required('Height is required'),
});

const BodyMassIndexForm = ({ onSubmit }) => {
  const formik = useFormik({
    initialValues,
    validationSchema,
    onSubmit,
  });

  return (
    <View>
      <TextInput
        placeholder="Weight (kg)"
        value={formik.values.mass}
        onChangeText={formik.handleChange('mass')}
      />
      {formik.touched.mass && formik.errors.mass && (
        <Text style={{ color: 'red' }}>{formik.errors.mass}</Text>
      )}
      <TextInput
        placeholder="Height (m)"
        value={formik.values.height}
        onChangeText={formik.handleChange('height')}
      />
      {formik.touched.height && formik.errors.height && (
        <Text style={{ color: 'red' }}>{formik.errors.height}</Text>
      )}
      <Pressable onPress={formik.handleSubmit}>
        <Text>Calculate</Text>
      </Pressable>
```

```
    </View>
  );
};

const BodyMassIndexCalculator = () => {
  // ...
```

Be aware that you need to include these Text components within the View returned by the form to display the validation errors:

```
{formik.touched.mass && formik.errors.mass && (
 <Text style={{ color: 'red' }}>{formik.errors.mass}</Text>
)}
```

copy

```
{formik.touched.height && formik.errors.height && (
 <Text style={{ color: 'red' }}>{formik.errors.height}</Text>
)}
```

copy

The validation is performed by default every time a field's value changes and when the `handleSubmit` function is called. If the validation fails, the function provided for the `onSubmit` prop of the `Formik` component is not called.

Exercise 10.9

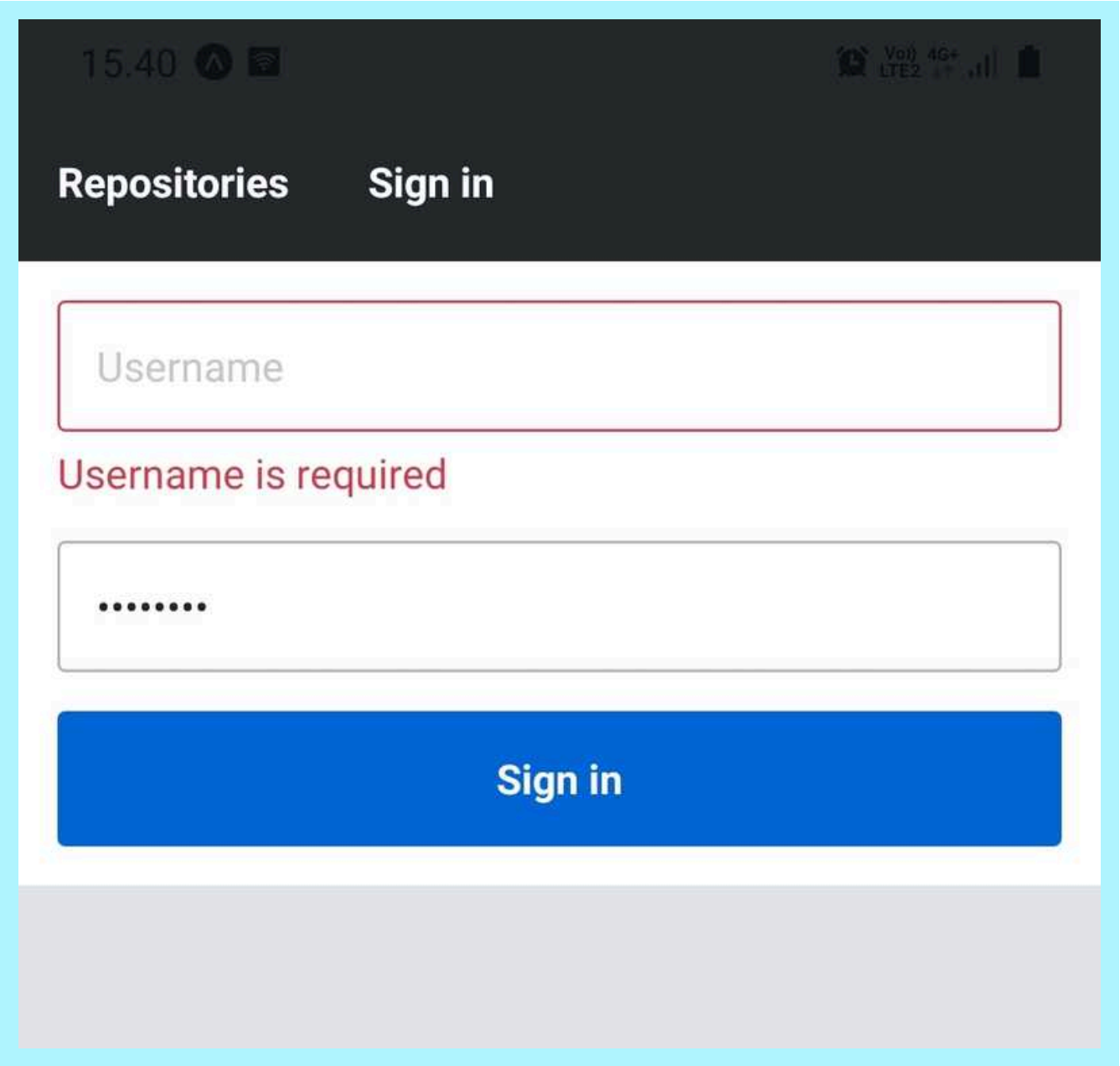
Exercise 10.9: validating the sign-in form

Validate the sign-in form so that both username and password fields are required. Note that the `onSubmit` callback implemented in the previous exercise, *should not be called* if the form validation fails.

The current implementation of the `TextInput` component should display an error message if a touched field has an error. Emphasize this error message by giving it a red color.

On top of the red error message, give an invalid field a visual indication of an error by giving it a red border color. Remember that if a field has an error, the `TextInput` component sets the `TextInput` component's `error` prop as `true` . You can use the value of the `error` prop to attach conditional styles to the `TextInput` component.

Here's what the sign-in form should roughly look like with an invalid field:



The red color used in this implementation is `#d73a4a`.

Platform-specific code

A big benefit of React Native is that we don't need to worry about whether the application is run on an Android or iOS device. However, there might be cases where we need to execute *platform-specific code*. Such cases could be for example using a different implementation of a component on a different platform.

We can access the user's platform through the `Platform.OS` constant:

```
import { React } from 'react';
import { Platform, Text, StyleSheet } from 'react-native';
```

```
const styles = StyleSheet.create({
  text: {
    color: Platform.OS === 'android' ? 'green' : 'blue',
  },
});

const WhatIsMyPlatform = () => {
  return <Text style={styles.text}>Your platform is: {Platform.OS}</Text>;
};
```

Possible values for the `Platform.OS` constants are `android` and `ios`. Another useful way to define platform-specific code branches is to use the `Platform.select` method. Given an object where keys are one of `ios`, `android`, `native` and `default`, the `Platform.select` method returns the most fitting value for the platform the user is currently running on. We can rewrite the `styles` variable in the previous example using the `Platform.select` method like this:

```
const styles = StyleSheet.create({
  text: {
    color: Platform.select({
      android: 'green',
      ios: 'blue',
      default: 'black',
    }),
  },
});
```

We can even use the `Platform.select` method to require a platform-specific component:

```
const MyComponent = Platform.select({
  ios: () => require('./MyIOSComponent'),
  android: () => require('./MyAndroidComponent'),
})();

<MyComponent />;
```

However, a more sophisticated method for implementing and importing platform-specific components (or any other piece of code) is to use the *.ios.jsx* and *.android.jsx* file extensions. Note that the *.jsx* extension can as well be any extension recognized by the bundler, such as *.js*. We can for example have files *Button.ios.jsx* and *Button.android.jsx* which we can import like this:

```
import Button from './Button';

const PlatformSpecificButton = () => {
```

```
  return <Button />;
};
```

Now, the Android bundle of the application will have the component defined in the *Button.android.jsx* whereas the iOS bundle the one defined in the *Button.ios.jsx* file.

Exercise 10.10

Exercise 10.10: a platform-specific font

Currently, the font family of our application is set to *System* in the theme configuration located in the *theme.js* file. Instead of the *System* font, use a platform-specific Sans-serif font. On the Android platform, use the *Roboto* font and on the iOS platform, use the *Arial* font. The default font can be *System*.

This was the last exercise in this section. It's time to push your code to GitHub and mark all of your finished exercises to the exercise submission system. Note that exercises in this section should be submitted to the section named part 2 in the exercise submission system.

Propose changes to material

Part 10a
Previous part

Part 10c
Next part

About course

Course contents

FAQ

Partners

Challenge


UNIVERSITY OF HELSINKI


HOUSTON

{() => fs}

Fullstack > Part 10 > Communicating with server

Communicating with server

Note: This course material was updated in Feb 2024. Some updates are not compatible anymore with older material. We recommend a fresh start with this new Part 10 material. However, if you´re returning to this course after a break, and you want to continue the exercises in your older project, please use Part 10 material before the upgrade.

So far we have implemented features to our application without any actual server communication. For example, the reviewed repositories list we have implemented uses mock data and the sign in form doesn't send the user's credentials to any authentication endpoint. In this section, we will learn how to communicate with a server using HTTP requests, how to use Apollo Client in a React Native application, and how to store data in the user's device.

Soon we will learn how to communicate with a server in our application. Before we get to that, we need a server to communicate with. For this purpose, we have a completed server implementation in the rate-repository-api repository. The rate-repository-api server fulfills all our application's API needs during this part. It uses SQLite database which doesn't need any setup and provides an Apollo GraphQL API along with a few REST API endpoints.

Before heading further into the material, set up the rate-repository-api server by following the setup instructions in the repository's README. Note that if you are using an emulator for development it is recommended to run the server and the emulator *on the same computer*. This eases network requests considerably.

HTTP requests

React Native provides Fetch API for making HTTP requests in our applications. React Native also supports the good old XMLHttpRequest API which makes it possible to use third-party libraries such as

Axios. These APIs are the same as the ones in the browser environment and they are globally available without the need for an import.

People who have used both Fetch API and XMLHttpRequest API most likely agree that the Fetch API is easier to use and more modern. However, this doesn't mean that XMLHttpRequest API doesn't have its uses. For the sake of simplicity, we will be only using the Fetch API in our examples.

Sending HTTP requests using the Fetch API can be done using the `fetch` function. The first argument of the function is the URL of the resource:

```
fetch('https://my-api.com/get-end-point');
```

copy

The default request method is *GET*. The second argument of the `fetch` function is an options object, which you can use to for example to specify a different request method, request headers, or request body:

```
fetch('https://my-api.com/post-end-point', {
  method: 'POST',
  headers: {
    Accept: 'application/json',
    'Content-Type': 'application/json',
  },
  body: JSON.stringify({
    firstParam: 'firstValue',
    secondParam: 'secondValue',
  }),
});
```

copy

Note that these URLs are made up and won't (most likely) send a response to your requests. In comparison to Axios, the Fetch API operates on a bit lower level. For example, there isn't any request or response body serialization and parsing. This means that you have to for example set the *Content-Type* header by yourself and use `JSON.stringify` method to serialize the request body.

The `fetch` function returns a promise which resolves a Response object. Note that error status codes such as 400 and 500 *are not rejected* like for example in Axios. In case of a JSON formatted response we can parse the response body using the `Response.json` method:

```
const fetchMovies = async () => {
  const response = await fetch('https://reactnative.dev/movies.json');
  const json = await response.json();

  return json;
};
```

copy

For a more detailed introduction to the Fetch API, read the Using Fetch article in the MDN web docs.

Next, let's try the Fetch API in practice. The rate-repository-api server provides an endpoint for returning a paginated list of reviewed repositories. Once the server is running, you should be able to access the endpoint at http://localhost:5000/api/repositories (unless you have changed the port). The data is paginated in a common cursor based pagination format. The actual repository data is behind the *node* key in the *edges* array.

Unfortunately, if we´re using external device, we can't access the server directly in our application by using the *http://localhost:5000/api/repositories* URL. To make a request to this endpoint in our application we need to access the server using its IP address in its local network. To find out what it is, open the Expo development tools by running `npm start`. In the console you should be able to see an URL starting with *exp://* below the QR code, after the "Metro waiting on" text:

Copy the IP address between the *exp://* and *:*, which is in this example *192.168.1.33*. Construct an URL in format *http://<IP_ADDRESS>:5000/api/repositories* and open it in the browser. You should see the same response as you did with the *localhost* URL.

Now that we know the end point's URL let's use the actual server-provided data in our reviewed repositories list. We are currently using mock data stored in the `repositories` variable. Remove the `repositories` variable and replace the usage of the mock data with this piece of code in the *RepositoryList.jsx* file in the *components* directory:

```
import { useState, useEffect } from 'react';
// ...

const RepositoryList = () => {
  const [repositories, setRepositories] = useState();

  const fetchRepositories = async () => {
    // Replace the IP address part with your own IP address!
    const response = await fetch('http://192.168.1.33:5000/api/repositories');
    const json = await response.json();
```

```
    console.log(json);

    setRepositories(json);
  };

  useEffect(() => {
    fetchRepositories();
  }, []);

  // Get the nodes from the edges array
  const repositoryNodes = repositories
    ? repositories.edges.map(edge => edge.node)
    : [];

  return (
    <FlatList
      data={repositoryNodes}
      // Other props
    />
  );
};

export default RepositoryList;
```

We are using the React's `useState` hook to maintain the repository list state and the `useEffect` hook to call the `fetchRepositories` function when the `RepositoryList` component is mounted. We extract the actual repositories into the `repositoryNodes` variable and replace the previously used `repositories` variable in the `FlatList` component's `data` prop with it. Now you should be able to see actual server-provided data in the reviewed repositories list.

It is usually a good idea to log the server's response to be able to inspect it as we did in the `fetchRepositories` function. You should be able to see this log message in the Expo development tools if you navigate to your device's logs as we learned in the Debugging section. If you are using the Expo's mobile app for development and the network request is failing, make sure that the computer you are using to run the server and your phone are *connected to the same Wi-Fi network*. If that's not possible either use an emulator in the same computer as the server is running in or set up a tunnel to the localhost, for example, using Ngrok.

The current data fetching code in the RepositoryList component could do with some refactoring. For instance, the component is aware of the network request's details such as the end point's URL. In addition, the data fetching code has lots of reuse potential. Let's refactor the component's code by extract the data fetching code into its own hook. Create a directory *hooks* in the *src* directory and in that *hooks* directory create a file *useRepositories.js* with the following content:

```
import { useState, useEffect } from 'react';

const useRepositories = () => {
  const [repositories, setRepositories] = useState();
  const [loading, setLoading] = useState(false);
```

```
  const fetchRepositories = async () => {
    setLoading(true);

    // Replace the IP address part with your own IP address!
    const response = await fetch('http://192.168.1.33:5000/api/repositories');
    const json = await response.json();

    setLoading(false);
    setRepositories(json);
  };

  useEffect(() => {
    fetchRepositories();
  }, []);

  return { repositories, loading, refetch: fetchRepositories };
};

export default useRepositories;
```

Now that we have a clean abstraction for fetching the reviewed repositories, let's use the `useRepositories` hook in the `RepositoryList` component:

copy

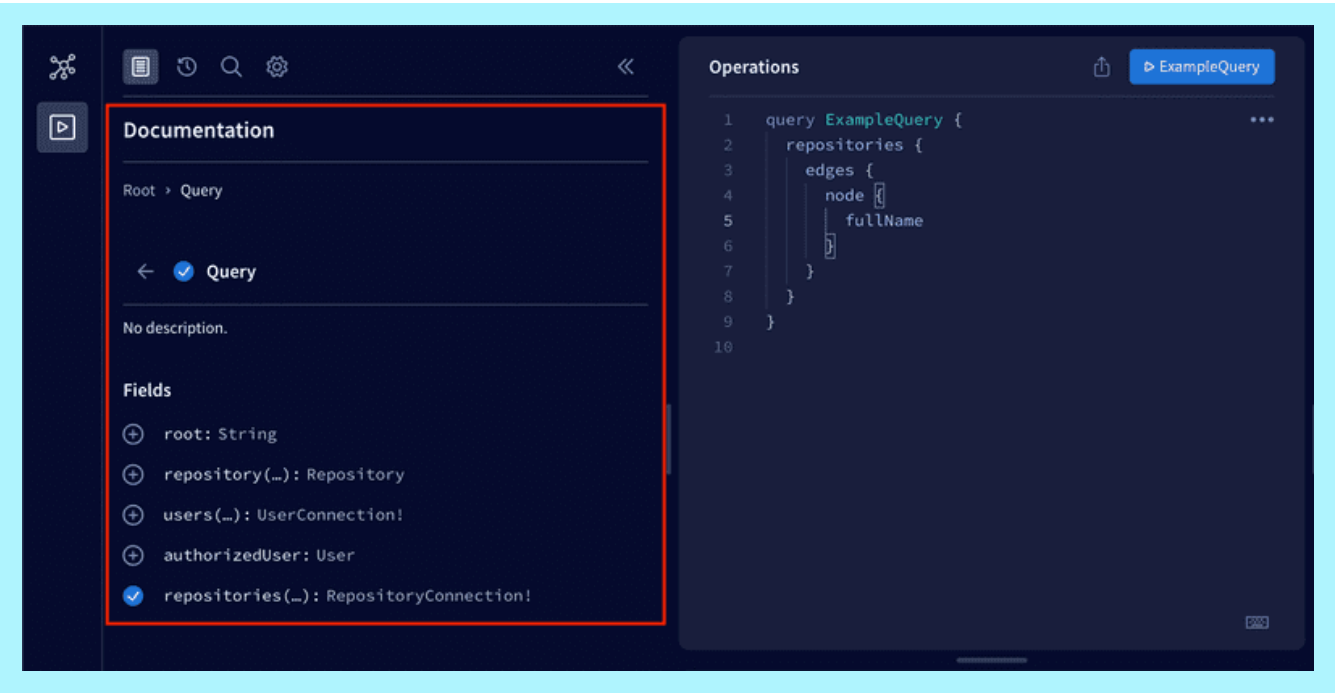
```
// ...
import useRepositories from '../hooks/useRepositories';

const RepositoryList = () => {
  const { repositories } = useRepositories();

  const repositoryNodes = repositories
    ? repositories.edges.map(edge => edge.node)
    : [];

  return (
    <FlatList
      data={repositoryNodes}
      // Other props
    />
  );
};

export default RepositoryList;
```

That's it, now the `RepositoryList` component is no longer aware of the way the repositories are acquired. Maybe in the future, we will acquire them through a GraphQL API instead of a REST API. We will see what happens.

GraphQL and Apollo client

In part 8 we learned about GraphQL and how to send GraphQL queries to an Apollo Server using the Apollo Client in React applications. The good news is that we can use the Apollo Client in a React Native application exactly as we would with a React web application.

As mentioned earlier, the rate-repository-api server provides a GraphQL API which is implemented with Apollo Server. Once the server is running, you can access the Apollo Sandbox at http://localhost:4000. Apollo Sandbox is a tool for making GraphQL queries and inspecting the GraphQL APIs schema and documentation. If you need to send a query in your application *always* test it with the Apollo Sandbox first before implementing it in the code. It is much easier to debug possible problems in the query in the Apollo Sandbox than in the application. If you are uncertain what the available queries are or how to use them, you can see the documentation next to the operations editor:



In our React Native application, we will be using the same @apollo/client library as in part 8. Let's get started by installing the library along with the graphql library which is required as a peer dependency:

```
npm install @apollo/client graphql
```

copy

Before we can start using Apollo Client, we will need to slightly configure the Metro bundler so that it handles the *.cjs* file extensions used by the Apollo Client. First, let's install the *@expo/metro-config* package which has the default Metro configuration:

```
npm install @expo/metro-config@0.17.4
```

copy

Then, we can add the following configuration to a *metro.config.js* in the root directory of our project:

```
const { getDefaultConfig } = require('@expo/metro-config');

const defaultConfig = getDefaultConfig(__dirname);

defaultConfig.resolver.sourceExts.push('cjs');

module.exports = defaultConfig;
```

Restart the Expo development tools so that changes in the configuration are applied.

Now that the Metro configuration is in order, let's create a utility function for creating the Apollo Client with the required configuration. Create a *utils* directory in the *src* directory and in that *utils* directory create a file *apolloClient.js*. In that file configure the Apollo Client to connect to the Apollo Server:

```
import { ApolloClient, InMemoryCache } from '@apollo/client';


const createApolloClient = () => {
  return new ApolloClient({
    uri: 'http://192.168.1.100:4000/graphql',
    cache: new InMemoryCache(),
  });
};

export default createApolloClient;
```

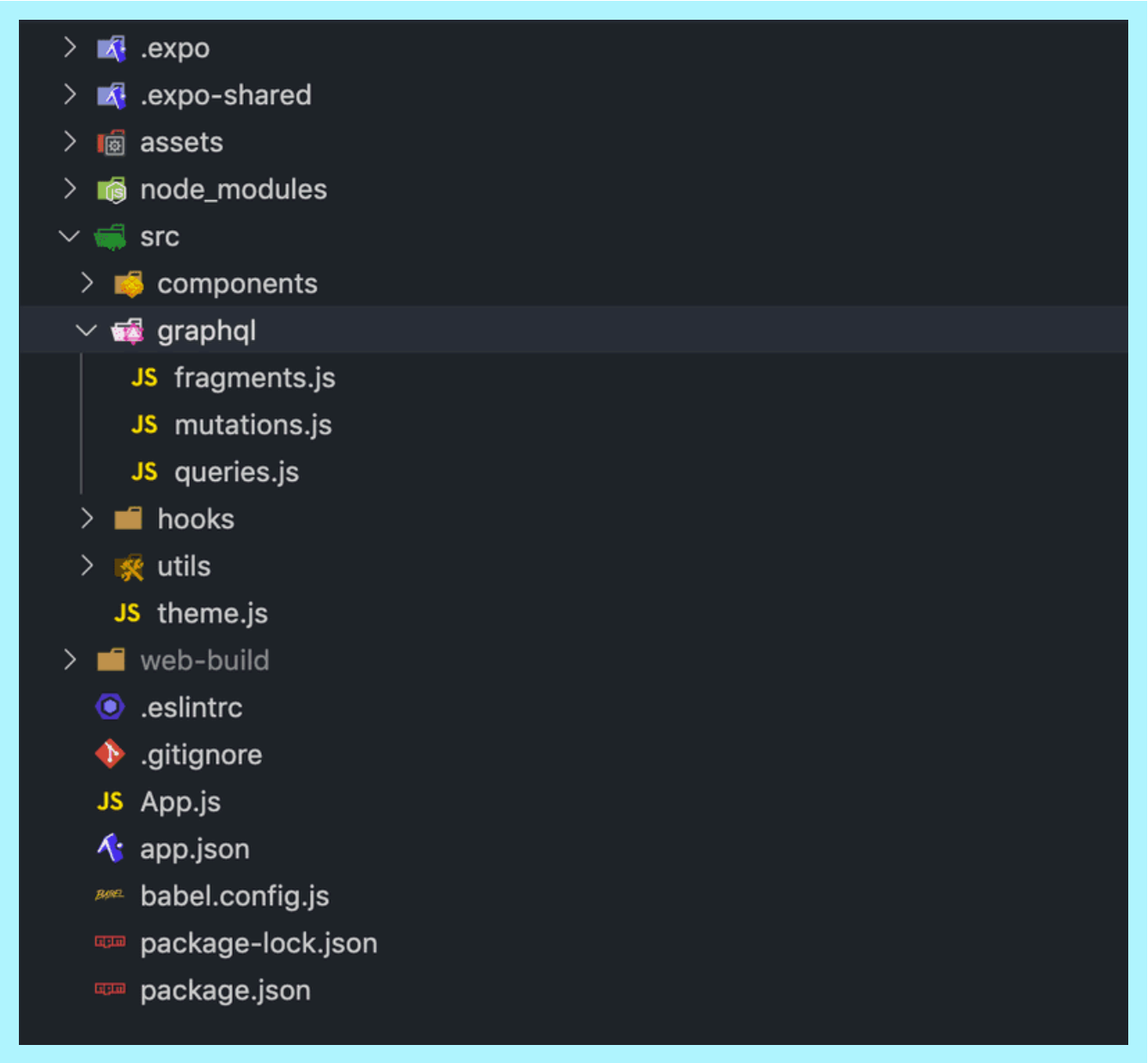
The URL used to connect to the Apollo Server is otherwise the same as the one you used with the Fetch API except the port is *4000* and the path is */graphql*. Lastly, we need to provide the Apollo Client using the `ApolloProvider` context. We will add it to the `App` component in the *App.js* file:

```
import { NativeRouter } from 'react-router-native';
import { ApolloProvider } from '@apollo/client';

import Main from './src/components/Main';
import createApolloClient from './src/utils/apolloClient';

const apolloClient = createApolloClient();

const App = () => {
  return (
    <NativeRouter>
      <ApolloProvider client={apolloClient}>
        <Main />
      </ApolloProvider>
    </NativeRouter>
  );
};
```

```
export default App;
```

Organizing GraphQL related code

It is up to you how to organize the GraphQL related code in your application. However, for the sake of a reference structure, let's have a look at one quite simple and efficient way to organize the GraphQL related code. In this structure, we define queries, mutations, fragments, and possibly other entities in their own files. These files are located in the same directory. Here is an example of the structure you can use to get started:



You can import the `gql` template literal tag used to define GraphQL queries from *@apollo/client* library. If we follow the structure suggested above, we could have a *queries.js* file in the *graphql* directory for our application's GraphQL queries. Each of the queries can be stored in a variable and exported like this:

```
import { gql } from '@apollo/client';

export const GET_REPOSITORIES = gql`
  query {
    repositories {
      ${/* ... */}
    }
  }
`;

// other queries...
```

We can import these variables and use them with the `useQuery` hook like this:

```
import { useQuery } from '@apollo/client';

import { GET_REPOSITORIES } from '../graphql/queries';

const Component = () => {
  const { data, error, loading } = useQuery(GET_REPOSITORIES);
  // ...
};
```

The same goes for organizing mutations. The only difference is that we define them in a different file, *mutations.js*. It is recommended to use fragments in queries to avoid retyping the same fields over and over again.

Evolving the structure

Once our application grows larger there might be times when certain files grow too large to manage. For example, we have component `A` which renders the components `B` and `C`. All these components are defined in a file *A.jsx* in a *components* directory. We would like to extract components `B` and `C` into their own files *B.jsx* and *C.jsx* without major refactors. We have two options:

- Create files *B.jsx* and *C.jsx* in the *components* directory. This results in the following structure:

```
components/
  A.jsx
  B.jsx
  C.jsx
  ...
```

- Create a directory *A* in the *components* directory and create files *B.jsx* and *C.jsx* there. To avoid breaking components that import the *A.jsx* file, move the *A.jsx* file to the *A* directory and rename it to

index.jsx. This results in the following structure:

```
components/
  A/
    B.jsx
    C.jsx
    index.jsx
  ...
```

The first option is fairly decent, however, if components `B` and `C` are not reusable outside the component `A`, it is useless to bloat the *components* directory by adding them as separate files. The second option is quite modular and doesn't break any imports because importing a path such as *./A* will match both *A.jsx* and *A/index.jsx*.

Exercise 10.11

Exercise 10.11: fetching repositories with Apollo Client

We want to replace the Fetch API implementation in the `useRepositories` hook with a GraphQL query. Open the Apollo Sandbox at http://localhost:4000 and take a look at the documentation next to the operations editor. Look up the `repositories` query. The query has some arguments, however, all of these are optional so you don't need to specify them. In the Apollo Sandbox form a query for fetching the repositories with the fields you are currently displaying in the application. The result will be paginated and it contains the up to first 30 results by default. For now, you can ignore the pagination entirely.

Once the query is working in the Apollo Sandbox, use it to replace the Fetch API implementation in the `useRepositories` hook. This can be achieved using the useQuery hook. The `gql` template literal tag can be imported from the *@apollo/client* library as instructed earlier. Consider using the structure recommended earlier for the GraphQL related code. To avoid future caching issues, use the `cache-and-network` fetch policy in the query. It can be used with the `useQuery` hook like this:

```
useQuery(MY_QUERY, {
  fetchPolicy: 'cache-and-network',
  // Other options
});
```

The changes in the `useRepositories` hook should not affect the `RepositoryList` component in any way.

Environment variables

Every application will most likely run in more than one environment. Two obvious candidates for these environments are the development environment and the production environment. Out of these two, the development environment is the one we are running the application right now. Different environments usually have different dependencies, for example, the server we are developing locally might use a local database whereas the server that is deployed to the production environment uses the production database. To make the code environment independent we need to parametrize these dependencies. At the moment we are using one very environment dependant hardcoded value in our application: the URL of the server.

We have previously learned that we can provide running programs with environment variables. These variables can be defined in the command line or using environment configuration files such as *.env* files and third-party libraries such as *Dotenv*. Unfortunately, React Native doesn't have direct support for environment variables. However, we can access the Expo configuration defined in the *app.json* file at runtime from our JavaScript code. This configuration can be used to define and access environment dependant variables.

The configuration can be accessed by importing the `Constants` constant from the *expo-constants* module as we have done a few times before. Once imported, the `Constants.expoConfig` property will contain the configuration. Let's try this by logging `Constants.expoConfig` in the `App` component:

```
import { NativeRouter } from 'react-router-native';
import { ApolloProvider } from '@apollo/client';
import Constants from 'expo-constants';

import Main from './src/components/Main';
import createApolloClient from './src/utils/apolloClient';

const apolloClient = createApolloClient();

const App = () => {
  console.log(Constants.expoConfig);

  return (
    <NativeRouter>
      <ApolloProvider client={apolloClient}>
        <Main />
      </ApolloProvider>
    </NativeRouter>
  );
};

export default App;
```

You should now see the configuration in the logs.

The next step is to use the configuration to define environment dependant variables in our application. Let's get started by renaming the *app.json* file to *app.config.js*. Once the file is renamed, we can use JavaScript inside the configuration file. Change the file contents so that the previous object:

```
{
  "expo": {
    "name": "rate-repository-app",
    // rest of the configuration...
  }
}
```

Is turned into an export, which contains the contents of the `expo` property:

```
export default {
   name: 'rate-repository-app',
   // rest of the configuration...
};
```

Expo has reserved an `extra` property in the configuration for any application-specific configuration. To see how this works, let's add an `env` variable into our application's configuration. Note, that the older versions used (now deprecated) manifest instead of expoConfig.

```
export default {
   name: 'rate-repository-app',
   // rest of the configuration...
   extra: {
     env: 'development'
   },
};
```

If you make changes in configuration, the restart may not be enough. You may need to start the application with cache cleared by command:

```
npx expo start --clear
```

Now, restart Expo development tools to apply the changes and you should see that the value of `Constants.expoConfig` property has changed and now includes the `extra` property containing our application-specific configuration. Now the value of the `env` variable is accessible through the `Constants.expoConfig.extra.env` property.

Because using hard coded configuration is a bit silly, let's use an environment variable instead:

```
export default {
   name: 'rate-repository-app',
   // rest of the configuration...
   extra: {
     env: process.env.ENV,
   },
};
```

As we have learned, we can set the value of an environment variable through the command line by defining the variable's name and value before the actual command. As an example, start Expo development tools and set the environment variable `ENV` as `test` like this:

```
ENV=test npm start
```

If you take a look at the logs, you should see that the `Constants.expoConfig.extra.env` property has changed.

We can also load environment variables from an `.env` file as we have learned in the previous parts. First, we need to install the Dotenv library:

```
npm install dotenv
```

Next, add a `.env` file in the root directory of our project with the following content:

```
ENV=development
```

Finally, import the library in the *app.config.js* file:

```
import 'dotenv/config';

export default {
   name: 'rate-repository-app',
   // rest of the configuration...
   extra: {
     env: process.env.ENV,
   },
};
```

You need to restart Expo development tools to apply the changes you have made to the *.env* file.

Note that it is *never* a good idea to put sensitive data into the application's configuration. The reason for this is that once a user has downloaded your application, they can, at least in theory, reverse engineer your application and figure out the sensitive data you have stored into the code.

Exercise 10.12

Exercise 10.12: environment variables

Instead of the hardcoded Apollo Server's URL, use an environment variable defined in the *.env* file when initializing the Apollo Client. You can name the environment variable for example `APOLLO_URI`.

Do not try to access environment variables like `process.env.APOLLO_URI` outside the *app.config.js* file. Instead use the `Constants.expoConfig.extra` object like in the previous example. In addition, do not import the dotenv library outside the *app.config.js* file or you will most likely face errors.

Storing data in the user's device

There are times when we need to store some persisted pieces of data in the user's device. One such common scenario is storing the user's authentication token so that we can retrieve it even if the user closes and reopens our application. In web development, we have used the browser's `localStorage` object to achieve such functionality. React Native provides similar persistent storage, the AsyncStorage.

We can use the `npx expo install` command to install the version of the *@react-native-async-storage/async-storage* package that is suitable for our Expo SDK version:

```
npx expo install @react-native-async-storage/async-storage
```

copy

The API of the `AsyncStorage` is in many ways same as the `localStorage` API. They are both key-value storages with similar methods. The biggest difference between the two is that, as the name implies, the operations of `AsyncStorage` are *asynchronous*.

Because `AsyncStorage` operates with string keys in a global namespace it is a good idea to create a simple abstraction for its operations. This abstraction can be implemented for example using a class. As an example, we could implement a shopping cart storage for storing the products user wants to buy:

```
import AsyncStorage from '@react-native-async-storage/async-storage';

class ShoppingCartStorage {
  constructor(namespace = 'shoppingCart') {
    this.namespace = namespace;
```

copy

```
  }

  async getProducts() {
    const rawProducts = await AsyncStorage.getItem(
      `${this.namespace}:products`,
    );

    return rawProducts ? JSON.parse(rawProducts) : [];
  }

  async addProduct(productId) {
    const currentProducts = await this.getProducts();
    const newProducts = [...currentProducts, productId];

    await AsyncStorage.setItem(
      `${this.namespace}:products`,
      JSON.stringify(newProducts),
    );
  }

  async clearProducts() {
    await AsyncStorage.removeItem(`${this.namespace}:products`);
  }
}

const doShopping = async () => {
  const shoppingCartA = new ShoppingCartStorage('shoppingCartA');
  const shoppingCartB = new ShoppingCartStorage('shoppingCartB');

  await shoppingCartA.addProduct('chips');
  await shoppingCartA.addProduct('soda');

  await shoppingCartB.addProduct('milk');

  const productsA = await shoppingCartA.getProducts();
  const productsB = await shoppingCartB.getProducts();

  console.log(productsA, productsB);

  await shoppingCartA.clearProducts();
  await shoppingCartB.clearProducts();
};

doShopping();
```

Because `AsyncStorage` keys are global, it is usually a good idea to add a *namespace* for the keys. In this context, the namespace is just a prefix we provide for the storage abstraction's keys. Using the namespace prevents the storage's keys from colliding with other `AsyncStorage` keys. In this example, the namespace is defined as the constructor's argument and we are using the `namespace:key` format for the keys.

We can add an item to the storage using the AsyncStorage.setItem method. The first argument of the method is the item's key and the second argument its value. The value *must be a string*, so we need to

serialize non-string values as we did with the `JSON.stringify` method. The AsyncStorage.getItem method can be used to get an item from the storage. The argument of the method is the item's key, of which value will be resolved. The AsyncStorage.removeItem method can be used to remove the item with the provided key from the storage.

NB: SecureStore is similar persisted storage as the `AsyncStorage` but it encrypts the stored data. This makes it more suitable for storing more sensitive data such as the user's credit card number.

Exercises 10.13. - 10.14

Exercise 10.13: the sign in form mutation

The current implementation of the sign in form doesn't do much with the submitted user's credentials. Let's do something about that in this exercise. First, read the rate-repository-api server's authentication documentation and test the provided queries and mutations in the Apollo Sandbox. If the database doesn't have any users, you can populate the database with some seed data. Instructions for this can be found in the getting started section of the README.

Once you have figured out how the authentication works, create a file `useSignIn.js` file in the *hooks* directory. In that file implement a `useSignIn` hook that sends the `authenticate` mutation using the useMutation hook. Note that the `authenticate` mutation has a *single* argument called `credentials`, which is of type `AuthenticateInput`. This input type contains `username` and `password` fields.

The return value of the hook should be a tuple `[signIn, result]` where `result` is the mutations result as it is returned by the `useMutation` hook and `signIn` a function that runs the mutation with a `{ username, password }` object argument. Hint: don't pass the mutation function to the return value directly. Instead, return a function that calls the mutation function like this:

```
const useSignIn = () => {
  const [mutate, result] = useMutation(/* mutation arguments */);

  const signIn = async ({ username, password }) => {
    // call the mutate function here with the right arguments
  };

  return [signIn, result];
};
```

Once the hook is implemented, use it in the `SignIn` component's `onSubmit` callback for example like this:

```
const SignIn = () => {
  const [signIn] = useSignIn();

  const onSubmit = async (values) => {
    const { username, password } = values;

    try {
      const { data } = await signIn({ username, password });
      console.log(data);
    } catch (e) {
      console.log(e);
    }
  };

  // ...
};
```

This exercise is completed once you can log the user's *authenticate* mutations result after the sign in form has been submitted. The mutation result should contain the user's access token.

Exercise 10.14: storing the access token step1

Now that we can obtain the access token we need to store it. Create a file *authStorage.js* in the *utils* directory with the following content:

```
import AsyncStorage from '@react-native-async-storage/async-storage';

class AuthStorage {
  constructor(namespace = 'auth') {
    this.namespace = namespace;
  }

  getAccessToken() {
    // Get the access token for the storage
  }

  setAccessToken(accessToken) {
    // Add the access token to the storage
  }

  removeAccessToken() {
    // Remove the access token from the storage
  }
}

export default AuthStorage;
```

Next, implement the methods `AuthStorage.getAccessToken`, `AuthStorage.setAccessToken` and `AuthStorage.removeAccessToken`. Use the `namespace` variable to give your keys a namespace like we did in the previous example.

Enhancing Apollo Client's requests

Now that we have implemented storage for storing the user's access token, it is time to start using it. Initialize the storage in the `App` component:

```
import { NativeRouter } from 'react-router-native';
import { ApolloProvider } from '@apollo/client';

import Main from './src/components/Main';
import createApolloClient from './src/utils/apolloClient';
import AuthStorage from './src/utils/authStorage';

const authStorage = new AuthStorage();
const apolloClient = createApolloClient(authStorage);

const App = () => {
  return (
    <NativeRouter>
      <ApolloProvider client={apolloClient}>
        <Main />
      </ApolloProvider>
    </NativeRouter>
  );
};

export default App;
```

We also provided the storage instance for the `createApolloClient` function as an argument. This is because next, we will send the access token to Apollo Server in each request. The Apollo Server will expect that the access token is present in the *Authorization* header in the format *Bearer <ACCESS_TOKEN>*. We can enhance the Apollo Client's request by using the setContext function. Let's send the access token to the Apollo Server by modifying the `createApolloClient` function in the *apolloClient.js* file:

```
import { ApolloClient, InMemoryCache, createHttpLink } from '@apollo/client';
import Constants from 'expo-constants';
import { setContext } from '@apollo/client/link/context';

// You might need to change this depending on how you have configured the Apollo Server's
URI
const { apolloUri } = Constants.expoConfig.extra;
```

```
const httpLink = createHttpLink({
  uri: apolloUri,
});

const createApolloClient = (authStorage) => {
  const authLink = setContext(async (_, { headers }) => {
    try {
      const accessToken = await authStorage.getAccessToken();
      return {
        headers: {
          ...headers,
          authorization: accessToken ? `Bearer ${accessToken}` : '',
        },
      };
    } catch (e) {
      console.log(e);
      return {
        headers,
      };
    }
  });
  return new ApolloClient({
    link: authLink.concat(httpLink),
    cache: new InMemoryCache(),
  });
};

export default createApolloClient;
```

Using React Context for dependency injection

The last piece of the sign-in puzzle is to integrate the storage to the `useSignIn` hook. To achieve this the hook must be able to access token storage instance we have initialized in the `App` component. React Context is just the tool we need for the job. Create a directory *contexts* in the *src* directory. In that directory create a file *AuthStorageContext.js* with the following content:

```
import { createContext } from 'react';

const AuthStorageContext = createContext();

export default AuthStorageContext;
```

Now we can use the `AuthStorageContext.Provider` to provide the storage instance to the descendants of the context. Let's add it to the `App` component:

```
import { NativeRouter } from 'react-router-native';
import { ApolloProvider } from '@apollo/client';
```

```
import Main from './src/components/Main';
import createApolloClient from './src/utils/apolloClient';
import AuthStorage from './src/utils/authStorage';
import AuthStorageContext from './src/contexts/AuthStorageContext';

const authStorage = new AuthStorage();
const apolloClient = createApolloClient(authStorage);

const App = () => {
  return (
    <NativeRouter>
      <ApolloProvider client={apolloClient}>
        <AuthStorageContext.Provider value={authStorage}>
          <Main />
        </AuthStorageContext.Provider>
      </ApolloProvider>
    </NativeRouter>
  );
};

export default App;
```

Accessing the storage instance in the `useSignIn` hook is now possible using the React's useContext hook like this:

```
// ...
import { useContext } from 'react';

import AuthStorageContext from '../contexts/AuthStorageContext';

const useSignIn = () => {
  const authStorage = useContext(AuthStorageContext);
  // ...
};
```

Note that accessing a context's value using the `useContext` hook only works if the `useContext` hook is used in a component that is a *descendant* of the Context.Provider component.

Accessing the `AuthStorage` instance with `useContext(AuthStorageContext)` is quite verbose and reveals the details of the implementation. Let's improve this by implementing a `useAuthStorage` hook in a *useAuthStorage.js* file in the *hooks* directory:

```
import { useContext } from 'react';
import AuthStorageContext from '../contexts/AuthStorageContext';

const useAuthStorage = () => {
  return useContext(AuthStorageContext);
};
```

```
export default useAuthStorage;
```

The hook's implementation is quite simple but it improves the readability and maintainability of the hooks and components using it. We can use the hook to refactor the `useSignIn` hook like this:

```
// ...
import useAuthStorage from '../hooks/useAuthStorage';

const useSignIn = () => {
  const authStorage = useAuthStorage();
  // ...
};
```

The ability to provide data to component's descendants opens tons of use cases for React Context, as we already saw in the last chapter of part 6.

To learn more about these use cases, read Kent C. Dodds' enlightening article How to use React Context effectively to find out how to combine the useReducer hook with the context to implement state management. You might find a way to use this knowledge in the upcoming exercises.

Exercises 10.15. - 10.16

Exercise 10.15: storing the access token step2

Improve the `useSignIn` hook so that it stores the user's access token retrieved from the *authenticate* mutation. The return value of the hook should not change. The only change you should make to the `SignIn` component is that you should redirect the user to the reviewed repositories list view after a successful sign in. You can achieve this by using the useNavigate hook.

After the *authenticate* mutation has been executed and you have stored the user's access token to the storage, you should reset the Apollo Client's store. This will clear the Apollo Client's cache and re-execute all active queries. You can do this by using the Apollo Client's resetStore method. You can access the Apollo Client in the `useSignIn` hook using the useApolloClient hook. Note that the order of the execution is crucial and should be the following:

```
const { data } = await mutate(/* options */);
await authStorage.setAccessToken(/* access token from the data */);
apolloClient.resetStore();
```

Exercise 10.16: sign out

The final step in completing the sign in feature is to implement a sign out feature. The `me` query can be used to check the authenticated user's information. If the query's result is `null`, that means that the user is not authenticated. Open the Apollo Sandbox and run the following query:

```
{
  me {
    id
    username
  }
}
```

copy

You will probably end up with the `null` result. This is because the Apollo Sandbox is not authenticated, meaning that it doesn't send a valid access token with the request. Revise the authentication documentation and retrieve an access token using the `authenticate` mutation. Use this access token in the `Authorization` header as instructed in the documentation. Now, run the `me` query again and you should be able to see the authenticated user's information.

Open the `AppBar` component in the *AppBar.jsx* file where you currently have the tabs "Repositories" and "Sign in". Change the tabs so that if the user is signed in the tab "Sign out" is displayed, otherwise show the "Sign in" tab. You can achieve this by using the `me` query with the useQuery hook.

Pressing the "Sign out" tab should remove the user's access token from the storage and reset the Apollo Client's store with the resetStore method. Calling the `resetStore` method should automatically re-execute all active queries which means that the `me` query should be re-executed. Note that the order of execution is crucial: access token must be removed from the storage *before* the Apollo Client's store is reset.

This was the last exercise in this section. It's time to push your code to GitHub and mark all of your finished exercises to the exercise submission system. Note that exercises in this section should be submitted to the part 3 in the exercise submission system.

Propose changes to material

Part 10b
Previous part

Part 10d
Next part

About course

Course contents

FAQ

Partners

Challenge


HOUSTON

{() => fs}

Fullstack > Part 10 > Testing and extending our application

Testing and extending our application

Now that we have established a good foundation for our project, it is time to start expanding it. In this section you can put to use all the React Native knowledge you have gained so far. Along with expanding our application we will cover some new areas, such as testing, and additional resources.

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 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 preset. In order to use ESLint in the Jest's test files, we also need the eslint-plugin-jest plugin for ESLint. Let's get started by installing the packages:

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

copy

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": {
```

copy

```
    // 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 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.

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.test.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.test.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 for Jest to find your test files you either have to put them into a *__tests__* directory, use the *.test* or *.spec* suffix, or manually configure 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 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 we got familiar with one of these libraries, the React Testing Library. 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. 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 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
```

copy

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';
```

copy

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"
    },
```

copy

```
    "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 and firing events. Queries are used to extract a set of nodes from the component that is rendered using the 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 query to check if the component's `Text` element has the correct textual content:

```
import { Text, View } from 'react-native';
import { render, screen } 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', () => {
    render(<Greeting name="Kalle" />);

    screen.debug();

    expect(screen.getByText('Hello Kalle!')).toBeDefined();
  });
});
```

Tests use the object screen to do the queries to the rendered component.

We acquire the `Text` node containing certain text by using the `getByText` function. The Jest matcher toBeDefined is used to ensure that the query has found the element.

React Native Testing Library's documentation has some good hints on how to query different kinds of elements. Another guide worth reading is Kent C. Dodds article Making your UI tests resilient to change.

The object screen also has a helper method debug that 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.

For all available queries, check the React Native Testing Library's documentation. The full list of available React Native specific matchers can be found in the documentation of the jest-native library. Jest's documentation 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 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, screen } 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();
    render(<Form onSubmit={onSubmit} />);

    fireEvent.changeText(screen.getByPlaceholderText('Username'), 'kalle');
    fireEvent.changeText(screen.getByPlaceholderText('Password'), 'password');
```

```
    fireEvent.press(screen.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. 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.

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. 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.

Exercises 10.17. - 10.18

Exercise 10.17: testing the reviewed repositories list

Implement a test that ensures that the `RepositoryListContainer` component renders repository's name, description, language, forks count, stargazers count, rating average, and review count correctly. One approach in implementing this test is to add a testID prop for the element wrapping a single repository's information:

```
const RepositoryItem = (/* ... */) => {
  // ...

  return (
    <View testID="repositoryItem" {/* ... */}>
      {/* ... */}
    </View>
  )
};
```

Once the `testID` prop is added, you can use the getAllByTestId query to get those elements:

```
const repositoryItems = screen.getAllByTestId('repositoryItem');
const [firstRepositoryItem, secondRepositoryItem] = repositoryItems;

// expect something from the first and the second repository item
```

Having those elements you can use the toHaveTextContent matcher to check whether an element has certain textual content. You might also find the Querying Within Elements guide useful. If you are unsure what is being rendered, use the debug function to see the serialized rendering result.

Use this as a base for your test:

```
describe('RepositoryList', () => {
  describe('RepositoryListContainer', () => {
    it('renders repository information correctly', () => {
      const repositories = {
        totalCount: 8,
        pageInfo: {
          hasNextPage: true,
          endCursor:
            'WyJhc3luYy1saWJyYXJ5LnJlYWN0LWFzeW5jIiwxNTg4NjU2NzUwMDc2XQ==',
          startCursor: 'WyJqYXJlZHBhbG1lci5mb3JtaWsiLDE1ODg2NjAzNTAwNzZd',
        },
        edges: [
          {
            node: {
              id: 'jaredpalmer.formik',
              fullName: 'jaredpalmer/formik',
              description: 'Build forms in React, without the tears',
              language: 'TypeScript',
              forksCount: 1619,
              stargazersCount: 21856,
              ratingAverage: 88,
              reviewCount: 3,
              ownerAvatarUrl:
                'https://avatars2.githubusercontent.com/u/4060187?v=4',
            },
```

```
            cursor: 'WyJqYXJlZHBhbG1lci5mb3JtaWsiLDE1ODg2NjAzNTAwNzZd',
          },
          {
            node: {
              id: 'async-library.react-async',
              fullName: 'async-library/react-async',
              description: 'Flexible promise-based React data loader',
              language: 'JavaScript',
              forksCount: 69,
              stargazersCount: 1760,
              ratingAverage: 72,
              reviewCount: 3,
              ownerAvatarUrl:
                'https://avatars1.githubusercontent.com/u/54310907?v=4',
            },
            cursor:
              'WyJhc3luYy1saWJyYXJ5LnJlYWN0LWFzeW5jIiwxNTg4NjU2NzUwMDc2XQ==',
          },
        ],
      };

      // Add your test code here
    });
  });
});
```

You can put the test file where you please. However, it is recommended to follow one of the ways of organizing test files introduced earlier. Use the `repositories` variable as the repository data for the test. There should be no need to alter the variable's value. Note that the repository data contains two repositories, which means that you need to check that both repositories' information is present.

Exercise 10.18: testing the sign in form

Implement a test that ensures that filling the sign in form's username and password fields and pressing the submit button *will call* the `onSubmit` handler with *correct arguments*. The *first argument* of the handler should be an object representing the form's values. You can ignore the other arguments of the function. Remember that the fireEvent methods can be used for triggering events and a mock function for checking whether the `onSubmit` handler is called or not.

You don't have to test any Apollo Client or AsyncStorage related code which is in the `useSignIn` hook. As in the previous exercise, extract the pure code into its own component and test it in the test.

Note that Formik's form submissions are *asynchronous* so expecting the `onSubmit` function to be called immediately after pressing the submit button won't work. You can get around this issue by making the test function an async function using the `async` keyword and using the React Native Testing Library's waitFor helper function. The `waitFor` function can be used to wait for expectations to pass. If the expectations don't pass within a certain period, the function will throw an error. Here is a rough example of how to use it:

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

```
describe('SignIn', () => {
  describe('SignInContainer', () => {
    it('calls onSubmit function with correct arguments when a valid form is submitted',
async () => {
      // render the SignInContainer component, fill the text inputs and press the submit
button

      await waitFor(() => {
        // expect the onSubmit function to have been called once and with a correct first
argument
      });
    });
  });
});
```

Extending our application

It is time to put everything we have learned so far to good use and start extending our application. Our application still lacks a few important features such as reviewing a repository and registering a user. The upcoming exercises will focus on these essential features.

Exercises 10.19. - 10.26

Exercise 10.19: the single repository view

Implement a view for a single repository, which contains the same repository information as in the reviewed repositories list but also a button for opening the repository in GitHub. It would be a good idea to reuse the `RepositoryItem` component used in the `RepositoryList` component and display the GitHub repository button for example based on a boolean prop.

The repository's URL is in the `url` field of the `Repository` type in the GraphQL schema. You can fetch a single repository from the Apollo server with the `repository` query. The query has a single argument, which is the id of the repository. Here's a simple example of the `repository` query:

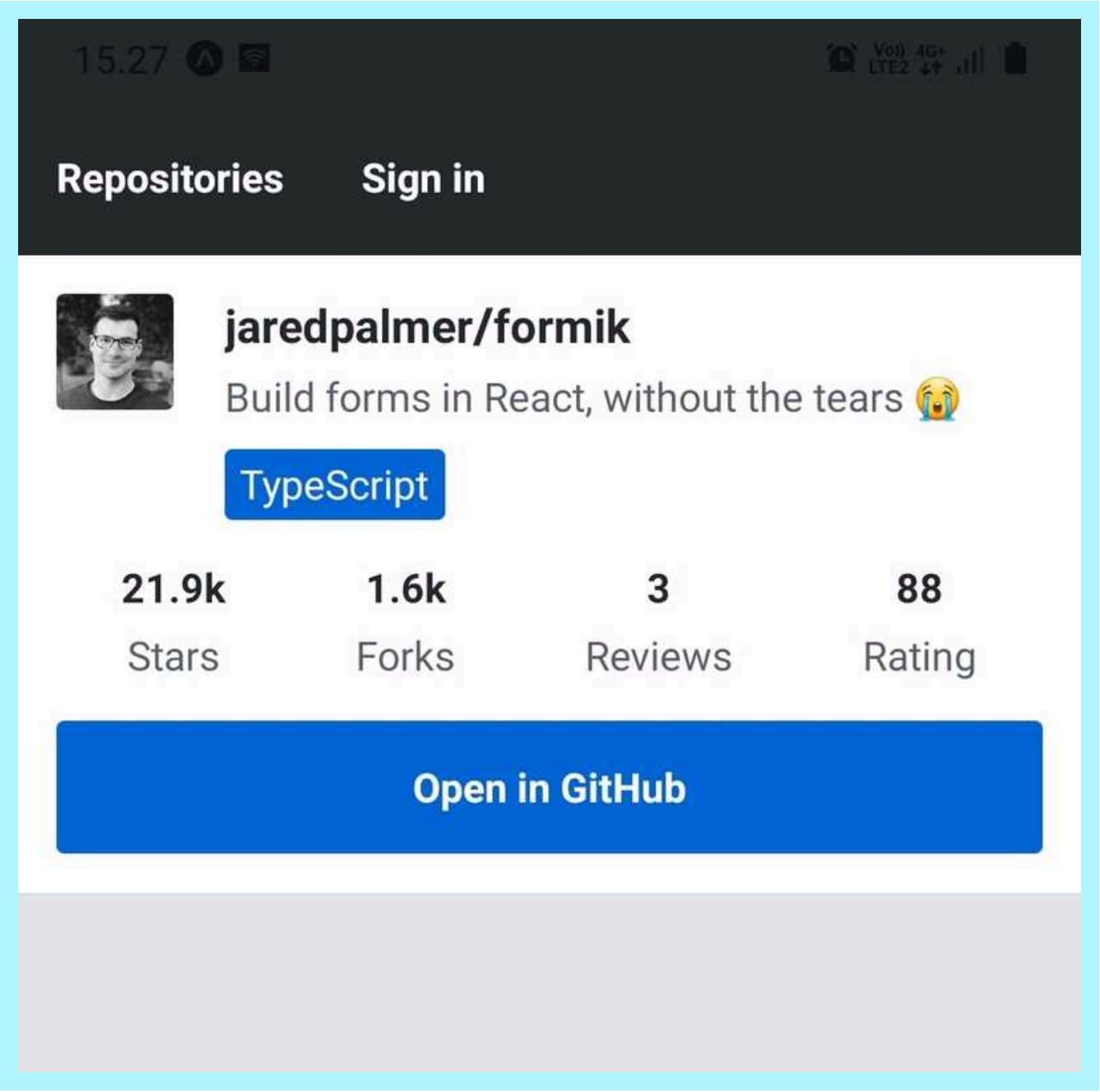
```
{
  repository(id: "jaredpalmer.formik") {
    id
    fullName
    url
  }
}
```

As always, test your queries in the Apollo Sandbox first before using them in your application. If you are unsure about the GraphQL schema or what are the available queries, take a look at the documentation next to the operations editor. If you have trouble using the id as a variable in the query, take a moment to study the Apollo Client's documentation on queries.

To learn how to open a URL in a browser, read the Expo's Linking API documentation. You will need this feature while implementing the button for opening the repository in GitHub. Hint: Linking.openURL method will come in handy.

The view should have its own route. It would be a good idea to define the repository's id in the route's path as a path parameter, which you can access by using the useParams hook. The user should be able to access the view by pressing a repository in the reviewed repositories list. You can achieve this by for example wrapping the `RepositoryItem` with a Pressable component in the `RepositoryList` component and using `navigate` function to change the route in an `onPress` event handler. You can access the `navigate` function with the useNavigate hook.

The final version of the single repository view should look something like this:



Note if the peer depencendy issues prevent installing the library, try the `--legacy-peer-deps` option:

```
npm install expo-linking --legacy-peer-deps
```

copy

Exercise 10.20: repository's review list

Now that we have a view for a single repository, let's display repository's reviews there. Repository's reviews are in the `reviews` field of the `Repository` type in the GraphQL schema. `reviews` is a similar paginated list as in the `repositories` query. Here's an example of getting reviews of a repository:

```
{
  repository(id: "jaredpalmer.formik") {
    id
    fullName
    reviews {
      edges {
        node {
          id
          text
          rating
          createdAt
          user {
            id
            username
          }
        }
      }
    }
  }
}
```

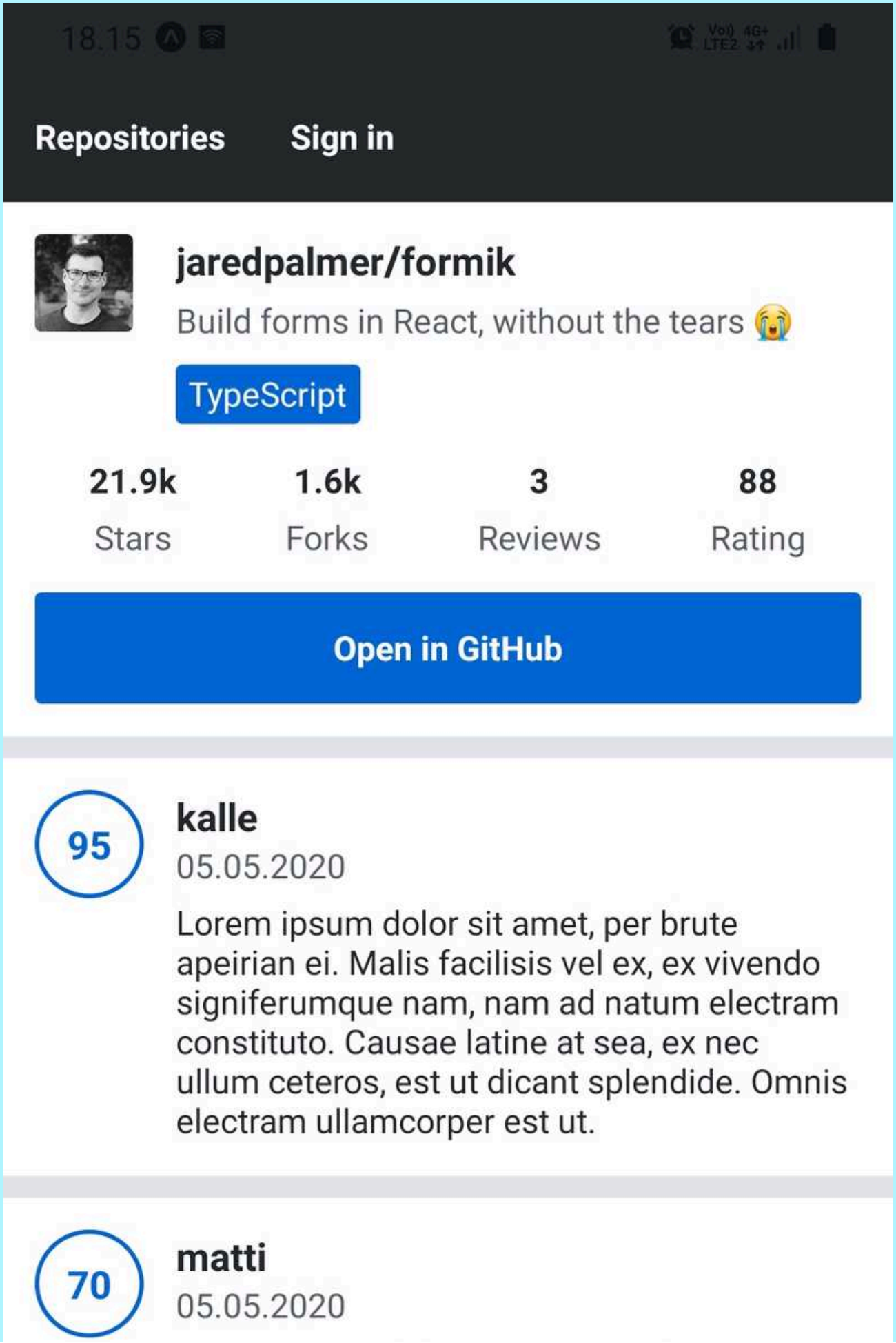
Review's `text` field contains the textual review, `rating` field a numeric rating between 0 and 100, and `createdAt` the date when the review was created. Review's `user` field contains the reviewer's information, which is of type `User`.

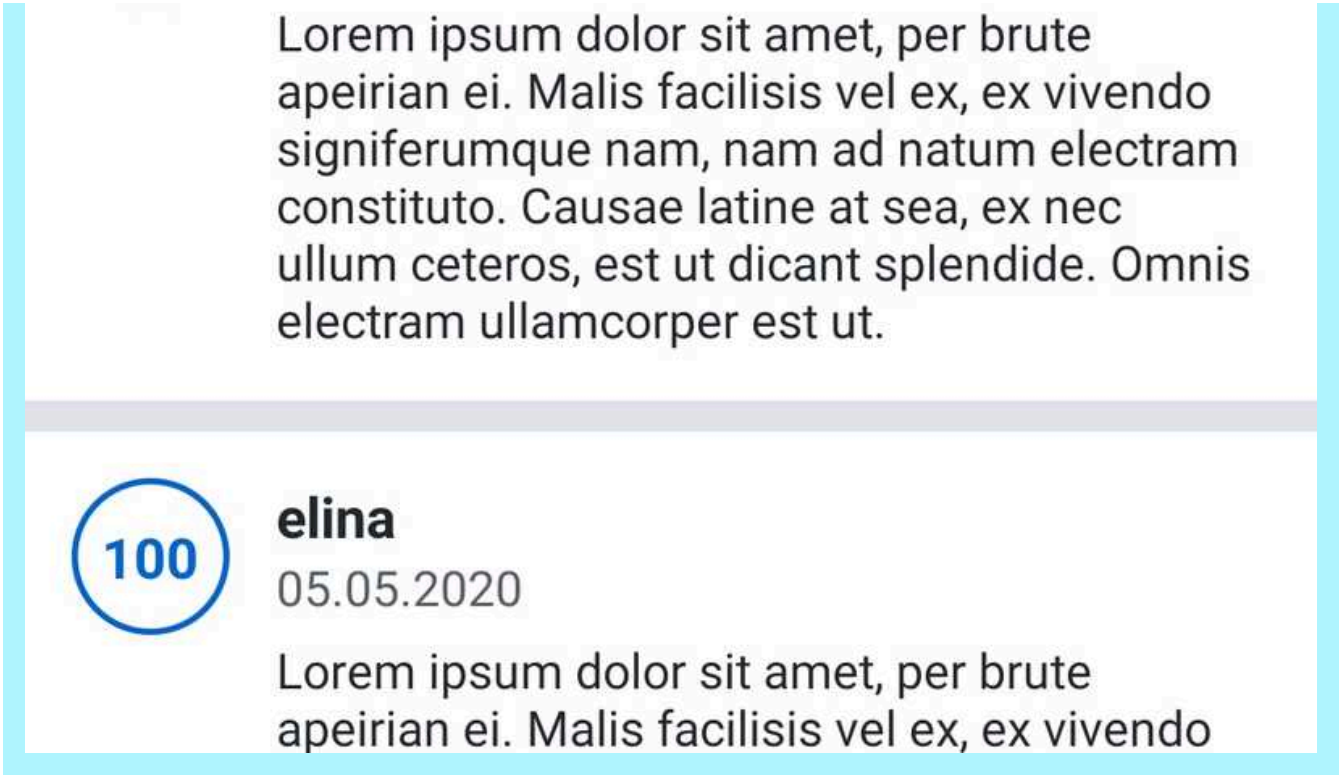
We want to display reviews as a scrollable list, which makes FlatList a suitable component for the job. To display the previous exercise's repository's information at the top of the list, you can use the `FlatList` component's ListHeaderComponent prop. You can use the ItemSeparatorComponent to add some space between the items like in the `RepositoryList` component. Here's an example of the structure:

```
const RepositoryInfo = ({ repository }) => {
  // Repository's information implemented in the previous exercise
};

const ReviewItem = ({ review }) => {
  // Single review item
};

const SingleRepository = () => {
  // ...

  return (
    <FlatList
      data={reviews}
      renderItem={({ item }) => <ReviewItem review={item} />}
      keyExtractor={({ id }) => id}
      ListHeaderComponent={() => <RepositoryInfo repository={repository} />}
      // ...
    />
```

```
    );
};

export default SingleRepository;
```

The final version of the repository's reviews list should look something like this:


18.15
Repositories
Sign in
jaredpalmer/formik
Build forms in React, without the tears 😭
TypeScript
21.9k
Stars
1.6k
Forks
3
Reviews
88
Rating
Open in GitHub
95
kalle
05.05.2020
Lorem ipsum dolor sit amet, per brute apeirian ei. Malis facilisis vel ex, ex vivendo signiferumque nam, nam ad natum electram constituto. Causae latine at sea, ex nec ullum ceteros, est ut dicant splendide. Omnis electram ullamcorper est ut.
70
matti
05.05.2020



The date under the reviewer's username is the creation date of the review, which is in the `createdAt` field of the `Review` type. The date format should be user-friendly such as *date.month.year*. You can for example install the date-fns library and use the format function for formatting the creation date.

The round shape of the rating's container can be achieved with the `borderRadius` style property. You can make it round by fixing the container's `width` and `height` style property and setting the border-radius as `width / 2`.

Exercise 10.21: the review form

Implement a form for creating a review using Formik. The form should have four fields: repository owner's GitHub username (for example "jaredpalmer"), repository's name (for example "formik"), a numeric rating, and a textual review. Validate the fields using Yup schema so that it contains the following validations:

- Repository owner's username is a required string
- Repository's name is a required string
- Rating is a required number between 0 and 100
- Review is a optional string

Explore Yup's documentation to find suitable validators. Use sensible error messages with the validators. The validation message can be defined as the validator method's `message` argument. You can make the review field expand to multiple lines by using `TextInput` component's multiline prop.

You can create a review using the `createReview` mutation. Check this mutation's arguments in the Apollo Sandbox. You can use the useMutation hook to send a mutation to the Apollo Server.

After a successful `createReview` mutation, redirect the user to the repository's view you implemented in the previous exercise. This can be done with the `navigate` function after you have obtained it using the useNavigate hook. The created review has a `repositoryId` field which you can use to construct the route's path.

To prevent getting cached data with the `repository` query in the single repository view, use the `cache-and-network` fetch policy in the query. It can be used with the `useQuery` hook like this:

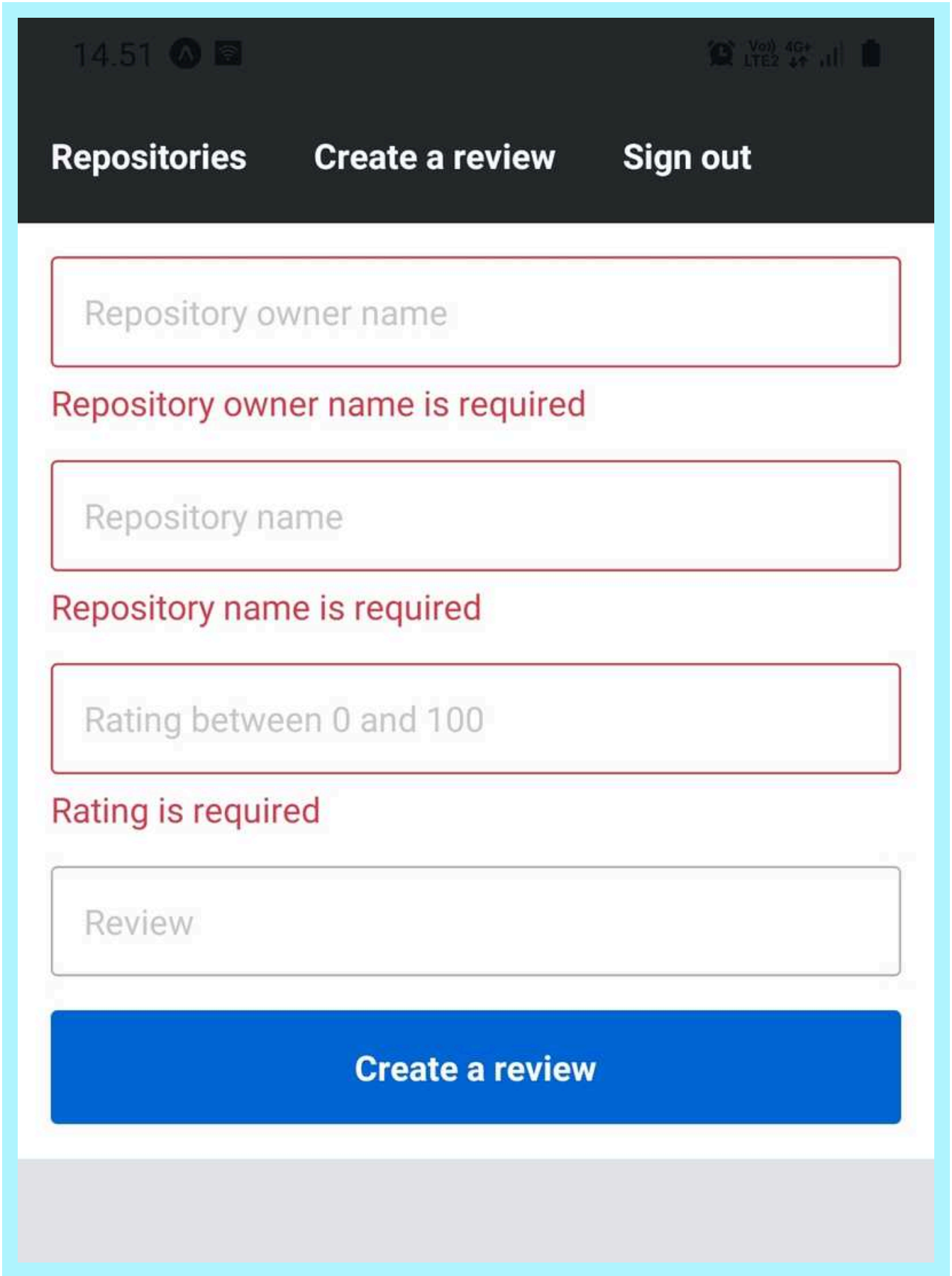
```
useQuery(GET_REPOSITORY, {
  fetchPolicy: 'cache-and-network',
  // Other options
});
```

copy

Note that only *an existing public GitHub repository* can be reviewed and a user can review the same repository *only once*. You don't have to handle these error cases, but the error payload includes specific codes and messages for these errors. You can try out your implementation by reviewing one of your own public repositories or any other public repository.

The review form should be accessible through the app bar. Create a tab to the app bar with a label "Create a review". This tab should only be visible to users who have signed in. You will also need to define a route for the review form.

The final version of the review form should look something like this:



This screenshot has been taken after invalid form submission to present what the form should look like in an invalid state.

Exercise 10.22: the sign up form

Implement a sign up form for registering a user using Formik. The form should have three fields: username, password, and password confirmation. Validate the form using Yup schema so that it contains the following validations:

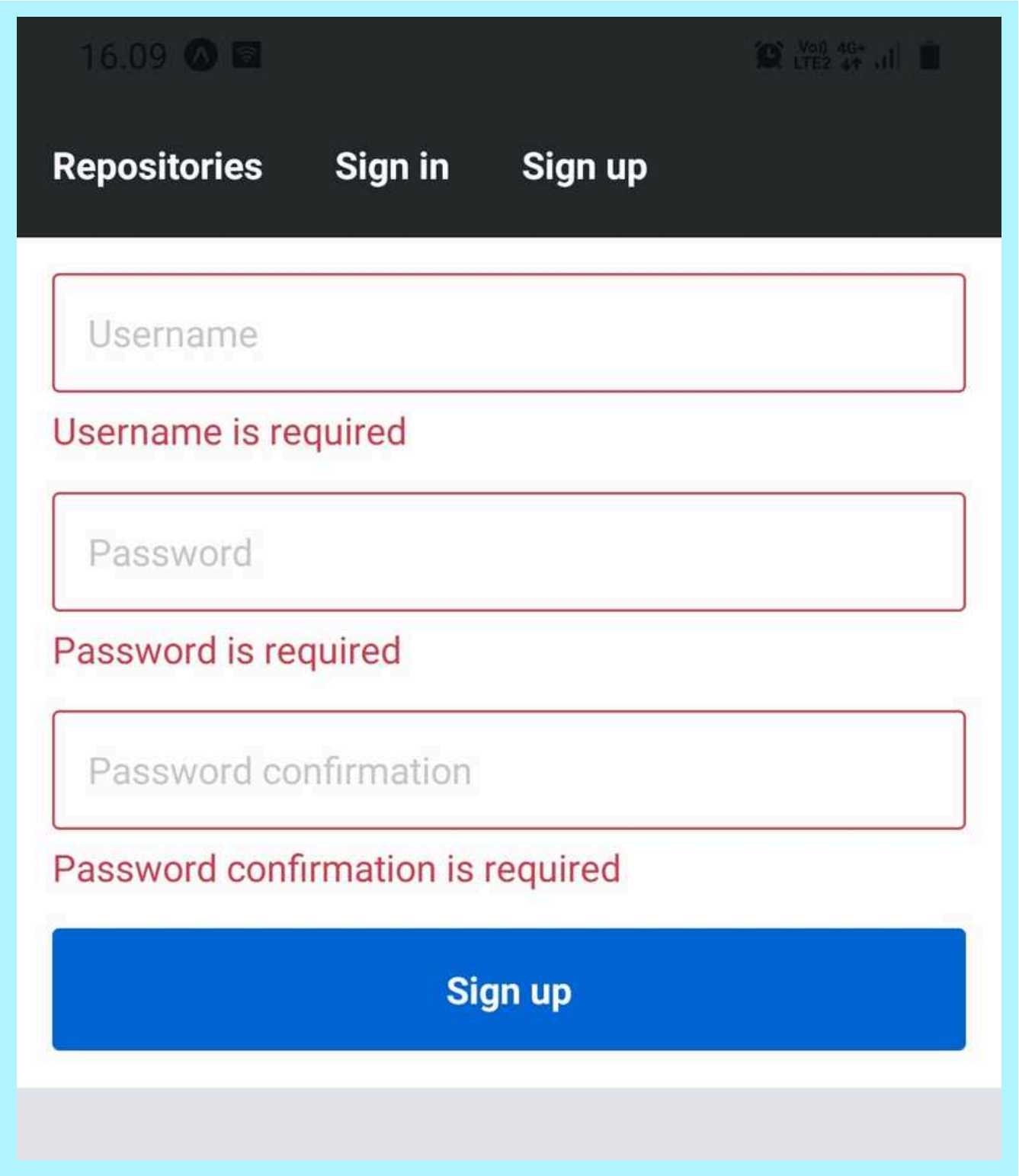
- Username is a required string with a length between 5 and 30
- Password is a required string with a length between 5 and 50
- Password confirmation matches the password

The password confirmation field's validation can be a bit tricky, but it can be done for example by using the `oneOf` and `ref` methods like suggested in this issue.

You can create a new user by using the `createUser` mutation. Find out how this mutation works by exploring the documentation in the Apollo Sandbox. After a successful `createUser` mutation, sign the created user in by using the `useSignIn` hook as we did in the sign in the form. After the user has been signed in, redirect the user to the reviewed repositories list view.

The user should be able to access the sign-up form through the app bar by pressing a "Sign up" tab. This tab should only be visible to users that aren't signed in.

The final version of the sign up form should look something like this:



This screenshot has been taken after invalid form submission to present what the form should look like in an invalid state.

Exercise 10.23: sorting the reviewed repositories list

At the moment repositories in the reviewed repositories list are ordered by the date of repository's first review. Implement a feature that allows users to select the principle, which is used to order the repositories. The available ordering principles should be:

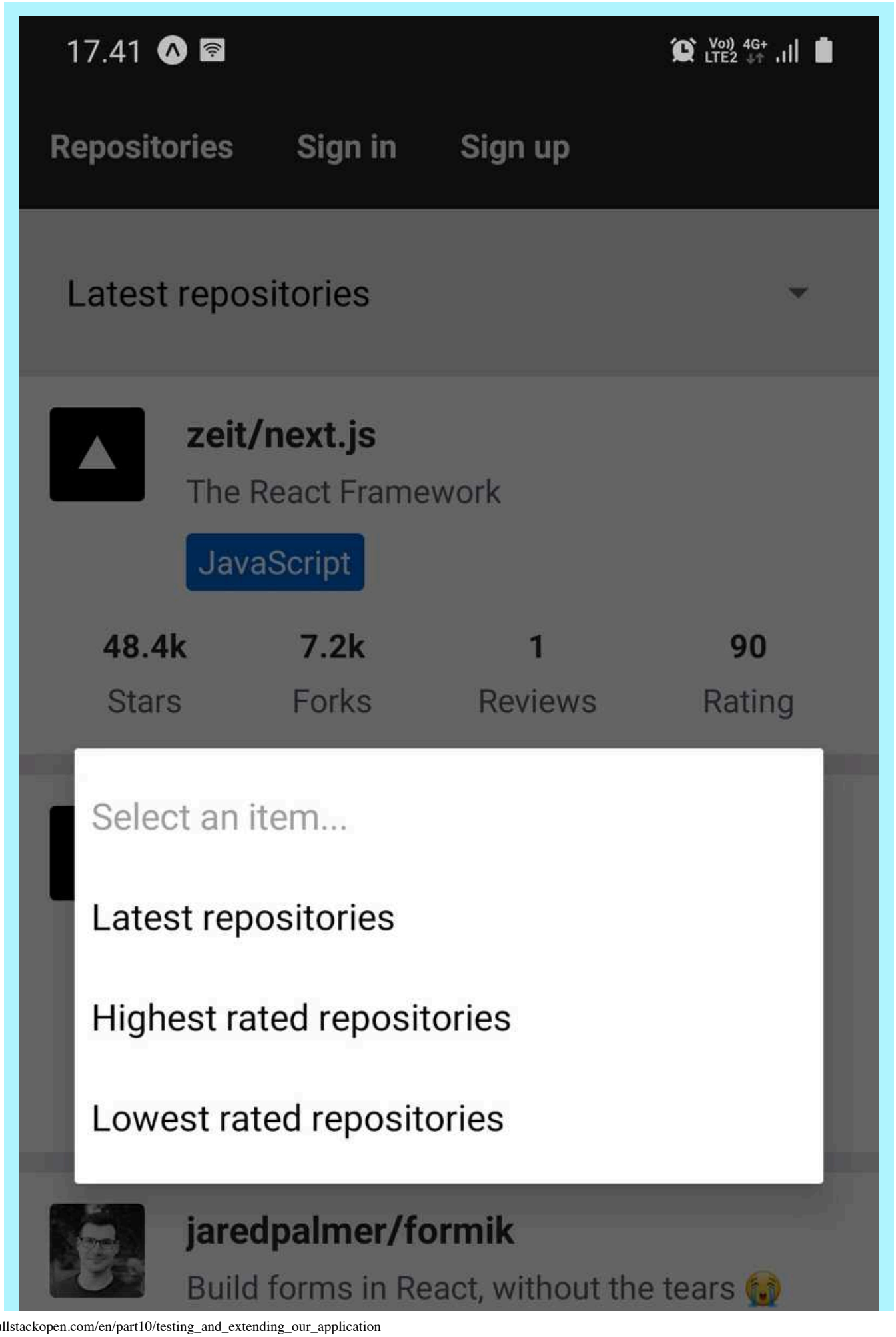
- Latest repositories. The repository with the latest first review is on the top of the list. This is the current behavior and should be the default principle.
- Highest rated repositories. The repository with the *highest* average rating is on the top of the list.
- Lowest rated repositories. The repository with the *lowest* average rating is on the top of the list.

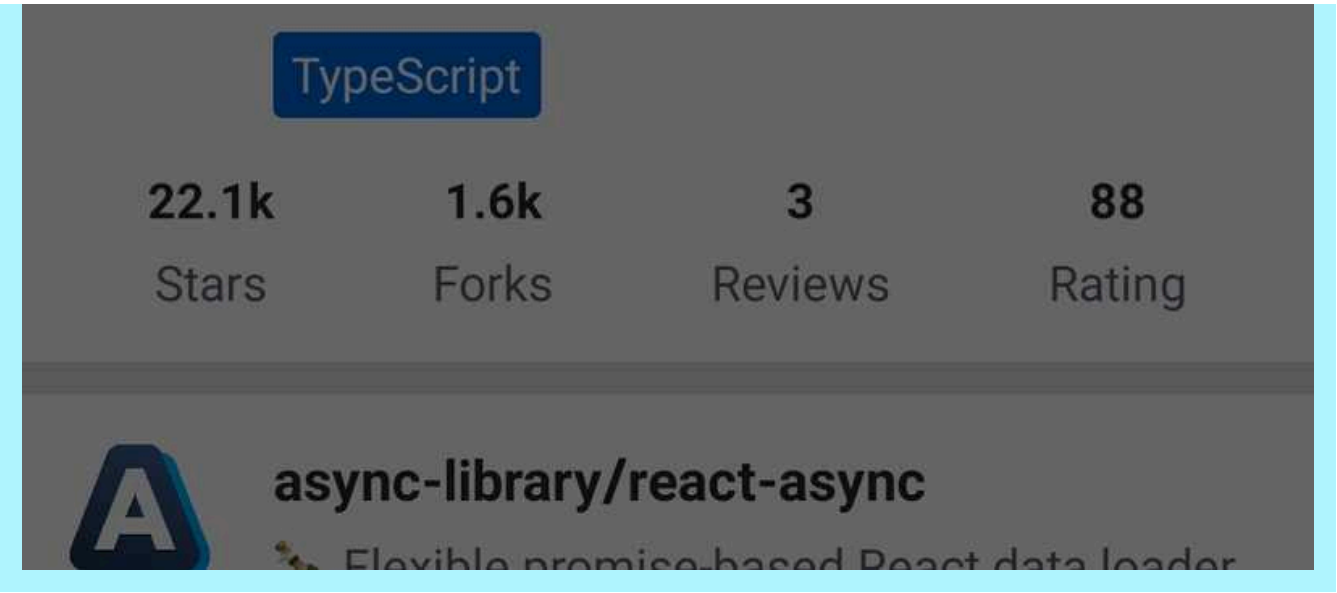
The `repositories` query used to fetch the reviewed repositories has an argument called `orderBy`, which you can use to define the ordering principle. The argument has two allowed values: CREATED_AT (order by the date of repository's first review) and RATING_AVERAGE, (order by the repository's average rating). The query also has an argument called `orderDirection` which can be used to change the order direction. The argument has two allowed values: `ASC` (ascending, smallest value first) and `DESC` (descending, biggest value first).

The selected ordering principle state can be maintained for example using the React's useState hook. The variables used in the `repositories` query can be given to the `useRepositories` hook as an argument.

You can use for example @react-native-picker/picker library, or React Native Paper library's Menu component to implement the ordering principle's selection. You can use the `FlatList` component's ListHeaderComponent prop to provide the list with a header containing the selection component.

The final version of the feature, depending on the selection component in use, should look something like this:


17.41
Repositories
Sign in
Sign up
Latest repositories
zeit/next.js
The React Framework
JavaScript
48.4k
Stars
7.2k
Forks
1
Reviews
90
Rating
Select an item...
Latest repositories
Highest rated repositories
Lowest rated repositories
jaredpalmer/formik
Build forms in React, without the tears



Exercise 10.24: filtering the reviewed repositories list

The Apollo Server allows filtering repositories using the repository's name or the owner's username. This can be done using the `searchKeyword` argument in the `repositories` query. Here's an example of how to use the argument in a query:

```
{
  repositories(searchKeyword: "ze") {
    edges {
      node {
        id
        fullName
      }
    }
  }
}
```

Implement a feature for filtering the reviewed repositories list based on a keyword. Users should be able to type in a keyword into a text input and the list should be filtered as the user types. You can use a simple `TextInput` component or something a bit fancier such as React Native Paper's Searchbar component as the text input. Put the text input component in the `FlatList` component's header.

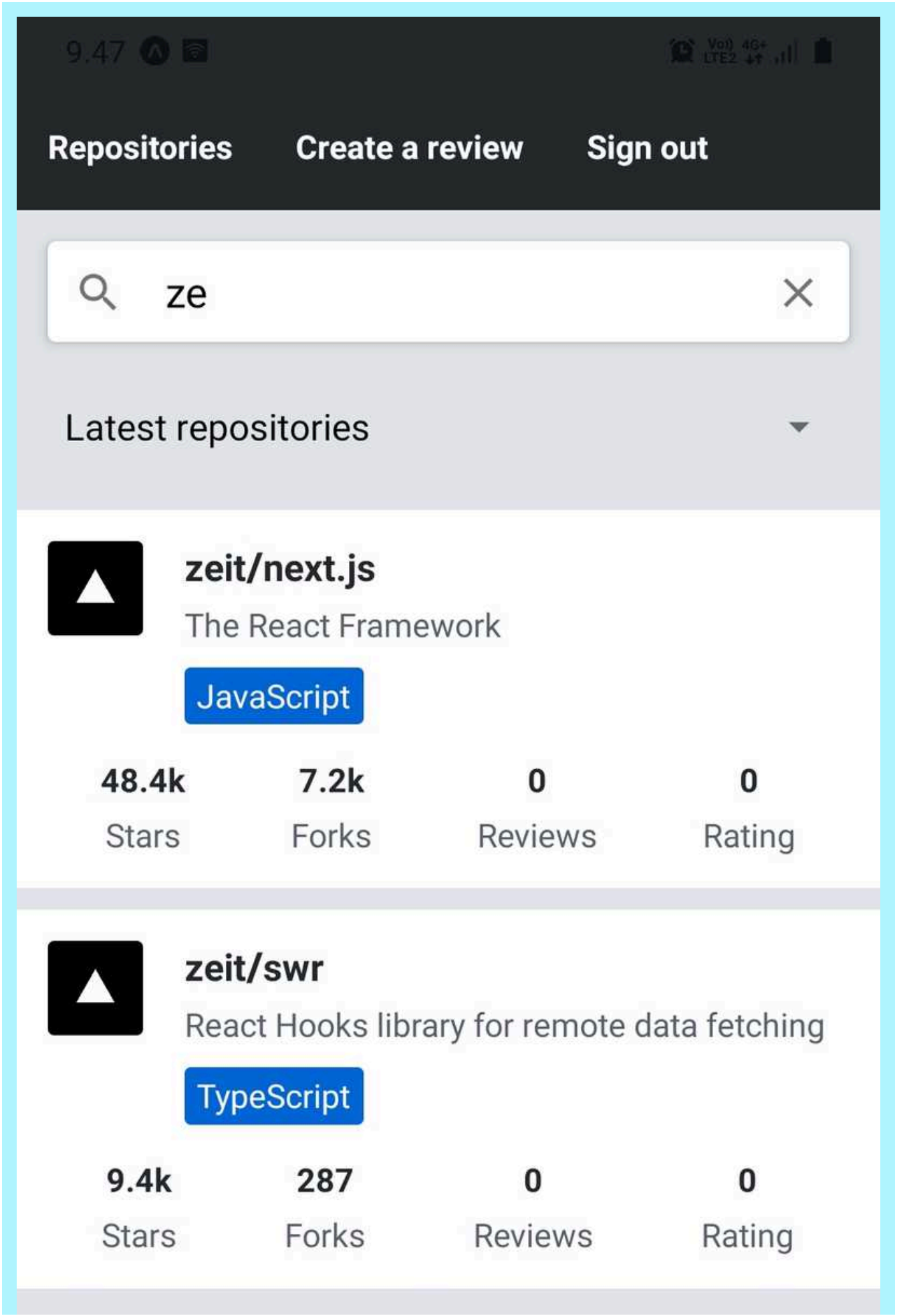
To avoid a multitude of unnecessary requests while the user types the keyword fast, only pick the latest input after a short delay. This technique is often referred to as debouncing. use-debounce library is a handy hook for debouncing a state variable. Use it with a sensible delay time, such as 500 milliseconds. Store the text input's value by using the `useState` hook and then pass the debounced value to the query as the value of the `searchKeyword` argument.

You probably face an issue that the text input component loses focus after each keystroke. This is because the content provided by the `ListHeaderComponent` prop is constantly unmounted. This can

be fixed by turning the component rendering the `FlatList` component into a class component and defining the header's render function as a class property like this:

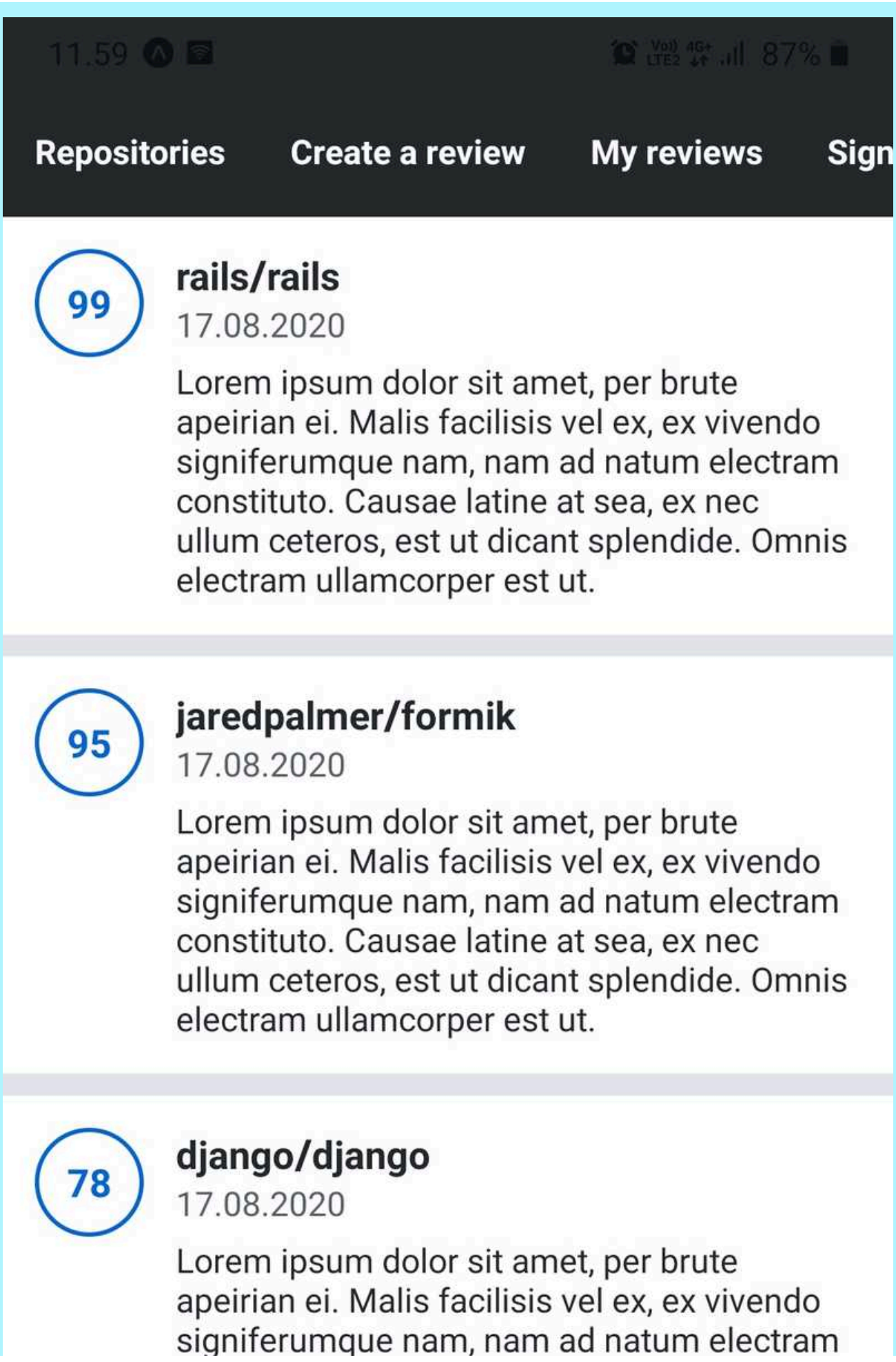
```
export class RepositoryListContainer extends React.Component {
  renderHeader = () => {
    // this.props contains the component's props
    const props = this.props;

    // ...

    return (
      <RepositoryListHeader
      // ...
      />
    );
  };

  render() {
    return (
      <FlatList
        // ...
        ListHeaderComponent={this.renderHeader}
      />
    );
  }
}
```

The final version of the filtering feature should look something like this:


9.47
Repositories
Create a review
Sign out
ze
Latest repositories
zeit/next.js
The React Framework
JavaScript
48.4k
Stars
7.2k
Forks
0
Reviews
0
Rating
zeit/swr
React Hooks library for remote data fetching
TypeScript
9.4k
Stars
287
Forks
0
Reviews
0
Rating

Exercise 10.25: the user's reviews view

Implement a feature which allows user to see their reviews. Once signed in, the user should be able to access this view by pressing a "My reviews" tab in the app bar. Here is what the review list view should roughly look like:


11.59
87%
Repositories
Create a review
My reviews
Sign
99
rails/rails
17.08.2020
Lorem ipsum dolor sit amet, per brute apeirian ei. Malis facilisis vel ex, ex vivendo signiferumque nam, nam ad natum electram constituto. Causae latine at sea, ex nec ullum ceteros, est ut dicant splendide. Omnis electram ullamcorper est ut.
95
jaredpalmer/formik
17.08.2020
Lorem ipsum dolor sit amet, per brute apeirian ei. Malis facilisis vel ex, ex vivendo signiferumque nam, nam ad natum electram constituto. Causae latine at sea, ex nec ullum ceteros, est ut dicant splendide. Omnis electram ullamcorper est ut.
78
django/django
17.08.2020
Lorem ipsum dolor sit amet, per brute apeirian ei. Malis facilisis vel ex, ex vivendo signiferumque nam, nam ad natum electram

Remember that you can fetch the authenticated user from the Apollo Server with the `me` query. This query returns a `User` type, which has a field `reviews`. If you have already implemented a reusable `me` query in your code, you can customize this query to fetch the `reviews` field conditionally. This can be done using GraphQL's include directive.

Let's say that the current query is implemented roughly in the following manner:

```
const GET_CURRENT_USER = gql`
  query {
    me {
      # user fields...
    }
  }
`;
```

copy

You can provide the query with an `includeReviews` argument and use that with the `include` directive:

```
const GET_CURRENT_USER = gql`
  query getCurrentUser($includeReviews: Boolean = false) {
    me {
      # user fields...
      reviews @include(if: $includeReviews) {
        edges {
          node {
            # review fields...
          }
        }
      }
    }
  }
`;
```

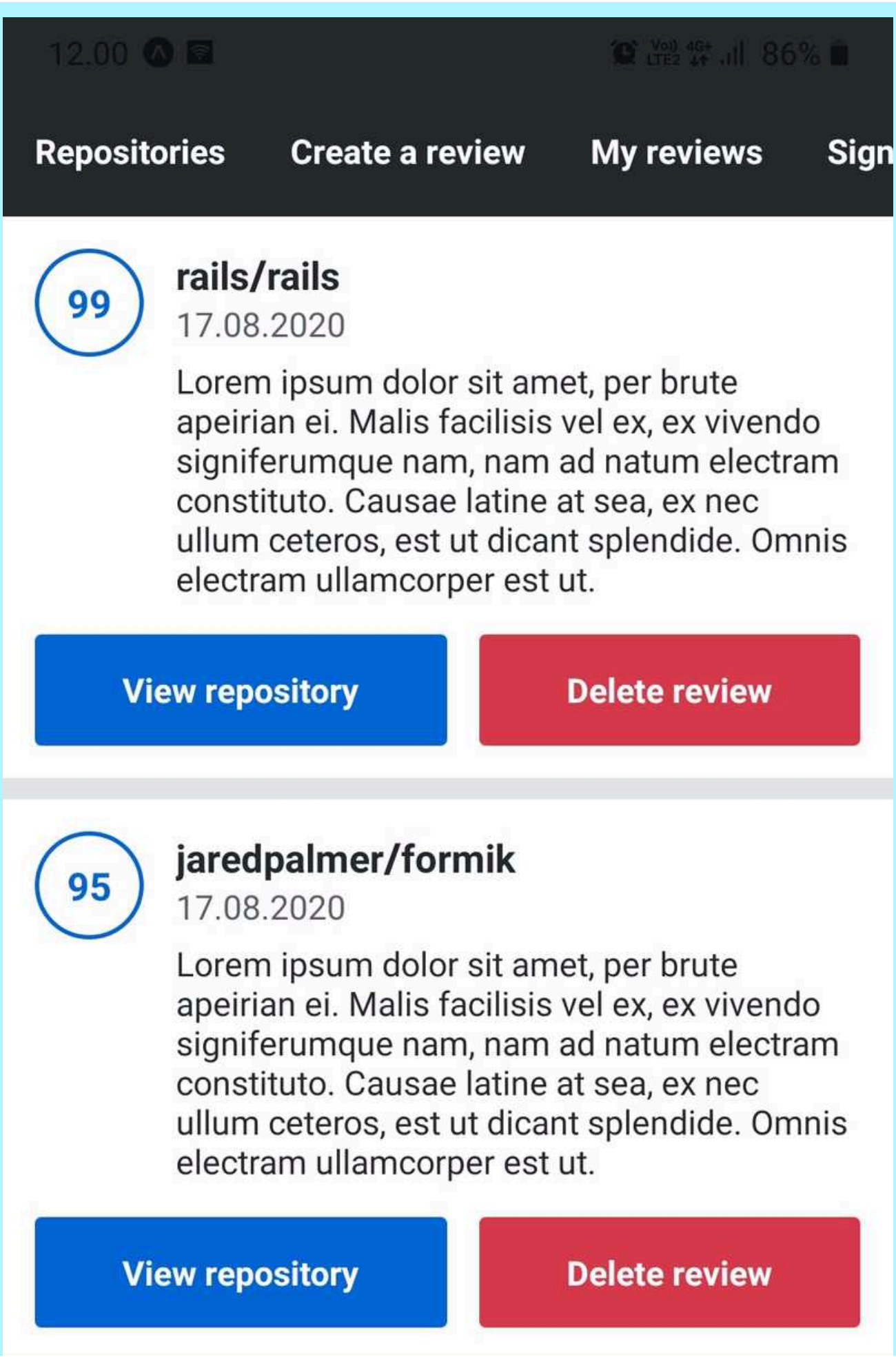
copy

The `includeReviews` argument has a default value of `false`, because we don't want to cause additional server overhead unless we explicitly want to fetch authenticated user's reviews. The principle of the `include` directive is quite simple: if the value of the `if` argument is `true`, include the field, otherwise omit it.

Exercise 10.26: review actions

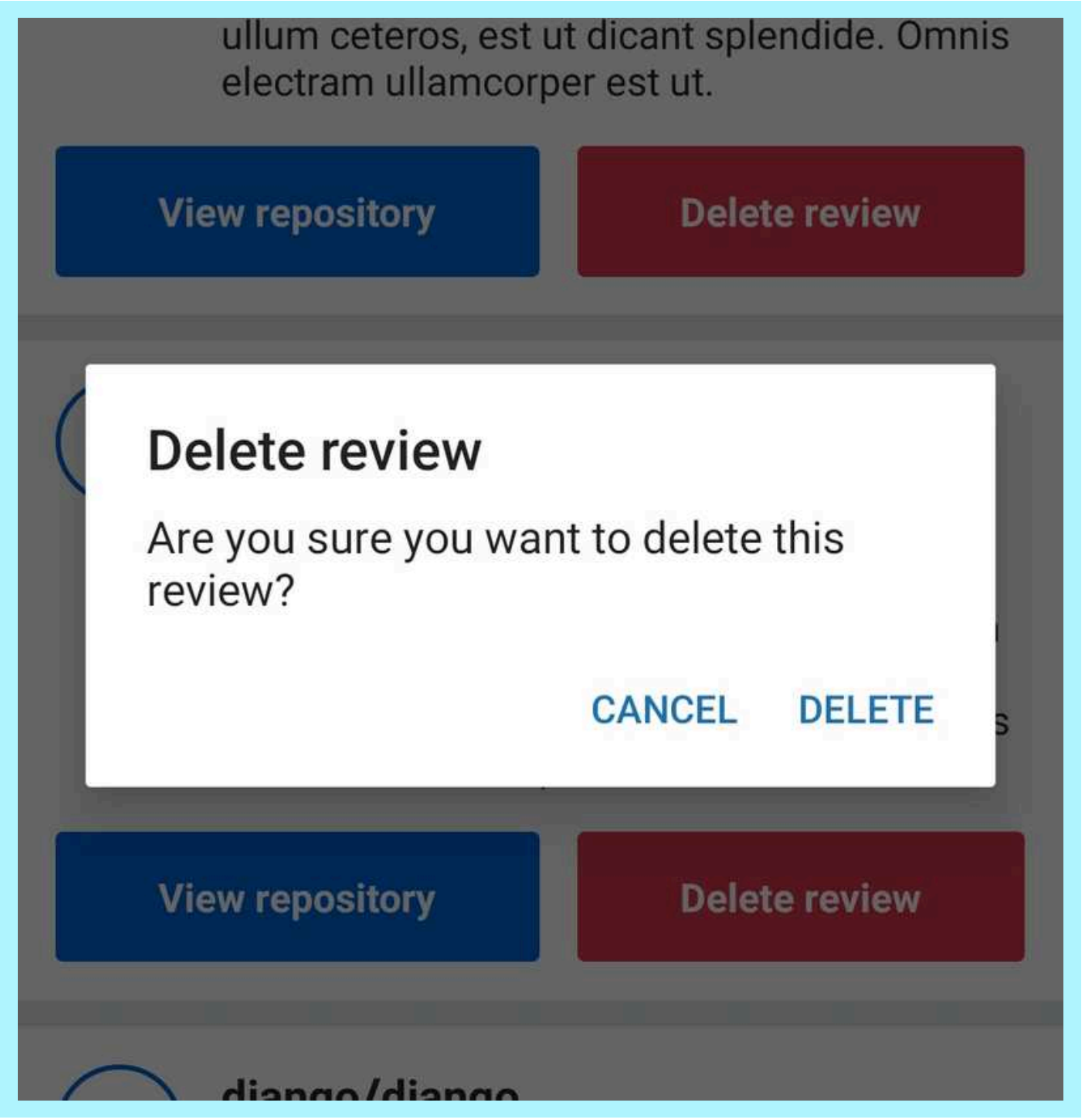
Now that user can see their reviews, let's add some actions to the reviews. Under each review on the review list, there should be two buttons. One button is for viewing the review's repository. Pressing this button should take the user to the single repository review implemented in the previous exercise. The

other button is for deleting the review. Pressing this button should delete the review. Here is what the actions should roughly look like:


12.00
86%
Repositories
Create a review
My reviews
Sign
99
rails/rails
17.08.2020
Lorem ipsum dolor sit amet, per brute apeirian ei. Malis facilisis vel ex, ex vivendo signiferumque nam, nam ad natum electram constituto. Causae latine at sea, ex nec ullum ceteros, est ut dicant splendide. Omnis electram ullamcorper est ut.
View repository
Delete review
95
jaredpalmer/formik
17.08.2020
Lorem ipsum dolor sit amet, per brute apeirian ei. Malis facilisis vel ex, ex vivendo signiferumque nam, nam ad natum electram constituto. Causae latine at sea, ex nec ullum ceteros, est ut dicant splendide. Omnis electram ullamcorper est ut.
View repository
Delete review

Pressing the delete button should be followed by a confirmation alert. If the user confirms the deletion, the review is deleted. Otherwise, the deletion is discarded. You can implement the confirmation using the Alert module. Note that calling the `Alert.alert` method won't open any window in Expo web preview. Use either Expo mobile app or an emulator to see the what the alert window looks like.

Here is the confirmation alert that should pop out once the user presses the delete button:



You can delete a review using the `deleteReview` mutation. This mutation has a single argument, which is the id of the review to be deleted. After the mutation has been performed, the easiest way to update the review list's query is to call the refetch function.

Cursor-based pagination

When an API returns an ordered list of items from some collection, it usually returns a subset of the whole set of items to reduce the required bandwidth and to decrease the memory usage of the client applications. The desired subset of items can be parameterized so that the client can request for example the first twenty items on the list after some index. This technique is commonly referred to as

pagination. When items can be requested after a certain item defined by a *cursor*, we are talking about *cursor-based pagination*.

So cursor is just a serialized presentation of an item in an ordered list. Let's have a look at the paginated repositories returned by the `repositories` query using the following query:

```
{
  repositories(first: 2) {
    totalCount
    edges {
      node {
        id
        fullName
        createdAt
      }
      cursor
    }
    pageInfo {
      endCursor
      startCursor
      hasNextPage
    }
  }
}
```

The `first` argument tells the API to return only the first two repositories. Here's an example of a result of the query:

```
{
  "data": {
    "repositories": {
      "totalCount": 10,
      "edges": [
        {
          "node": {
            "id": "zeit.next.js",
            "fullName": "zeit/next.js",
            "createdAt": "2020-05-15T11:59:57.557Z"
          },
          "cursor": "WyJ6ZWl0Lm5leHQuanMiLDE1ODk1NDM5OTc1NTdd"
        },
        {
          "node": {
            "id": "zeit.swr",
            "fullName": "zeit/swr",
            "createdAt": "2020-05-15T11:58:53.867Z"
          },
          "cursor": "WyJ6ZWl0LnN3ciIsMTU4OTU0MzkzMzg2N10="
        }
```

```
      ],
      "pageInfo": {
        "endCursor": "WyJ6ZWl0LnN3ciIsMTU4OTU0MzkzMzg2N10=",
        "startCursor": "WyJ6ZWl0Lm5leHQuanMiLDE1ODk1NDM5OTc1NTdd",
        "hasNextPage": true
      }
    }
  }
}
```

The format of the result object and the arguments are based on the Relay's GraphQL Cursor Connections Specification, which has become a quite common pagination specification and has been widely adopted for example in the GitHub's GraphQL API. In the result object, we have the `edges` array containing items with `node` and `cursor` attributes. As we know, the `node` contains the repository itself. The `cursor` on the other hand is a Base64 encoded representation of the node. In this case, it contains the repository's id and date of repository's creation as a timestamp. This is the information we need to point to the item when they are ordered by the creation time of the repository. The `pageInfo` contains information such as the cursor of the first and the last item in the array.

Let's say that we want to get the next set of items *after* the last item of the current set, which is the "zeit/swr" repository. We can set the `after` argument of the query as the value of the `endCursor` like this:

```
{
  repositories(first: 2, after: "WyJ6ZWl0LnN3ciIsMTU4OTU0MzkzMzg2N10=") {
    totalCount
    edges {
      node {
        id
        fullName
        createdAt
      }
      cursor
    }
    pageInfo {
      endCursor
      startCursor
      hasNextPage
    }
  }
}
```

copy

Now that we have the next two items and we can keep on doing this until the `hasNextPage` has the value `false`, meaning that we have reached the end of the list. To dig deeper into cursor-based pagination, read Shopify's article Pagination with Relative Cursors. It provides great details on the implementation itself and the benefits over the traditional index-based pagination.

Infinite scrolling

Vertically scrollable lists in mobile and desktop applications are commonly implemented using a technique called *infinite scrolling*. The principle of infinite scrolling is quite simple:

- Fetch the initial set of items
- When the user reaches the last item, fetch the next set of items after the last item

The second step is repeated until the user gets tired of scrolling or some scrolling limit is exceeded. The name "infinite scrolling" refers to the way the list seems to be infinite - the user can just keep on scrolling and new items keep on appearing on the list.

Let's have a look at how this works in practice using the Apollo Client's `useQuery` hook. Apollo Client has a great documentation on implementing the cursor-based pagination. Let's implement infinite scrolling for the reviewed repositories list as an example.

First, we need to know when the user has reached the end of the list. Luckily, the `FlatList` component has a prop onEndReached, which will call the provided function once the user has scrolled to the last item on the list. You can change how early the `onEndReach` callback is called using the onEndReachedThreshold prop. Alter the `RepositoryList` component's `FlatList` component so that it calls a function once the end of the list is reached:

```
export const RepositoryListContainer = ({
  repositories,
  onEndReach,
  /* ... */,
}) => {
  const repositoryNodes = repositories
    ? repositories.edges.map((edge) => edge.node)
    : [];

  return (
    <FlatList
      data={repositoryNodes}
      // ...
      onEndReached={onEndReach}
      onEndReachedThreshold={0.5}
    />
  );
};

const RepositoryList = () => {
  // ...

  const { repositories } = useRepositories(/* ... */);

  const onEndReach = () => {
    console.log('You have reached the end of the list');
  };
```

```
  return (
    <RepositoryListContainer
      repositories={repositories}
      onEndReach={onEndReach}
      // ...
    />
  );
};

export default RepositoryList;
```

Try scrolling to the end of the reviewed repositories list and you should see the message in the logs.

Next, we need to fetch more repositories once the end of the list is reached. This can be achieved using the fetchMore function provided by the `useQuery` hook. To describe to Apollo Client how to merge the existing repositories in the cache with the next set of repositories, we can use a field policy. In general, field policies can be used to customize the cache behavior during read and write operations with read and merge functions.

Let's add a field policy for the `repositories` query in the *apolloClient.js* file:

```
import { ApolloClient, InMemoryCache, createHttpLink } from '@apollo/client';
import { setContext } from '@apollo/client/link/context';
import Constants from 'expo-constants';
import { relayStylePagination } from '@apollo/client/utilities';

const { apolloUri } = Constants.manifest.extra;

const httpLink = createHttpLink({
  uri: apolloUri,
});

const cache = new InMemoryCache({
  typePolicies: {
    Query: {
      fields: {
        repositories: relayStylePagination(),
      },
    },
  },
});

const createApolloClient = (authStorage) => {
  const authLink = setContext(async (_, { headers }) => {
    try {
      const accessToken = await authStorage.getAccessToken();

      return {
        headers: {
          ...headers,
          authorization: accessToken ? `Bearer ${accessToken}` : '',
        },
```

```
      };
    } catch (e) {
      console.log(e);

      return {
        headers,
      };
    }
  });

  return new ApolloClient({
    link: authLink.concat(httpLink),
    cache,
  });
};

export default createApolloClient;
```

As mentioned earlier, the format of the pagination's result object and the arguments are based on the Relay's pagination specification. Luckily, Apollo Client provides a predefined field policy, `relayStylePagination`, which can be used in this case.

Next, let's alter the `useRepositories` hook so that it returns a decorated `fetchMore` function, which calls the actual `fetchMore` function with appropriate arguments so that we can fetch the next set of repositories:

copy

```
const useRepositories = (variables) => {
  const { data, loading, fetchMore, ...result } = useQuery(GET_REPOSITORIES, {
    variables,
    // ...
  });

  const handleFetchMore = () => {
    const canFetchMore = !loading && data?.repositories.pageInfo.hasNextPage;

    if (!canFetchMore) {
      return;
    }

    fetchMore({
      variables: {
        after: data.repositories.pageInfo.endCursor,
        ...variables,
      },
    });
  };

  return {
    repositories: data?.repositories,
    fetchMore: handleFetchMore,
    loading,
    ...result,
```

```
  };
};
```

Make sure you have the `pageInfo` and the `cursor` fields in your `repositories` query as described in the pagination examples. You will also need to include the `after` and `first` arguments for the query.

The `handleFetchMore` function will call the Apollo Client's `fetchMore` function if there are more items to fetch, which is determined by the `hasNextPage` property. We also want to prevent fetching more items if fetching is already in process. In this case, `loading` will be `true` . In the `fetchMore` function we are providing the query with an `after` variable, which receives the latest `endCursor` value.

The final step is to call the `fetchMore` function in the `onEndReach` handler:

```
const RepositoryList = () => {
  // ...

  const { repositories, fetchMore } = useRepositories({
    first: 8,
    // ...
  });

  const onEndReach = () => {
    fetchMore();
  };

  return (
    <RepositoryListContainer
      repositories={repositories}
      onEndReach={onEndReach}
      // ...
    />
  );
};

export default RepositoryList;
```

Use a relatively small `first` argument value such as 3 while trying out the infinite scrolling. This way you don't need to review too many repositories. You might face an issue that the `onEndReach` handler is called immediately after the view is loaded. This is most likely because the list contains so few repositories that the end of the list is reached immediately. You can get around this issue by increasing the value of `first` argument. Once you are confident that the infinite scrolling is working, feel free to use a larger value for the `first` argument.

Exercise 10.27

Exercise 10.27: infinite scrolling for the repository's reviews list

Implement infinite scrolling for the repository's reviews list. The `Repository` type's `reviews` field has the `first` and `after` arguments similar to the `repositories` query. `ReviewConnection` type also has the `pageInfo` field just like the `RepositoryConnection` type.

Here's an example query:

```
{
  repository(id: "jaredpalmer.formik") {
    id
    fullName
    reviews(first: 2, after:
"WyIxYjEwZTRkOC01N2VlLTRkMDAtODg4Ni1lNGEwNDlkN2ZmOGYuamFyZWRwYWxtZXIuZm9ybWlrIiwxNTg4NjU2Nzl
{
      totalCount
      edges {
        node {
          id
          text
          rating
          createdAt
          repositoryId
          user {
            id
            username
          }
        }
        cursor
      }
      pageInfo {
        endCursor
        startCursor
        hasNextPage
      }
    }
  }
}
```

The cache's field policy can be similar as with the `repositories` query:

```
const cache = new InMemoryCache({
  typePolicies: {
```

```
    Query: {
      fields: {
        repositories: relayStylePagination(),
      },
    },
    Repository: {
      fields: {
        reviews: relayStylePagination(),
      },
    },
  },
});
```

As with the reviewed repositories list, use a relatively small `first` argument value while you are trying out the infinite scrolling. You might need to create a few new users and use them to create a few new reviews to make the reviews list long enough to scroll. Set the value of the `first` argument high enough so that the `onEndReach` handler isn't called immediately after the view is loaded, but low enough so that you can see that more reviews are fetched once you reach the end of the list. Once everything is working as intended you can use a larger value for the `first` argument.

This was the last exercise in this section. It's time to push your code to GitHub and mark all of your finished exercises to the exercise submission system. Note that exercises in this section should be submitted to the part 4 in the exercise submission system.

Additional resources

As we are getting closer to the end of this part, let's take a moment to look at some additional React Native related resources. Awesome React Native is an extremely encompassing curated list of React Native resources such as libraries, tutorials, and articles. Because the list is exhaustively long, let's have a closer look at few of its highlights

React Native Paper

> Paper is a collection of customizable and production-ready components for React Native, following Google's Material Design guidelines.

React Native Paper is for React Native what Material-UI is for React web applications. It offers a wide range of high-quality UI components, support for custom themes and a fairly simple setup for Expo based React Native applications.

Styled-components

> Utilising tagged template literals (a recent addition to JavaScript) and the power of CSS, styled-components allows you to write actual CSS code to style your components. It also removes the mapping between components and styles – using components as a low-level styling construct could not be easier!

Styled-components is a library for styling React components using CSS-in-JS technique. In React Native we are already used to defining component's styles as a JavaScript object, so CSS-in-JS is not so uncharted territory. However, the approach of styled-components is quite different from using the `StyleSheet.create` method and the `style` prop.

In styled-components components' styles are defined with the component using a feature called tagged template literal or a plain JavaScript object. Styled-components makes it possible to define new style properties for component based on its props `at runtime`. This brings many possibilities, such as seamlessly switching between a light and a dark theme. It also has a full theming support. Here is an example of creating a `Text` component with style variations based on props:

```
import styled from 'styled-components/native';
import { css } from 'styled-components';

const FancyText = styled.Text`
  color: grey;
  font-size: 14px;

  ${({ isBlue }) =>
    isBlue &&
    css`
      color: blue;
    `}

  ${({ isBig }) =>
    isBig &&
    css`
      font-size: 24px;
      font-weight: 700;
    `}
`;

const Main = () => {
  return (
    <>
      <FancyText>Simple text</FancyText>
      <FancyText isBlue>Blue text</FancyText>
      <FancyText isBig>Big text</FancyText>
      <FancyText isBig isBlue>
        Big blue text
      </FancyText>
    </>
  );
};
```

Because styled-components processes the style definitions, it is possible to use CSS-like snake case syntax with the property names and units in property values. However, units don't have any effect because property values are internally unitless. For more information on styled-components, head out to the documentation.

React-spring

> react-spring is a spring-physics based animation library that should cover most of your UI related animation needs. It gives you tools flexible enough to confidently cast your ideas into moving interfaces.

React-spring is a library that provides a clean API for animating React Native components.

React Navigation

> Routing and navigation for your React Native apps

React Navigation is a routing library for React Native. It shares some similarities with the React Router library we have been using during this and earlier parts. However, unlike React Router, React Navigation offers more native features such as native gestures and animations to transition between views.

Closing words

That's it, our application is ready. Good job! We have learned many new concepts during our journey such as setting up our React Native application using Expo, using React Native's core components and adding style to them, communicating with the server, and testing React Native applications. The final piece of the puzzle would be to deploy the application to the Apple App Store and Google Play Store.

Deploying the application is entirely *optional* and it isn't quite trivial, because you also need to fork and deploy the rate-repository-api. For the React Native application itself, you first need to create either iOS or Android builds by following Expo's documentation. Then you can upload these builds to either Apple App Store or Google Play Store. Expo has documentation for this as well.

Propose changes to material

Part 10c
Previous part

Part 11
Next part

About course

Course contents

FAQ

Partners

Challenge


UNIVERSITY OF HELSINKI


HOUSTON