Building Web Applications using ReactJS

QuickLabs Guide

CONTENTS

QuickLabs Environment Set-Up

Code Editing

1. Open **VSCode** (or download and install if not present).
    - Use the *desktop shortcut* to open the **VSCode** download page:
    - For **Windows** users download the **64-bit System Installer**.
2. Check for *updates* and download and install if necessary:
    - For **Windows** Users click **Help - Check for updates**;
    - For **MacOS** Users click **Code - Check for updates**.
3. Using **File - Open**, navigate to the **QuickLabs** folder and click **Open**. This will give you access to all of the **QuickLab** files and solutions needed to complete the **QuickLabs**.

NodeJS

1. Use the *desktop shortcut* to open the **NodeJS** download page.
2. Download and install the **LTS** version for the operating system you are working in:
    - For **Windows** users, download the **Installer file (.msi)**;
    - For **MacOS** users, download the **Installer file (.pkg)**.

Do This Before Each QuickLab

Unless specifically directed to do otherwise, the following steps should be taken before starting each QuickLab:

1. Point the terminal/command line at the QuickLab **starter** folder that contains the **package.json**.
2. Run the command:

```
npm i
```

3. Compile and output the project by running the command:

```
npm start
```

4. Navigate the browser to:

https://localhost:3000/

if it does not open automatically.

Code Snippets

Whilst the QuickLabs provided are supposed to challenge you, they are not supposed to baffle! Annotated snippets for each instruction are provided at the end of each QuickLab. Try not to use them until you have had a go at writing the code yourself!

Quick Lab 1 – Get a ReactJS App Up and Running

Objectives

- To be able to use the create-react-app node package extractor to quickly scaffold a ReactJS application
- To be able to launch the application in the browser using the command line

Overview

In this Quicklab, you will set up a ReactJS application using a special node package extractor called create-react-app. Once the installation of files has completed, you will launch the application in the browser and see it running.

Activity

Skip the 'Before Each QuickLab' for this Activity.

1. Using CTRL + ' on the keyboard (CTRL + ` on MacOS) or by using click-path View – Terminal (or Terminal – New Terminal on MacOS), open VSCode's integrated terminal or click the terminal icon on the bottom bar.
2. Using the cd command, navigate to the QuickLabs/a-react-app/ folder.
3. Create a *new* ReactJS application using the command:

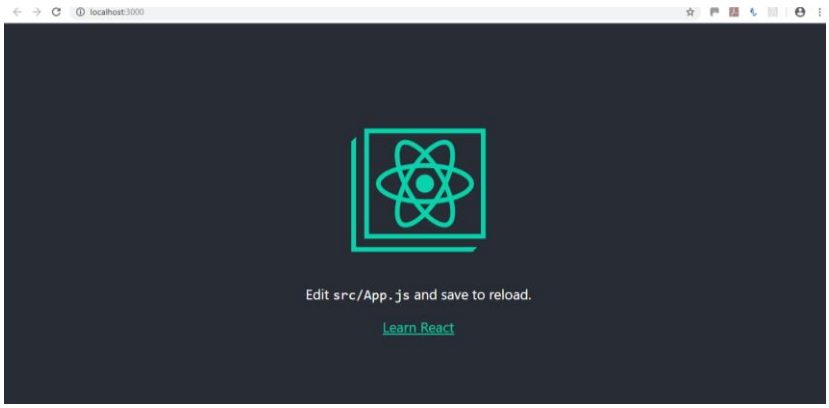
```
npx create-react-app starter
```

Wait for the installation to complete.

4. Use the cd command to change into the starter folder and then run the application by using the command:

```
cd starter && npm start
```

Your browser should open at *http://localhost:3000* with the following screen (or similar):



This is the end of Quick Lab 1

Quick Lab 2 – Build and Run the Application

Objectives

- To be able to build production-ready code using the scripts provided in the application
- To understand what the build process creates and where the files are put

Overview

In this QuickLab, you will produce a production-ready set of code for the skeleton application. This will make bundles of the HTML, CSS and JavaScript needed to efficiently deploy the application. You will explore the files that are created and view the application in the browser.

Activity

Skip the section 'Before Each QuickLab' before continuing using the a-react-app/starter folder.

1. In VSCode, if the server is running on the command line, press CTRL+C to stop it.
2. Make sure that the command line is pointing to QuickLabs/a-react-app/starter.
3. Make a production ready version of the application by typing:

```
npm run build
```

4. Once the process has finished, install a server to view the application:

```
npm i -g serve
```

5. Once the installation is complete, run the App in the server:

```
serve -s build
```

6. Open the browser to *http://localhost:5000* and view the application.
7. Browse the files created in a new folder called **build** in the application root:
    - Find **index.html** and its reference to the JS files;
    - Find the JavaScript files – view these in VSCode.

Building the application optimises the files for the fastest download without affecting functionality.

This is the end of Quick Lab 2

Quick Lab 3 – Create a Function Component

Objectives

- To be able to create function components
- To be able render components as children of others

Overview

In this QuickLab, you will create function components in their own files. You will then import these components into parent components, rendering as part of the parent's return.

Activity – Part 1 – MyComponent

Skip the section 'Before Each QuickLab' before continuing using the a-react-app/starter folder.

1. Create a new file called MyComponent.jsx in the a-react-app/starter/src folder.
2. Add an `import` for `React` from `'react'`.
3. Create a `const` called `MyComponent` as an *arrow function* that takes *no arguments*.
4. Make the function `return` a single `<h1>` with the text `Hello World`.
5. `export` `MyComponent` as a `default`.
6. Open App.js from the same folder and *delete EVERYTHING* in its `return`.
7. Put `MyComponent` as an *element* in the `return`, ensuring that it is *imported*.
8. Save all files and run the application – use `npm start` from the command line if required.

The app's display should now have been replaced with the content provided in MyComponent.

Activity – Part 2 – AnotherComponent

1. Create a new file called AnotherComponent.jsx in the a-react-app/starter/src folder.
2. Add an `import` for `React` from `'react'`.
3. Create a `const` called `AnotherComponent` as an *arrow function* that takes *no arguments*.
4. Make the function `return` a React Fragment `<></>` with 2 paragraphs that contain some text – we used 10 '*lorem ipsum*' words.

5. `export` `AnotherComponent` as a `default`.
6. Open MyComponent.jsx and add `<AnotherComponent />` *under* the `<h1>` and wrap both in a React Fragment `<></>` (ensuring AnotherComponent is *imported*).
7. Save all files and run the application (npm start from the command line).

You should see the text from AnotherComponent seamlessly displayed.

Code Snippets

Part 1 – MyComponent.jsx

```
import React from 'react';                                  // 2

const MyComponent = () => {                                 // 3
  return (                                                  // 4
    <h1>Hello World</h1>
  );
};
export default MyComponent;                                 // 5
```

Part 1 – App.jsx

```
import React from 'react';
import MyComponent from './MyComponent';                    // 7

function App() {
    return (
        <MyComponent />                                     // 7
    );
}
export default App;
```

Part 2 – AnotherComponent.jsx

```
import React from 'react';                                  // 2

const AnotherComponent = () => {                            // 3
  return (                                                  // 4
    <>
      <p>lorem ispum…</p>
      <p>lorem ipsum…</p>
    </>
  );
};
export default AnotherComponent;                            // 5
```

MyComponent.jsx

```
import React from 'react';
import AnotherComponent from './AnotherComponent';          // 6

const MyComponent = () => {
  return (                                                  // 6
    <>
      <h1>Hello World</h1>
      <AnotherComponent />
    </>
  );
};
export default MyComponent;
```

This is the end of Quick Lab 3

Quick Lab 4 - Creating a Class Component

Objectives

- To be able to create a class component
- To be able to nest components in others

Overview

In this QuickLab, you will create a new Angular component using the CLI, exploring the files that are created and modified as part of the process. You will then nest this new component in the existing App component.

Activity

Skip the section 'Before Each QuickLab' before continuing using the a-react-app/starter folder.

1. Create a new file called MyClassComponent.jsx in the a-react-app/starter/src folder.
2. Add an `import` for `React` and `{ Component }` from `'react'`.
3. Create a `class` called `MyClassComponent` that `extends` `Component`.
4. Make the `render` function `return` a React Fragment `<></>` with a `<h2>` and a `<p>` that contains some text.
5. `export` the `MyClassComponent` as a `default`.
6. Open MyComponent.jsx and add `<MyClassComponent />` under the `<AnotherComponent />` (ensuring MyClassComponent is imported).
7. Save all files and run the application (npm start from the command line if not running already)

You should see the MyClassComponent seamlessly displayed with all of the others.

Code Snippets

MyClassComponent.jsx

```
import React, { Component } from 'react';                          // 2

class MyClassComponent extends Component {                         // 3
  render () {                                                      // 4
    return (
      <>
        <h2>Class Components</h2>
        <p>Work in a similar way to Function Components</p>
      </>
    );
  }
}

export default MyClassComponent;                                   // 5
```

MyComponent.jsx

```
import React from 'react';
import AnotherComponent from './AnotherComponent';
import MyClassComponent from './MyClassComponent';                 // 6

const MyComponent = () => {
  return (
    <>
      <h1>Hello World</h1>
      <AnotherComponent />
      <MyClassComponent />                                         // 6
    </>
  );
};

export default MyComponent;
```

This is the end of Quick Lab 4

Quick Lab 5 - Jest

There are no activities in this QuickLab.

Quick Lab 6 – Thinking in React Part 1 – Component Hierarchy

Objectives

- To be able to take acceptance criteria, a mock-up and some static data to produce a suitable component hierarchy for an application

Overview

In this QuickLab, you will use acceptance criteria, the provided mock-ups and static data to identify a component hierarchy for a Todo application. A hierarchy is needed for an 'AllTodos' UI and an 'Add/Edit Todo' UI.

Activity

Skip the 'Before Each QuickLab' for this Activity.

1. Using the information provided below, create a component hierarchy for the 'AllTodos' UI and the 'Add/Edit Todo' UI.

Acceptance Criteria – ALL UIs

Footer	Header
Acceptance Criteria Delete 0% ☐ The UI should be created using ReactJS and JSX ☐ Every UI should have a <footer> ☐ The <footer> section should use the Bootstrap CSS classes to automatically set the top margin and left and right padding to 1rem ☐ The text in the footer should be centered using the "container" and "align-center" Bootstrap CSS classes You have unsaved edits on this field. View edits - Discard ☐ The 'footer' should contain the text '© QA Ltd 2019-'	**Acceptance Criteria** Delete 0% ☐ The UI should be created using ReactJS and JSX ☐ Every UI should have a <header> which wraps a <nav> with Bootstrap classes to set it as a navbar that is responsive and stacks the navigation on small screens ☐ The <nav> should contain the QA Logo having alt text of "QA Ltd" and a width of 100 ☐ The logo should be linked to https://www.qa.com, opening in a new window. The link should use Bootstrap class to signify that this is the brand and have rel properties of noopener and noreferrer. ☐ The <nav> section should contain the title 'Todo App' in a <h1> that is linked to / in an <a> that has a Bootstrap class to signify that this is for the brand

Acceptance Criteria – Specific UIs

All Todos

Acceptance Criteria Delete

0%

- [ ] The UI should be created using ReactJS and JSX
- [ ] All Todos UI should display a title of 'Todos List'
- [ ] Todos List should be presented in a striped table.
- [ ] Todos List table should have 3 columns: "Description", "Date Created" and "Action".
- [ ] Each Todo in the list should be presented as a row in the table
- [ ] Completed Todos should be struck through and an action of "N/A"
- [ ] Todos not completed should have an action of "Edit" which is a link

Add/Edit Todos

Acceptance Criteria Delete

0%

- [ ] The UI should be created using ReactJS and JSX
- [ ] Add/Edit UI should be wrapped in a with a class of addEditTodo and display a title of a <h3> with text 'Add/Edit Todo'.
- [ ] Todo to add or edit should be presented in a <form>.
- [ ] The todoDescription should be wrapped in a with a Bootstrap CSS class of "form-group"
- [ ] The todoDescription should have a <label> with the text 'Description:'
- [ ] The todoDescription should be an <input> with a 'type' of "text", a 'name' of "todoDescription" and a Bootstrap CSS class of "form-control".
- [ ] The todoDateCreated should be wrapped in a with a Bootstrap CSS class of "form-group".
- [ ] The todoDateCreated should have a <label> with the text 'Created on:' and display the current date and time in a next to it.
- [ ] The submit button should be wrapped in a with a Bootstrap CSS class of "form-group".
- [ ] The todoCompleted should be wrapped in a with a Bootstrap CSS class of "form-group".
- [ ] The todoCompleted should have a <label> with the text 'Completed:'
- [ ] The todoCompleted should be an <input> with a 'type' of "checkbox", a 'name' of "todoCompleted" and a Bootstrap CSS class of "form-control"
- [ ] The submit button should be an <input> with a 'type' of "submit", have a 'value' of "Submit" and Bootstrap CSS classes of "btn btn-primary".

Mock Data

A copy of this data is also available in the file **sampleTodos.json** in **b-static-version/starter/src**.

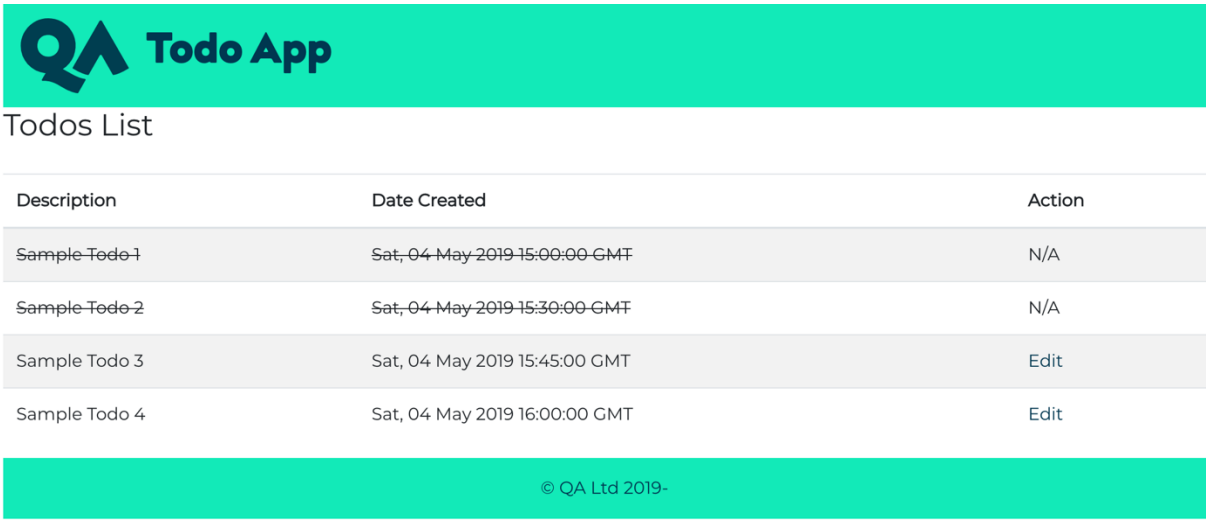
Notes:

- _id is in the format generated when an item is added to MongoDB
- todoDateCreated is in ISO Date format as this is used to store dates/times in MongoDB.

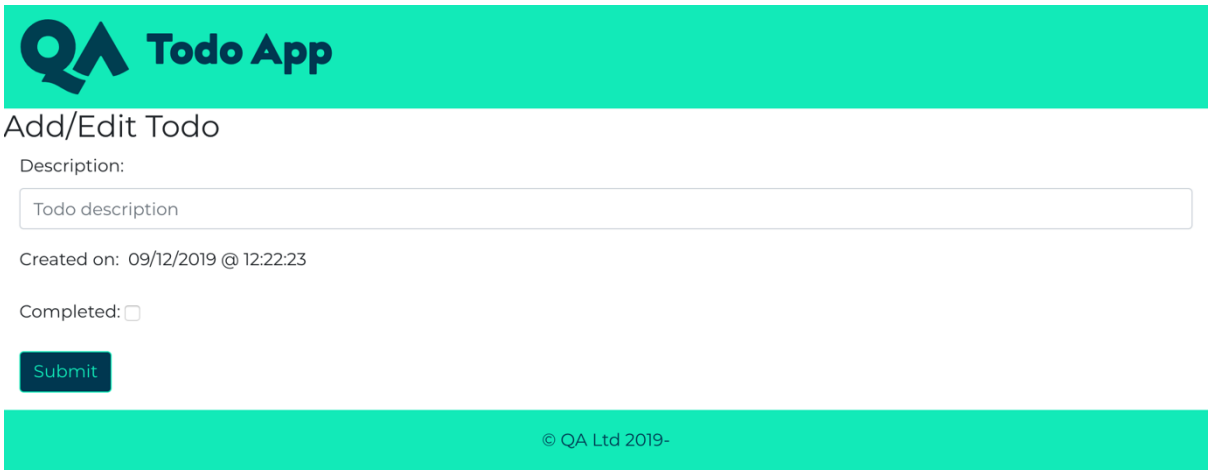
```
[
  {
        "_id": "5cc084952deb33810d2ec464",
        "todoDescription": "Sample Todo 1",
        "todoDateCreated": "2019-05-04T15:00:00.000Z",
        "todoCompleted": true
    },
    {
        "_id": "5cc08495bf3fd62d03f2f4c2",
        "todoDescription": "Sample Todo 2",
        "todoDateCreated": "2019-05-04T15:30:00.000Z",
        "todoCompleted": true
    },
    {
        "_id": "5cc08495bf3fd62d03f2f4c2",
        "todoDescription": "Sample Todo 3",
        "todoDateCreated": "2019-05-04T15:45:00.000Z",
        "todoCompleted": false
    },

    {
        "_id": "5cc08495bf3fd62d03f2f4c2",
        "todoDescription": "Sample Todo 4",
        "todoDateCreated": "2019-05-04T16:00:00.000Z",
        "todoCompleted": false
    }
]
```

Wireframes/Mock Ups

All Todos



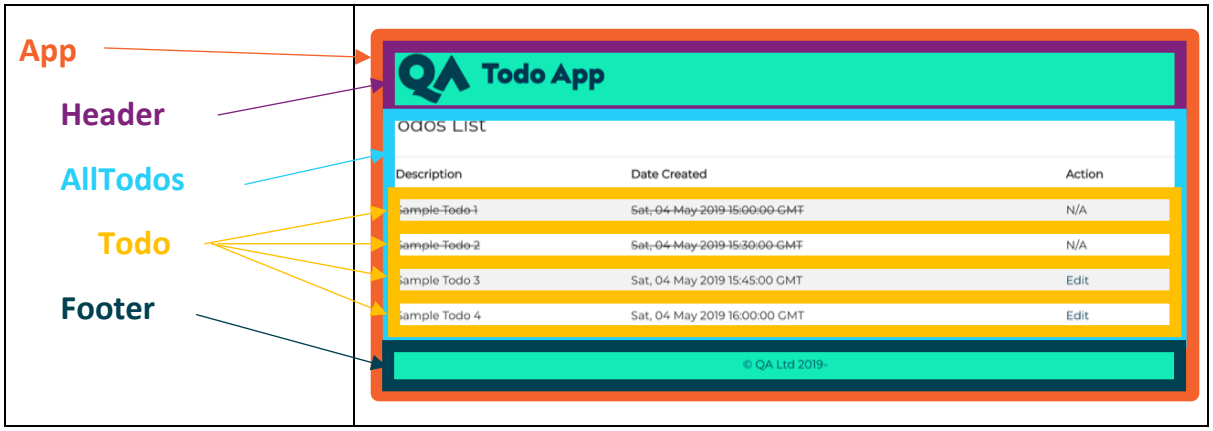
Add/Edit Todo



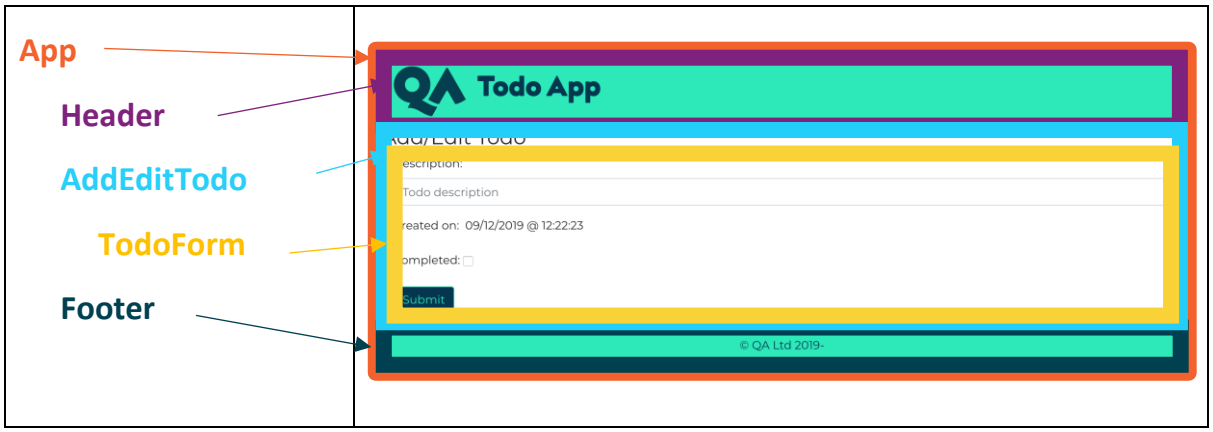
This is the end of Quick Lab 6

QuickLab 6 – Thinking in React – Part 1 – Component Hierarchy – Sample Solution

All Todos



Add/Edit Todo



Quick Lab 7a – Create Common Header and Footer Components

Objectives

- To be able to create static function components and integrate them into an application

Overview

In this QuickLab, you will create the components for the header and footer sections of the application. These components will be placed in the Components folder of the application and linked to the App component to display them. The acceptance criteria should be used as a guide to create the components in the first instance. A step-by-step guide is provided, as are the code snippets for reference if find that you aren't sure where to start.

Activity – Header and Footer Acceptance Criteria

Complete the section 'Before Each QuickLab' for b-static-version/starter before continuing.

You will find that this project has already been set up and imports Bootstrap (along with popper.js and jQuery) to enable a fully responsive application to be made. The logo can be found in the Components/images folder as an SVG and should be imported into the header. Additional CSS has been provided (along with branding fonts) and is imported into the App component so it is available anywhere in the component tree.

1. Use the acceptance criteria below to create Header and Footer components.

Footer	Header
Acceptance Criteria Delete 0% ☐ The UI should be created using ReactJS and JSX ☐ Every UI should have a <footer> ☐ The <footer> section should use the Bootstrap CSS classes to automatically set the top margin and left and right padding to 1rem ☐ The text in the footer should be centered using the "container" and "align-center" Bootstrap CSS classes You have unsaved edits on this field. View edits - Discard ☐ The 'footer' should contain the text '© QA Ltd 2019-'	Acceptance Criteria Delete 0% ☐ The UI should be created using ReactJS and JSX ☐ Every UI should have a <header> which wraps a <nav> with Bootstrap classes to set it as a navbar that is responsive and stacks the navigation on small screens ☐ The <nav> should contain the QA Logo having alt text of "QA Ltd" and a width of 100 ☐ The logo should be linked to https://www.qa.com, opening in a new window. The link should use Bootstrap class to signify that this is the brand and have rel properties of noopener and noreferrer. ☐ The <nav> section should contain the title 'Todo App' in a <h1> that is linked to / in an <a> that has a Bootstrap class to signify that this is for the brand

2. Import these into the App component to display them.

Desired Outcome

Activity – Header Stepped Instructions

3. In b-static-version/starter/src/Components create a new file called Header.jsx.
4. Add an `import` for `React` from `react`.
5. Add an `import` of `logo` from `'./images/qalogo.svg'`.
6. Create a Function component called `Header` that has *no parameters*.
7. The `return` of the component should have wrapping `<header>` and `<nav>` elements:
    - `<nav>` should have classes `navbar` `navbar-expand-sm`;
    - A link to https://www.qa.com with a class of `navbar-brand`, a target of `_blank` and a rel of `noreferrer`;
    - The link should contain an image whose src is `{logo}`, alt is `QA Ltd` and width is `100`;
    - A sibling link to `/` with a class of `navbar-brand` and text of `Todo App`.
8. `export` `Header` as `default`.
9. Save the file.
10. Open App.js from b-static-version/src for editing.
11. Add an `import` for `Header` from `./Components/Header`.
12. Within the outer `<div>`, add a child of `<Header />` as an older sibling of the inner `<div>`.
13. Save the file.

Activity – Footer Stepped Instructions

1. In b-static-version/starter/src/Components create a new file called Footer.jsx.
2. Add an `import` for `React` from `react`.
3. Create a Function component called Footer that has *no parameters*.

4. The `return` of the component should have wrapping `<footer>` element that:
    - Has Bootstrap classes of `mt-auto` (to set the top margins to automatic), `py-3` (to set padding left and right to 1rem), `text-center` and `container`;
    - Has text content of `©QA Ltd 2019-`.
5. `export` `Footer` as `default`.
6. Save the file.
7. Open **App.js** from **b-static-version/src** for editing.
8. Add an `import` for `Footer` from `./Components/Footer`.
9. Within the outer `<div>`, add a **child** of `<Footer />` as a **younger sibling** to the inner `<div>`.
10. Save the file.

Use `npm start` to run the application and check that the output is as shown in the desired outcome as above.

Code Snippets

Header.jsx

```
import React from 'react';                                          // 2
import logo from './images/qalogo.svg';                             // 3
const Header = () => {                                              // 4
  return (
    <header>                                                        // 5
      <nav className="navbar navbar-expand-sm">                     // 5.1
        <a                                                          // 5.2
          href="https://www.qa.com"
          className="navbar-brand"
          target="_blank"
          rel="noopener noreferrer"
        >
          <img                                                      // 5.3
            src={logo}
            alt="QA Ltd"
            style={{ width: '100px'}}
          />
        </a>
        <a className="navbar-brand" href="/">                       // 5.4
          <h1>Todo App</h1>
        </a>
      </nav>
    </header>
  );
};
export default Header;                                              // 6
```

Footer.jsx

```
import React from 'react';                                          // 2
const Footer = () => {                                              // 3
  return (                                                          // 4.1
    <footer className="mt-auto py-3 container text-center">
      &copy; QA Ltd 2019-                                           // 4.2
    </footer>
  );
};
export default Footer;                                              // 5
```

App.js

```
import React from 'react';
import 'bootstrap/dist/css/bootstrap.min.css';
import 'bootstrap';
import 'popper.js';
import 'jquery';
import './Components/css/qa.css';
import Header from './Components/Header';                           // H9
import Footer from './Components/Footer';                           // F8
function App() {
  return (
    <div className=container>
      <Header />                                                    // H10
      <div className=container><h1>Other UIs go here</h1></div>
      <Footer />                                                    // F9
    </div>
  );
};
export default App;
```

This is the end of Quick Lab 7a

Quick Lab 7b – Write Snapshot Tests

Objectives

- To understand how to write a simple snapshot test using react-test-renderer

Overview

In this QuickLab, you will add 2 tests to the Tests folder, one for the Header component and one for the Footer component. These should be simple snapshot tests as the components will not change much one development is complete. The tests should use the react-test-renderer npm package. This will need installing into the project. You will use the create function to create a version of the component and then expect the toJSON version toMatchSnapshot. The npm test -- --coverage command will be used to run the tests and create coverage reports.

Remember that the test may initially fail as there is no snapshot and that if you change the code in the component, you will need to update the snapshot to make the test pass again.

Activity – Test the Header

Skip the 'Before Each QuickLab' for this Activity and continue working in the b-static-version/starter folder.

1. Point a command line to the starter folder for this QuickLab and install the react-test-renderer as a development dependency:

```
npm i react-test-renderer --save-dev
```

2. In the tests folder, create a new file called Header.test.js.
3. `import` `React` from `react` and `{ create }` from `react-test-renderer`.
4. `import` the `Header` component from the Header file in the Components folder.
5. Write a `test` with the title `Header matches snapshot` and a callback function that:
   - Declares a `const` `header` set to the result of a call to `create`, passing `<Header />` as an argument
   - `expects` a call to `toJSON` on `header` `toMatchSnapshot`.
6. Save the file.
7. Check that the test runs and passes.

Activity – Test the Footer

1. Repeat the process above, substituting the `Header` for `Footer`.

Activity – Ensure the tests fail

1. Change some of the displayed text inside appropriate tags in both of the component files and save them.
2. Observe that the tests now fail – DO NOT UPDATE the snapshots.
3. Change the text back to the original and ensure that the tests pass again.

Code Snippets

Header.test.js

```
import React from 'react';                                          // 3
import { create } from 'react-test-renderer';                       // 3

import Header from '../Components/Header';                          // 4

test(`Header matches snapshot`, () => {                             // 5
  const header = create(<Header />);                                // 5.1
  expect(header.toJSON()).toMatchSnapshot();                        // 5.2
});
```

Footer.test.js

```
import React from 'react';                                          // 3
import { create } from 'react-test-renderer';                       // 3

import Footer from '../Components/Footer';                          // 4

test(`Footer matches snapshot`, () => {                             // 5
  const footer = create(<Footer />);                                // 5.1
  expect(footer.toJSON()).toMatchSnapshot();                        // 5.2
});
```

This is the end of Quick Lab 7b

Quick Lab 8 – Exploring Props

Objectives

- To be able to use props in a component
- To be able to define propTypes for a component's props and ensure that they are present if needed
- To be able to supply defaultProps for a component
- To be able to pass props to a child from its parent

Overview

In this QuickLab, you will create a component called ComponentWithProps that uses 4 props (header, content, number and nonexistent) to populate a header and 3 paragraphs in its return. This will be rendered by the MyComponent component. You will observe the output and then add PropTypes for header, content (both strings) and number that are all required. The browser output will be observed again. Next, defaultProps will be added for header, content and number before providing actual props for content and number when it is called in the render of MyComponent. Finally, you will inspect the browser output to ensure all warnings have been removed.

Activity – ComponentWithProps - Step-by-step

Skip the 'Before Each QuickLab' for this Activity and work in the a-react-app/starter project.

1. Create a new file called ComponentWithProps.jsx in a-react-app/starter/src.
2. Add an `import` for `React` from `react`.
3. Define a Function component called `ComponentWithProps` that has `props` as an argument and a `return` that has:
    - A wrapping React Fragment;
    - A `<h1>` that uses `header` from `props` as its content;
    - A `<p>` that uses `content` from `props` as its content;
    - A `<p>` that uses `number` from `props` as its content along with some text;
    - A `<p>` that uses `nonexistent` from `props` as its content along with some text.
4. `export` `ComponentWithProps` as `default`.
5. Save the file.
6. Open MyComponent.jsx for editing and `import` the *new component*.

7. *Add it* to the `return` but *DO NOT* supply any props at this point.
8. Ensure that you wrap the `return` of MyComponent in a React Fragment.
9. Save the file and run the application.

You should find that the application runs without errors (check the console), although there are empty elements where props were not found and spaces where props where included as part of other text.

Activity – Using PropTypes – Step-by-step

1. In ComponentWithProps.jsx, add an `import` for `PropTypes` from `prop-types`.
2. Before the `export` statement, add `ComponentWithProps.propTypes` and set it to an object.
   - Add keys of `header` and `content` with values that will ensure that both are required strings;
   - Add a key of `number` with a value that will ensure it is a required number.
3. Save the file and return to the browser console output.

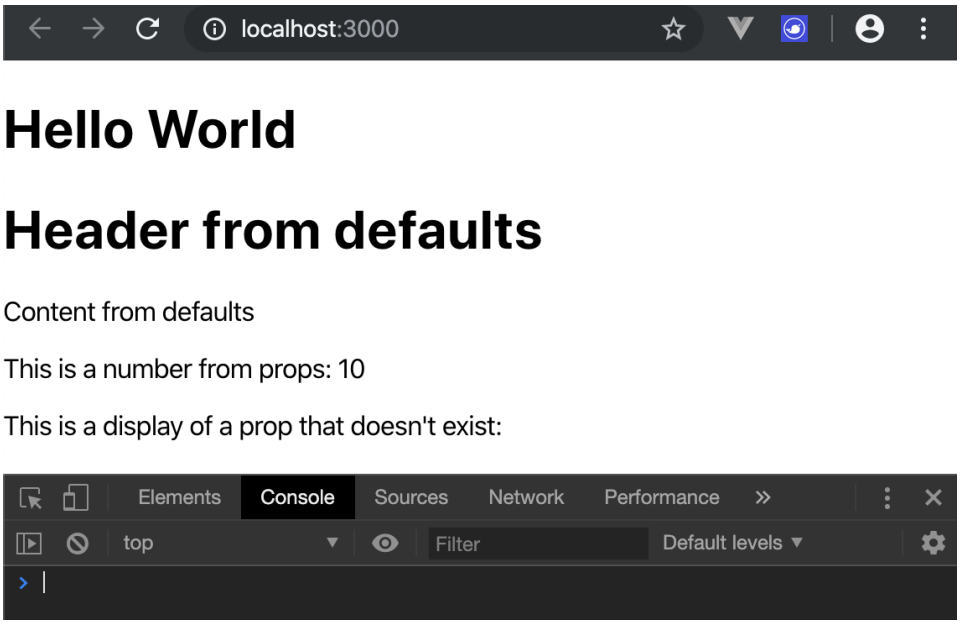
You should see that the application still renders the same but there are 3 warnings displayed on the console.

```
▶Warning: Failed prop type: The prop `header` is marked as        index.js:1375
required in `ComponentWithProps`, but its value is `undefined`.
    in ComponentWithProps (at MyComponent.jsx:8)
    in MyComponent (at App.js:6)
    in App (at src/index.js:7)
▶Warning: Failed prop type: The prop `content` is marked as       index.js:1375
required in `ComponentWithProps`, but its value is `undefined`.
    in ComponentWithProps (at MyComponent.jsx:8)
    in MyComponent (at App.js:6)
    in App (at src/index.js:7)
▶Warning: Failed prop type: The prop `number` is marked as        index.js:1375
required in `ComponentWithProps`, but its value is `undefined`.
    in ComponentWithProps (at MyComponent.jsx:8)
    in MyComponent (at App.js:6)
    in App (at src/index.js:7)
```

Activity – Using defaultProps – Step-by-step

1. Under the PropTypes defined in the last part, add a declaration for `ComponentWithProps.defaultProps` and set it to an object.
   - Add a key of `header` with value `` `Header from defaults` ``;
   - Add a key of `content` with value `` `Content from defaults` ``;
   - Add a key of `number` with a value of `100`.
2. Save the file and check the browser console output.

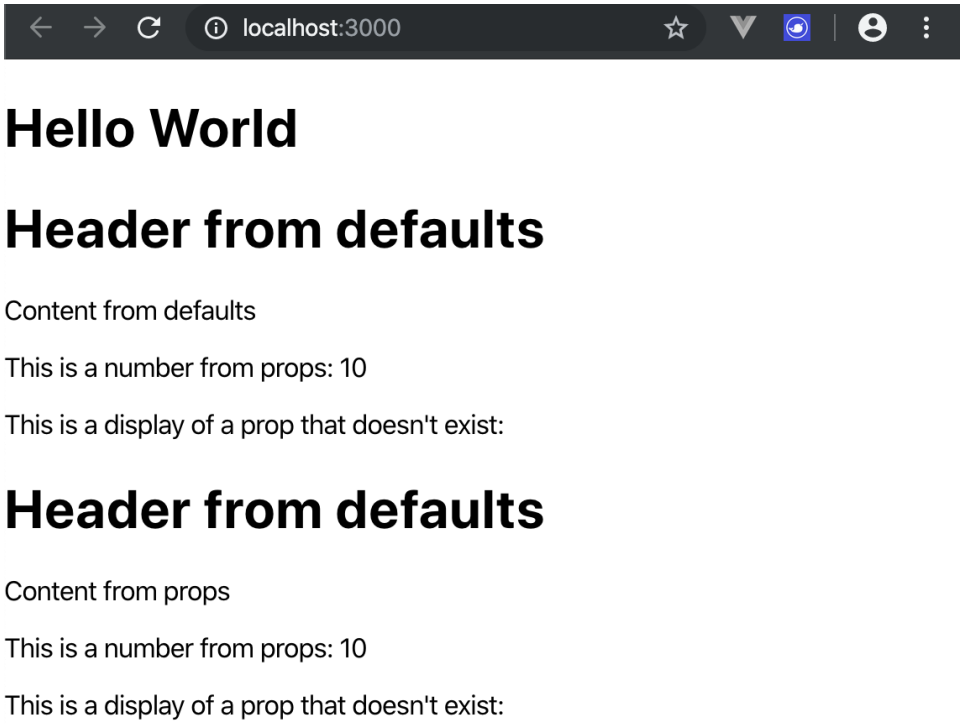
You should see that the warnings have disappeared, with values supplied as defaults displayed on the web page.



Activity – Supplying props from the Parent – Step-by-step

3. Open MyComponent.jsx for editing and add another ComponentWithProps to the `return`, supplying the following **props** (attributes):
    - `content` with a **value** of `"Content passed from props"`;
    - `number` evaluated in JavaScript to `10`.
4. Save the file and view the browser.

You should notice that the values displayed on the web page in the second rendering of ComponentWithProps match those to where they are picked up from in the code.


localhost:3000
Hello World
Header from defaults
Content from defaults
This is a number from props: 10
This is a display of a prop that doesn't exist:
Header from defaults
Content from props
This is a number from props: 10
This is a display of a prop that doesn't exist:

Code Snippets

ComponentWithProps.jsx

```
import React from 'react';                                          // CWP1
import PropTypes from 'prop-types';                                 // PT1

const ComponentWithProps = props => {                               // CWP2
  return (
    <>                                                              // CWP3.1
      <h1>{props.header}</h1>                                       // CWP3.2
      <p>{props.content}</p>                                        // CWP3.3
      <p>                                                           // CWP3.4
        This is a number from props:
        {props.number}
      </p>
      <p>                                                           // CWP3.5
        This is a display of a prop that
        doesn't exist: {props.nonexistent}
      </p>
    </>
  );
}

ComponentWithProps.propTypes = {                                    // PT2
  header:   PropTypes.string.isRequired,                            // PT2.1
  content:  PropTypes.string.isRequired,                            // PT2.1
  number:   PropTypes.number.isRequired                             // PT2.2
};

ComponentWithProps.defaultProps = {                                 // DP1
  header:   `Header from defaults`,                                 // DP1.2
  content:  `Content from defaults`,                                // DP1.3
  number:   10                                                      // DP1.3
};

export default ComponentWithProps;                                  // CWP4
```

MyComponent.jsx

```
import React from 'react';
import ComponentWithProps                                           // CWP6
from 'ComponentWithProps';

const MyComponent = () => {
  return (
    <>                                                              // CWP8
      <h1>Hello World</h1>
      <ComponentWithProps />                                        // CWP7
      <ComponentWithProps
        content="Content passed from props"                         // SP1.1
        number={10}                                                 // SP1.2
      />
    </>                                                             // CWP8
  );
};
```

This is the end of Quick Lab 8

Quick Lab 9 – Testing Props

Objectives

- To be able to use react-test-renderer to be able to test that supplied props are rendered in components

Overview

In this QuickLab, you will write 3 tests for the ComponentWithProps component. The tests will ensure that if one of the props that is rendered as part of the component is supplied, it is in fact rendered. The tests will use the create function supplied by react-test-renderer and generate a test instance of the component. Following this, the component will be interrogated using the testInstance functions findByType and findAllByType. Checking the children property of the 'rendered' instance will allow the value that will be displayed to be checked. These are tested as they affect the view that the user will see.

It is assumed that PropTypes and defaultProps that are supplied to the component are not the concern of the developer as these are part of React and are therefore assumed to be tested!

Activity – Tests Set Up

Skip the 'Before Each QuickLab' for this Activity and work in the a-react-app/starter project.

1. Point the command line to the starter folder and install react-test-renderer as a development dependency:

```
npm i --save-dev react-test-renderer
```

2. Add `.skip` to the test call (as shown below) in the App.test.js file to skip this test that will now fail due to the changes to App.js we have made previously.

```
test.skip(`…
```

Activity 1 – Test the header prop

1. Create a folder in src called tests and create a file called ComponentWithProps.test.js.
2. Add an `import` for `React` from `react`.
3. Add an `import` for `{ create }` from `react-test-renderer`.
4. Import `ComponentWithProps` from its file.
5. Define a `test` with the title of `` `it should render the correct heading from props when a header prop is supplied` `` and initialise an arrow function.

6. Populate the arrow function with code that:
    - Defines a `const` called `testHeader` set to a string with *some text*;
    - Defines a `const` called `testRenderer` set to a call to `create`, making a `ComponentWithProps` component that has a prop of `header` set to `testHeader`;
    - Defines a `const` called `testInstance` that returns the `root` test instance;
    - Asserts that the `children` of the `h1` found in `testInstance` contains `testHeader`.
7. Save the file and run the command `npm test` on the command line.

This test should pass – change the assertion to make it fail and then make it pass again.

Activity 2 – Test the content prop

1. Define a `test` with the title of `` `it should render the correct content from props when a content prop is supplied` `` and initialise an arrow function.
2. Populate the arrow function with code that:
    - Defines a `const` called `testContent` set to a string with *some text*;
    - Defines a `const` called `testRenderer` set to a `call` to create, making a `ComponentWithProps` component that has a prop of `content` set to `testContent`;
    - Defines a `const` called `testInstance` that returns the `root` test instance;
    - Defines a `const` called `renderedParagraphs` set to the result of `findingAllByType` using `` `p` `` as an argument on `testInstance`;
    - Asserts that the `children` of `renderedParagraphs` with index of `0` contains `testContent`.
3. Save the file.

This test should pass – change the assertion to make it fail and then make it pass again.

Activity 3 – Test the number prop

1. Define a `test` with the title of `` `it should render the correct number from props when a number prop is supplied` `` and initialise an arrow function.
2. Populate the arrow function with code that:
    - Defines a `const` called `testNumber` set to any number;

- Defines a `const` called `testRenderer` set to a call to `create`, making a `ComponentWithProps` component that has a prop of `number` set to `testNumber`;
- Defines a `const` called `testInstance` that returns the root test instance;
- Defines a `const` called `renderedParagraphs` set to the result of `findingAllByType` using `` `p` `` as an argument on `testInstance`;
- Asserts that the `children` of the `renderedParagraphs` array element that has an index of `1` contains `testNumber` explicitly converted to a string.

3. Save the file.

This test should pass – change the assertion to make it fail and then make it pass again.

Code Snippet

ComponentWithProps.test.js

```
import React from 'react';                                          // 1.2
import { create } from 'react-test-renderer';                       // 1.3
import ComponentWithProps from '../ComponentWithProps';             // 1.4

test(`it should render the correct heading from props when a header prop
is supplied`, () => {                                               // 1.5
  const testHeader = `Test Header`;                                 // 1.6
  const testRenderer = create(
    <ComponentWithProps header={testHeader} />);                    //
  const testInstance = testRenderer.root;                           //
  expect(testInstance.findByType(`h1`).children).                   //
  toContain(testHeader);
});

test(`it should render the correct content from props when a content prop
is supplied`, () => {                                               // 2.1

  const testContent = `Test Content`;                               // 2.2
  const testRenderer = create(
    <ComponentWithProps content={testContent} />);                  //
  const testInstance = testRenderer.root;                           //
  const renderedParagraphs = testInstance.findAllByType(`p`);       //
  expect(renderedParagraphs[0].children).                           //
  toContain(testContent);
});

test(`it should render the correct number from props when a number prop is
supplied`, () => {                                                  // 3.1

  const testNumber = 1000;                                          // 3.2
  const testRenderer = create(
    <ComponentWithProps number={testNumber} />);                    //
  const testInstance = testRenderer.root;                           //
  const renderedParagraphs = testInstance.findAllByType(`p`);

  expect(renderedParagraphs[1].children).                           //
  toContain(testNumber.toString());

});
```

This is the end of Quick Lab 9

Quick Lab 10a – Thinking in React Part 2 – A Static Version – Components with Static Data

Objectives

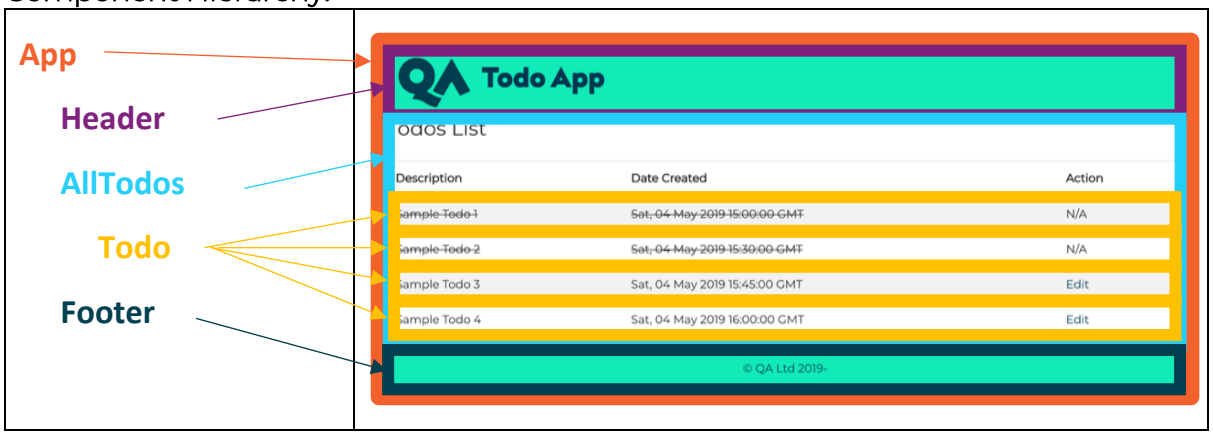
- To be able to use static external data to populate components
- To be able to use the map function to create multiple components
- To be able to conditionally render items dependent on some value

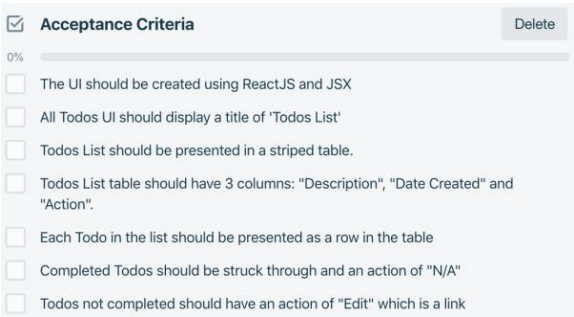
Overview

In this QuickLab, you will use the data supplied in the file src/sampleTodos.json to populate the AllTodos view. You should use the component hierarchy identified earlier (and shown below) and the Acceptance Criteria to produce the components needed for the AllTodos UI. A Todo model (basically a JavaScript class to define the shape of a Todo) has been defined in the ./utils folder for use with the instanceof PropTypes check.

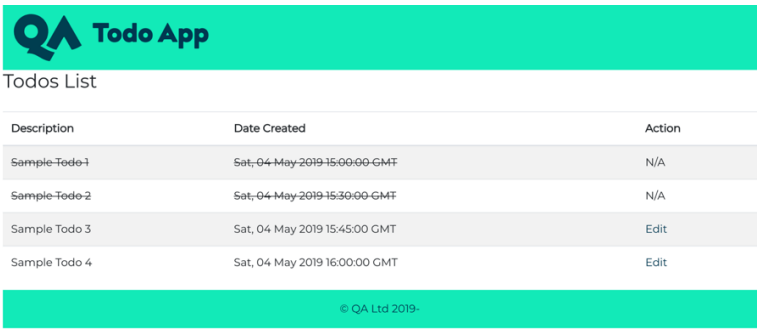
Continue working in the b-static-version/starter folder.

Component Hierarchy:





Desired Outcome



Activity 1 – The Todo component

Skip the 'Before Each QuickLab' for this Activity and continue working in the b-static-version/starter folder.

1. Create a new file in the src/Components folder called Todo.jsx.
2. Insert the *boilerplate code* for an *empty Function component* that receives a prop of `{todo}`.
3. Import `PropTypes` from `prop-types`.
4. Import `TodoModel` from `./utils/Todo.model`.
5. Set a `const` `dateCreated` to be a `new` `Date` that parses the `todo.todoDateCreated` and converts it to a UTC string.
6. Set a `const` `completedClassName` that is *conditionally set* to `` `completed` `` if `todo.todoCompleted` is true and an *empty string* if not.
7. Declare a variable `completed`.
8. Use an `if` statement to set `completed` to the string `` `N/A` `` if `todo.todoCompleted` is true and to the markup `<a href="/">Edit</a>` if not.
9. Return a table row that has *3 cells* whose *first 2* have a `className` set by `completedClassName` and whose content is the `todo.todoDescription` and `dateCreated` respectively. The *final cell* should render the `completed` variable.
10. Before the `export` statement add `Todo.propTypes` as an object that sets a key of `todo` to be a call to `instanceOf` on `PropTypes`, passing `TodoModel` as the argument.
11. Save the file.

Activity 2 – The AllTodos component

1. Create a new file in the src/Components folder called AllTodos.jsx.
2. Insert the boilerplate code for an empty Function component that receives *no props*.
3. Import the CSS file for AllTodos found in the css folder.

4. Import `sampleTodos` from the sampleTodos.json file.
5. Import `Todo` from the Todo file.
6. `TodoModel` should be imported from ./utils/Todo.model;
7. Inside the component function, set a `const` `todos` that maps the `sampleTodos` array with an arrow function that:
    - Takes `currentTodo` as an argument;
    - Has a line in the function body that creates a new `TodoModel` called `todo` by passing in the properties from `currentTodo` in the order description, date created, completed and _id into the TodoModel *constructor*;
    - Returns a `Todo` component with a property of `todo` set to the `todo` and a `key` of the *todo's _id*.
8. Make the component return a wrapping `div` with a `className` of `row` with:
    - A `h3` with text of `Todo List`;
    - A *sibling table* with `className`s `table` and `table-striped`;
    - A `thead` that has a *table row* that has the 3 headings Description, Date Created and Action;
    - A `tbody` that renders the array of `todos`.
9. Save the file.

Activity 3 – Render the AllTodos component

1. Open App.js.
2. Replace the placeholder text inside the inner `div` with the `className` container with an `AllTodos` component.
3. Save the file and fire up the application.

Code Snippets

Todo.jsx

```
import React from 'react';                                          // 1.2
import PropTypes from 'prop-types';                                 // 1.3
import TodoModel from './utils/Todo.model';                         // 1.4
const Todo = ({ todo }) => {                                        // 1.2
  const dateCreated =                                               // 1.5
  new Date(Date.parse(todo.todoDateCreated)).toUTCString();
  const completedClassName = todo.todoCompleted ? `completed` : ``; // 1.6
    let completed;                                                  // 1.7
    if (todo.todoCompleted) {                                       // 1.8
        completed = `N/A`                                           //
    } else {                                                        //
        completed = <a href="/">Edit</a>                            //
    }                                                               //
    return (                                                        // 1.9
        <tr>                                                        //
            <td className={completedClassName}>                     //
              {todo.todoDescription}                                //
            </td>                                                   //
            <td className={completedClassName}>                     //
              {dateCreated}                                         //
            </td>                                                   //
            <td>{completed}</td>                                    //
        </tr>                                                       //
    );                                                              //
};                                                                  //
Todo.propTypes = {                                                  // 1.10
  todo: PropTypes.instanceOf(TodoModel)
};

export default Todo;
```

AllTodos.jsx

```
import React from 'react';                                          // 2.2
import './css/AllTodos.css';                                        // 2.3
import sampleTodos from '../sampleTodos.json';                      // 2.4
import Todo from './Todo';                                          // 2.5
import TodoModel from './utils/Todo.model';                         // 2.6
const AllTodos = () => {                                            // 2.2
  const todos = sampleTodos.map(currentTodo => {                    // 2.7
    const todo = new TodoModel(currentTodo.todoDscription,          //
      currentTodo.todoDateCreated, currentTodo.todoCompleted,       //
      currentTodo._id);                                             //
    return <Todo todo={todo} key={todo._id} />});                   //
  return (                                                          // 2.8
    <div className="row">                                           //
      <h3>Todos List</h3>                                           //
      <table className="table table-striped">                      //
        <thead>                                                     //
          <tr>                                                      //
            <th>Description</th>                                    //
            <th>Date Created</th>                                   //
            <th>Action</th>                                         //
          </tr>                                                     //
        </thead>                                                    //
        <tbody>{todos}</tbody>                                      //
      </table>                                                      //
    </div>                                                          //
  );                                                                //
};

export default AllTodos;
```

App.jsx

```
import React from 'react';
import 'bootstrap/dist/css/bootstrap.min.css';
import 'bootstrap';
import 'popper.js';
import 'jquery';
import './Components/css/qa.css';

import Header from './Components/Header';
import Footer from './Components/Footer';
import AllTodos from './Components/AllTodos';

function App() {
  return (
    <div className="container">
      <Header />
      <div className="container">
        <AllTodos />                                                // 3.2
      </div>
      <Footer />
    </div>
  );
}

export default App;
```

This is the end of Quick Lab 10a

Quick Lab 10b – Thinking in React Part 2 – A Static Version – Testing Components with static data

Objectives

- To be able to test that a component renders the correct number of children
- To be able to test that a component renders conditional items correctly dependent on a prop

Overview

In this QuickLab, you will write tests for the components you have just made. The AllTodos component should be tested to ensure that the number of Todo components rendered is the same as the length of the static array that has been used to generate it. The Todo component is a little more convoluted, as there are 2 conditional statements that affect the output, both based on the todoCompleted status. You should test to see if the correct className is added to the Description and Date Created cells and also that the correct text is displayed in the Action cell.

Continue working in the b-static-version/starter folder.

Activity 1 – Write Tests for the AllTodos Component

1. In the src/tests folder, create a new file called AllTodos.test.js.
2. Import `React` from `react`, `{ create }` from `react-test-renderer`, `AllTodos` from ../Components/AllTodos and `sampleTodos` from ../sampleTodos.json.
3. Write a `test` with the title `` `it should render the correct number of Todo components based on the todo array supplied` `` and add the arrow function.
4. Populate the arrow function with:
    - A `const` called `sampleTodosLength` set to the `length` of `sampleTodos`;
    - Defines a `const` called `testRenderer` set to a call to `create`, making a `AllTodos` component that has no props;
    - Defines a `const` called `testInstance` that returns the root test instance;
    - Defines a `const` called `tableBody` set to the result of `findingByType` using `` `tbody` `` as an argument on `testInstance`;
    - Asserts that the length of the `children` of the `tableBody` array is the same as the `sampleTodosLength`.
5. Save the file and run the tests.

All tests should pass, including the new test on AllTodos. For peace of mind, change the value of sampleTodosLength to any value other than 4 (as this is the actual value) and check that it fails.

Activity 2 – Write Tests for the Todo Component - className

1. In the src/tests folder, create a new file called Todo.test.js.
2. Import `React` from `react`, `{ create }` from `react-test-renderer`, `AllTodos` from ../Components/AllTodos and `TodoModel` from ../Components/utils/Todo.model.
3. Create a mock of the `TodoModel` to return a `class` called `TodoModel` that has a `constructor` that sets:
   - `todoDescription` to `` `Test Todo` ``;
   - `todoDateCreated` to `` `2019-05-04T15:30:00.000Z` ``;
   - `todoCompleted` to `true`.

   Use the `jest.mock` function with a string argument of the *relative path to the component* and an arrow function that returns the above – *see the next page for the codeif you aren't sure as mocking has not been covered yet*!
4. Write a `test` with the title `` `it should render 2 tds with className completed if props.todo.todoCompleted is true` `` and add the arrow function.
5. Populate the arrow function with:
   - A `const` called `testTodo` set to be a new instance of `TodoModel`;
   - Defines a `const` called `testRenderer` set to a call to `create`, making a `Todo` component that has a prop `todo` set to `{testTodo}`;
   - Defines a `const` called `testInstance` that returns the root test instance;
   - Defines a `const` called `cells` set to the result of `findingAllByType` using `` `td` `` as an argument on `testInstance`;
   - Loops through the `cells` array (stopping one before the end of it) and each time asserting that the current cell's `props.className` array is `` `completed` ``.
6. Save the file and verify that the test passes.
7. Write a similar test that has a `testTodo.todoCompleted` property of `false` and checks to see if the `className` of the cells is Falsy.

Activity – Write Tests for the Todo Component – Action render

1. Write another `test` with the title `` `it should render `N/A` in the last td of the row if props.todo.todoCompleted is true` `` and

add the arrow function.

2. Populate the arrow function with:
    - A `const` called `testTodo` set to be a new instance of `TodoModel;`
    - Defines a `const` called `testRenderer` set to a call to `create`, making a `Todo` component that has a prop `todo` set to `{testTodo}`;
    - Defines a `const` called `testInstance` that returns the root test instance;
    - Defines a `const` called `cells` set to the result of `findingAllByType` using `` `td` `` as an argument on `testInstance`;
    - Assert that the last cell's `children` array contains `` `N/A` ``.
3. Save the file and verify that the test passes
4. Write a similar test that has a `testTodo.todoCompleted` property of `false` and checks to see if the `children` array of the cell contains `` `Edit` ``.

Code Snippets

AllTodos.test.js

```
import React from 'react';                                          // 1.2
import { create } from 'react-test-renderer';                       //
import AllTodos from '../Components/AllTodos';                      //
import sampleTodos from '../sampleTodos.json';                      //

test(`it should render the correct number of Todo components based on the
todo array supplied`, () => {                                       // 1.3

    const sampleTodosLength = sampleTodos.length;                   // 1.4
    const testRenderer = create(<AllTodos />);                      //
    const testInstance = testRenderer.root;                         //
    const tableBody = testInstance.findByType(`tbody`);             //
    expect(tableBody.children.length).toBe(sampleTodosLength);      //
});
```

Todo.test.js

```
import React from 'react';                                          // 2.2
import { create } from 'react-test-renderer';                       //
import Todo from '../Components/Todo';                              //
import TodoModel from '../Components/utils/Todo.model';             //

jest.mock("../Components/utils/Todo.model", () => {                 // 2.3
  return class TodoModel {
    constructor() {
      this.todoDescription = `Test Todo`;
      this.todoDateCreated = `2019-05-04T15:30:00.000Z`;
      this.todoCompleted = true;
    }
  };
});
test(`it should render 2 tds with className completed if
props.todo.todoCompleted is true`, () => {                          //2.4

  const testTodo = new TodoModel()                                  // 2.5
  const testRenderer = create(<Todo todo={testTodo} />);            //
```

```
  const testInstance = testRenderer.root;                          //
  const cells = testInstance.findAllByType(`td`);                  //
  for (let i = 0, j = cells.length - 1; i < j; i++) {              //
      expect(cells[i].props.className).toBe(`completed`);          //
    }                                                              //
});                                                                //
test(`it should render 2 tds with no className if props.todo.todoCompleted
is false`, () => {                                                 // 2.7
  const testTodo = new TodoModel();
  testTodo.todoCompleted = false;
  const testRenderer = create(<Todo todo={testTodo} />);
  const testInstance = testRenderer.root;
  const cells = testInstance.findAllByType(`td`);
  for (let i = 0, j = cells.length - 1; i < j; i++) {
        expect(cells[i].props.className).toBeFalsy();
    }
});

test(`it should render 'N/A' in the final td of the row if
props.todo.todoCompleted is true`, () => {                         // 3.1
  const testTodo = new TodoModel();                                // 3.2
  const testRenderer = create(<Todo todo={testTodo} />);           //
  const testInstance = testRenderer.root;                          //
  const cells = testInstance.findAllByType(`td`);                  //
  expect(cells[cells.length - 1].children).toContain(`N/A`);       //
});

test(`it should render 'Edit' in the final td of the row if
props.todo.todoCompleted is false`, () => {
  const testTodo = new TodoModel();                                // 3.4
  testTodo.todoCompleted = false;
  const testRenderer = create(<Todo todo={testTodo} />);
  const testInstance = testRenderer.root;
  const links = testInstance.findByType(`a`);
  expect(links.children).toContain(`Edit`);
});
```

This is the end of Quick Lab 10b

Quick Lab 10c – Thinking in React Part 2 – A Static Version – Adding a Form

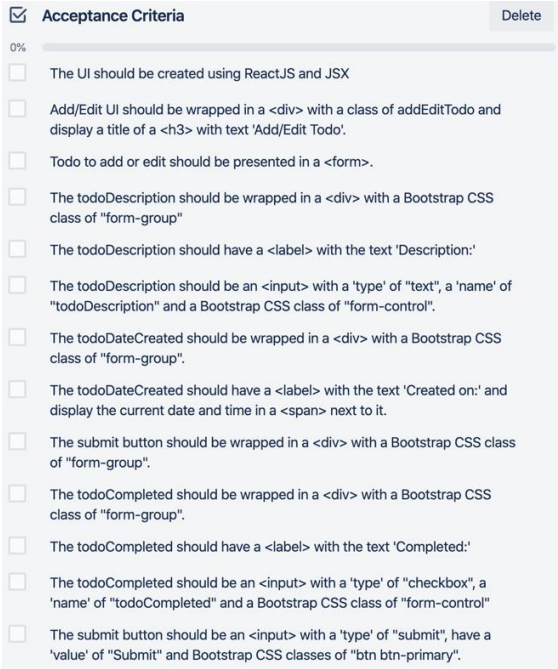
Objectives

- To be able to add a static, non-interactive form to an application

Overview

In this QuickLab, you will create the components needed to put the UI to add or edit a Todo into the application. Use the acceptance criteria and the mock-up provided to help. A TodoForm component will be created that allows the input of the todo's description, uses a supplied utility component called DateCreated (available in /Components/utils), provides a checkbox for the 'completed' status and a submit button. A wrapping AddEditTodo component will be created to provide the title and render the form and this will be added under the AllTodos component in the App component.

Continue working in the b-static-version/starter folder.

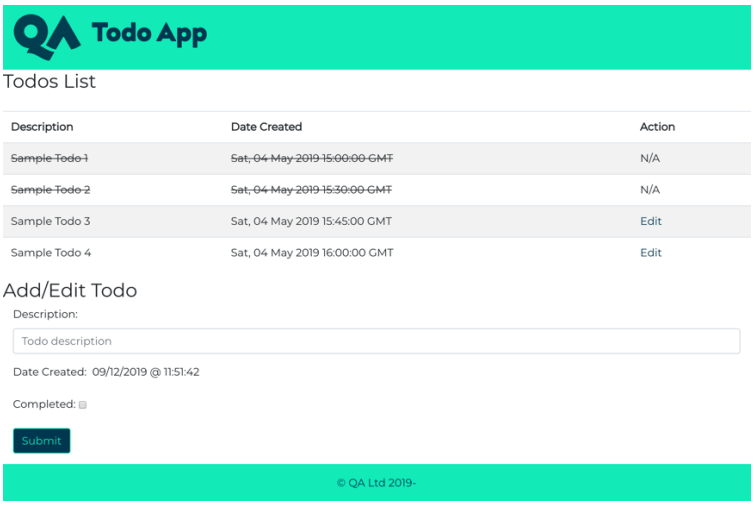


Acceptance Criteria Delete

0%

- [ ] The UI should be created using ReactJS and JSX
- [ ] Add/Edit UI should be wrapped in a with a class of addEditTodo and display a title of a <h3> with text 'Add/Edit Todo'.
- [ ] Todo to add or edit should be presented in a <form>.
- [ ] The todoDescription should be wrapped in a with a Bootstrap CSS class of "form-group"
- [ ] The todoDescription should have a <label> with the text 'Description:'
- [ ] The todoDescription should be an <input> with a 'type' of "text", a 'name' of "todoDescription" and a Bootstrap CSS class of "form-control".
- [ ] The todoDateCreated should be wrapped in a with a Bootstrap CSS class of "form-group".
- [ ] The todoDateCreated should have a <label> with the text 'Created on:' and display the current date and time in a next to it.
- [ ] The submit button should be wrapped in a with a Bootstrap CSS class of "form-group".
- [ ] The todoCompleted should be wrapped in a with a Bootstrap CSS class of "form-group".
- [ ] The todoCompleted should have a <label> with the text 'Completed:'
- [ ] The todoCompleted should be an <input> with a 'type' of "checkbox", a 'name' of "todoCompleted" and a Bootstrap CSS class of "form-control"
- [ ] The submit button should be an <input> with a 'type' of "submit", have a 'value' of "Submit" and Bootstrap CSS classes of "btn btn-primary".

Desired Outcome



Activity 1 – Create the TodoForm Component

1. In the **Components** folder, create a new file called **TodoForm.jsx**.
2. Add the boilerplate code to create a Functional component that does not receive any props.
3. Import `DateCreated` from **'./utils/DateCreated**.
4. Make the function **return** a wrapping **form** element that encloses:
    - A **div** with a **className** of `form-group` containing:
        - A **label** for `todoDescription` with the **content** of `Description: `
        - A **text input** with a **name** of `todoDescription`, a **placeholder** of `Todo Description` and a **className** of `form-control`.
    - A **div** with a **className** of `form-group` containing:
        - A **label** for `todoDateCreated` with the **content** of `Created on: `
        - A **DateCreated** component.
    - A **div** with a **className** of `form-group` containing:
        - A **label** for `todoCompleted` with the **content** of `Completed: `
        - A **checkbox input** with a **name** of `todoCompleted`.
    - A **div** with a **className** of `form-group` containing:
        - A **submit input** with a **value** of `Submit` and **classNames** `btn` and `btn-primary`.
5. Save the file.

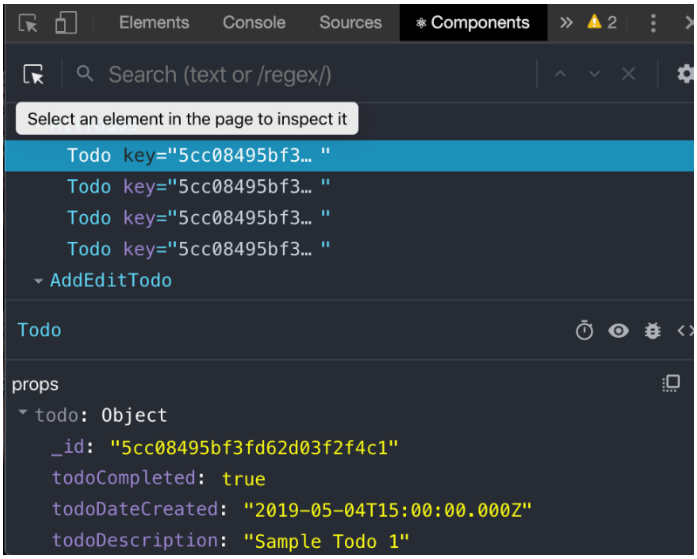
Activity 2 – Create the AddEditTodo Component

1. In the **Components** folder, create a new file called **AddEditTodo.jsx**.
2. Add the boilerplate code to create a Functional component that does not receive any props.
3. **Import AddEditTodo.css** from the appropriate path (**./css/AddEditTodo.css**).
4. The **return** of the function should be a wrapping **React.Fragment** that encloses:
    - A **div** with **classNames** of `addEditTodo` and `row` that wraps a **h3** with the content `Add/Edit Todo`;
    - A **TodoForm** component (imported from **./TodoForm**).
5. Save the File.

Activity 3 – Add the new components to the app

1. Open **App.js** for editing.
2. **Import** and then **add** the **AddEditTodo** component under the **AllTodos** component.
3. Save the file.

Launching the application in the browser should show the UI as shown in the desired outcome. Additionally, check that the 4 rendered Todo components prop values show in the Component section of the React Developer Tools:



Code Snippets

TodoForm.jsx

```
import React from 'react';                                          // 1.2
import DateCreated from './utils/DateCreated';                      // 1.3

const TodoForm = () => {                                            // 1.2
  return (                                                          // 1.4
    <form>                                                          //
      <div className="form-group">                                  //
        <label htmlFor="todoDescription">                           //
          Description:                                         //
        </label>                                                    //
        <input type="text" name="todoDescription"                  //
          placeholder="Todo description"                            //
          className="form-control" />                               //
      </div>                                                        //
      <div className="form-group">                                  //
        <label htmlFor="todoDateCreated">                           //
          Date Created:                                        //
        </label>                                                    //
        <DateCreated />                                             //
      </div>                                                        //
      <div className="form-group">                                  //
        <label htmlFor="todoCompleted">                             //
          Completed:                                           //
        </label>                                                    //
        <input type="checkbox" name="todoCompleted" />              //
      </div>                                                        //
      <div className="form-group">                                  //
        <input type="submit" value="Submit"                         //
         className="btn btn-primary" />                             //
      </div>                                                        //
    </form>                                                         //
  );                                                                //
};                                                                  // 1.2

export default TodoForm;                                            // 1.2
```

AddEditTodo.jsx

```
import React from 'react';                                          // 2.2
import './css/AddEditTodo.css';                                     // 2.3
import TodoForm from './TodoForm';                                  // 2.4

const AddEditTodo = () => {                                         // 2.2
  return (                                                          // 2.4
    <>                                                              //
      <div className="addEditTodo row">                             //
        <h3>Add/Edit Todo</h3>                                      //
      </div>                                                        //
      <TodoForm />                                                  //
    </>                                                             //
  );                                                                //
}                                                                   // 2.2

export default AddEditTodo;                                         // 2.2
```

App.js

```
import React from 'react';
import 'bootstrap/dist/css/bootstrap.min.css';
import 'bootstrap';
import 'popper.js';
import 'jquery';
import './Components/css/qa.css';

import Header from './Components/Header';
import Footer from './Components/Footer';
import AllTodos from './Components/AllTodos';
import AddEditTodo from './Components/AddEditTodo';                // 3.2

function App() {
  return (
    <div className="container">
      <Header />
      <div className="container">
        <AllTodos />
        <AddEditTodo />                                            // 3.2
      </div>
      <Footer />
    </div>
  );
}

export default App;.
```

This is the end of Quick Lab 10c

Quick Lab 10d – Thinking in React Part 2 – A Static Version – Test the Form rendering

Objectives

- To be able to make mock components to ensure correct rendering

Overview

In this QuickLab, you will create a test file for the TodoForm component. It has a sub-component of DateCreated, so you will define a mock for this component and then check to see if the mock is rendered. The mock should return a function that renders a span with a testid property set to dateCreated. The content of the span should be Date Created Component. The test should assert that the span with a prop of testid contains this text.

Continue working in the b-static-version/starter folder.

Activity

1. Add a new file called TodoForm.test.js to the tests folder.
2. Import `React` from `react` and `{ create }` from `react-test-renderer` along with the `TodoForm` component.
3. Provide a `jest.mock` implementation for the `DateCreated` component that can be found in the folder /src/Components/utils/.
4. The callback for the mock should return a `function` called `MockDateCreated`.
5. This function should return a `<span>` with an attribute of `testid` set to `dateCreated` and content of `Date Created Component`.
6. Create a test suite with the title `TodoForm test suite`.
7. Nest a test suite inside this with the title `DateCreated function and render tests`.
8. Create a `test` inside this that:
    - Uses the `create` function to make a `testRenderer` using `TodoForm`;
    - Makes a `testInstance` from the `testRenderer` `root`;
    - Uses the `find()` function, called on `testInstance`, to define a `dateCreated` that has an argument of an arrow function:
    - It receives an argument of `el`;
    - Returns if the `type` property of `el` is `span` AND `el.props.testid` is `dateCreated`

- Expects `dateCreated` to be truthy;
- Expects the `children` of `dateCreated` to contain the text `Date Created Component`.

9. Save the file and run the tests.

There should be no test failures.

If you have time:

- Write tests for the AddEditTodo component, checking that the correct classNames are rendered in the div and that it does actually render a form;
- Review the additional tests provided for App.js.

Code Snippets

TodoForm.test.js

```
import React from 'react';                                          // 2
import { create } from 'react-test-renderer';                       // 2
import TodoForm from '../Components/TodoForm';                      // 2

// Provide mock implementation for DateCreated component

jest.mock("../Components/utils/DateCreated", () => {                // 3
  return function MockDateCreated() {                               // 4
    return <span testid="dateCreated">                              // 5
             Date Created Component                                 //
           </span>                                                  //
  }                                                                 //
});                                                                 //

describe(`TodoForm test suite`, () => {                             // 6
  describe(`DateCreated function and render tests`, () => {         // 7
    test(`it should render a DateCreated component a date`, () => {  // 8
      const testRenderer = create(<TodoForm />);                      //
      const testInstance = testRenderer.root;                         //
      const dateCreated = testInstance.find(                          //
        el => el.type === `span` &&                                   //
              el.props.testid === `dateCreated`                       //
      );                                                              //
      expect(dateCreated).toBeTruthy();                               //
      expect(dateCreated.children).                                   //
        toContain(`Date Created Component`)                           //
    });                                                               //
  });                                                                 //
});
```

This is the end of Quick Lab 10d

Quick Lab 11 – Thinking in React Part 3 – Identifying State in the Todo Application

Objectives

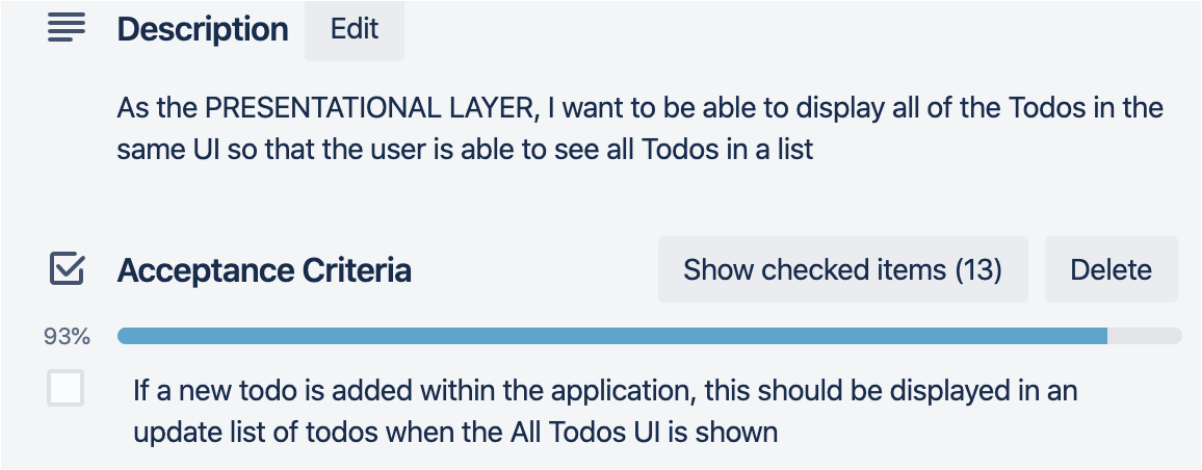
- To be able to make decisions about what should and should not be state in an application.

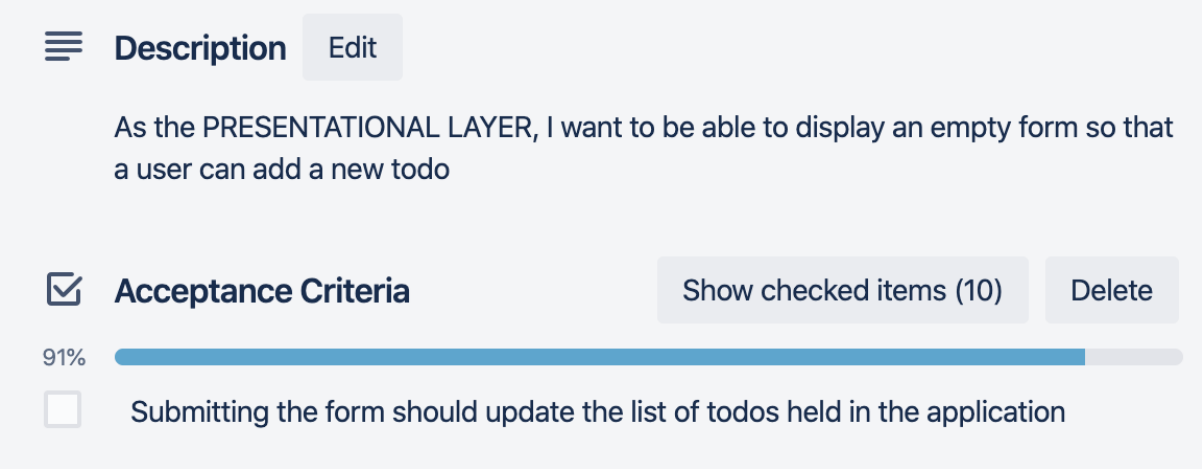
Activity

Consider the following data that is needed in the application and decide whether it should be state or not:

- The List of Todos;
- The Description property of a NEW todo;
- The Date Created property of a NEW todo;
- The Completed property of a NEW todo.

The Acceptance Criteria should be considered:





This is the end of Quick Lab 11

Quick Lab 11 – Thinking in React Part 3 – Identifying State in the Todo Application

EXAMPLE SOLUTION

Note that this is not necessarily the 'right' answer, it is one of several possibilities:

List of Todos

Does it remain unchanged over time?

According to the new Acceptance Criteria, this list may change as a result of the user adding new todos when they are using the application. On this basis, this data needs to be held in state at this point in the application development.

It is not passed in from a parent via props and it cannot be computed based on any other state or props.

The Description property of a new Todo

Does it remain unchanged over time?

As the description of a new Todo is provided by user input on the form, this is likely to change over time as the user will input text on the form. On this basis, this data needs to be held in state so that it is available for the process of adding a new Todo.

It is not passed in from a parent via props and it cannot be computed based on any other state or props.

The Date Created property of a new Todo

Does it remain unchanged over time?

As the value of time increments as the application is used and the time is required for recording when the todo was created, this needs to be held in state so that the most current value can be used when submitting a new todo.

It is not passed in from a parent via props and it cannot be computed based on any other state or props.

The Completed property of a new Todo

Does it remain unchanged over time?

All new todos to the list will start with a false completed property when they are created. As the application currently has no mechanism to update/edit a todo, this property will not change over time and therefore does not need to be stored in state as a result of this question.

Essentially, it can be computed based on other data within the application - in this case it will be a constant of false!

Quick Lab 12 – Thinking in React Part 4 – Identifying where State should live in the Todo Application

Objectives

- To be able to place the identified state in an application in an appropriate component.

Activity – List of todos

Identify where the list of todos should live as state in the application.

To work out where state should live:

- Identify every component that renders something based on the list of todos;
- Find common owner component of all of the components;
    - Either common component or component even higher up should own state;
- If no component makes sense, create new component to hold state and add it into the hierarchy above the common owner component.

Activity – All properties of a 'new' or 'updated' Todo

Identify where the all of the properties of a 'new' or 'updated' Todo should live as state in the application.

To work out where state should live:

- Identify every component that renders something based on the individual properties of a 'new' or 'updated' Todo;
- Find common owner component of all of the components;
    - Either common component or component even higher up should own state;
- If no component makes sense, create new component to hold state and add it into the hierarchy above the common owner component.

This is the end of Quick Lab 12

Quick Lab 12 – Thinking in React Part 4 – Identifying where State should live in the Todo Application

Example Solution

List of todos

The list of todos needs to be accessed by the AllTodos component but it also needs to be updated as a result of the AddEditTodo component. This strongly suggest that the list of todos should live in a common parent component and in the hierarchy of this application, that is seemingly the App component. The todos can then be passed to the AllTodos as props and when the list is updated, the application will update as a result of this state changing. The functionality for updating the list is not considered yet.

New Todo

As data for the new or updated todo comes from the TodoForm, there needs to be state within the TodoForm for the Todo Description and Todo Date Created and Todo Completed. *We are not dealing with the submission of the todo at this point, so functionality to pass the data back to the parent component is not considered*. Each todo property's state will be updated by the use of an onChange event trigger for the Description or Completed state and an updateDateCreated trigger on the DateCreated component.

You will notice that the DateCreated component also holds internal state for its functionality.

QuickLab 13a – Thinking In React Part 4 – Adding State

Objectives

- To be able to add state to components and pass it as props to child components
- To be able to update state using events and test that the state change updates the render

Overview

In this QuickLab, you will add **state** to the **App** and **TodoForm** components and modify the application to pass the state down as props where it is needed. The import of **sampleTodos** will need to be moved to the **App** component and then set as the *initial state* for **todos**. This state will then need passing to the **AllTodos** component, removing some of the technical debt introduced through using static data straight into this component. The *values* on the **TodoForm** will also be tracked by adding **state** to this component and using the **onChange** event to update state.

Once this has been completed, the application will be tested again to ensure that all tests pass, fixing the tests where necessary to reflect changes made.

Note: The submission of the **TodoForm** is not covered in this QuickLab. It is covered in Part 5 of Thinking in React.

Remember to refer back to the Acceptance Criteria for this part of the application too.

You may either continue in the **b-static-version/starter** folder *OR* if you did not complete all parts of that, you can pick up with the **c-stateful-version/starter** folder. Remember, if you choose the latter, you will need to run **npm install** and start the application.

Activity – Add state to the App Component

1. Open **App.js** for editing from your project folder.
2. Add `{ useState }` to the imports from `react`.
3. Add an import of `sampleTodos` from **./sampleTodos.json**.
4. In the `App` function, *before the return*, add the line introducing a **state** called `todos` with an initial value of an *object* that has a **key** of `todos` and a **value** of `sampleTodos`.
5. In the **return**, add a **prop** to the **AllTodos** component called `data` set to `{{todos}}`.
6. Save the file.

Activity – Make AllTodos use the todos from props

1. Open Components/AllTodos.jsx for editing.
2. *Remove* the import for `sampleTodos`.
3. Allow AllTodos to receive a prop of `data` by adding this to the *arrow function* arguments.
4. Modify the `map` function to use `data.todos` rather than `sampleTodos`.
5. Ensure that the `data` prop is check through `PropTypes.` It needs to:
   - Have a key of `data` that is an exact *object shape* of:
     - A key of `todos` that is an array of an exact *object shape* that has the *4 properties of a todo* listed with the *correct types* associated to each key.

   Don't forget to import the PropTypes symbol.
6. Save the file.
7. Run the tests for this project.

You should find that the test for AllTodos now fails. This is because AllTodos now expects to receive a prop of data and the create function doesn't provide this.

8. Open tests/AllTodos.test.js for editing.
9. In the `create` function that renders the AllTodo component, add a property of `data` set to an object with a key of `todos` and a value of `sampleTodos`.
10. Save the file and run the tests again.

You should find that the test passes as a result of the change.

Activity – Add state to the TodoForm Component

1. Open TodoForm.js for editing from your project Components folder.
2. Add `{ useState }` to the imports from react.
3. In the function, add state for:
   - `todoDescription`, initially setting it as an *empty string*;
   - `todoDateCreated`, initially setting it to `null`;
   - `todoCompleted`, initially setting it to `false`.
4. In the form, find the input for `todoDescription` and add properties:
   - `value` set to `todoDescription`;
   - `onChange` set to an arrow function that receives `event` and calls `setTodoDescription` with `event.target.value`.

5. Find the `DateCreated` component and add a property:
    - `updateDateCreated` set to an arrow function that receives `dateCreated` and calls `setTodoDateCreated` with it.
6. Find the input for `todoCompleted` and add properties:
    - `checked` set to `todoCompleted`;
    - `onChange` set to an arrow function that receives `event` and calls `setTodoCompleted` with `event.target.checked`.
7. Find the `input` for `submit` and:
    - Add a `disabled` property that is set to `!todoDescription`;
    - Set the input's `className` to `btn` and conditionally append either `btn-primary` when enabled or `btn-danger` when disabled – you should use a ternary and the `todoDescription` property.
8. Save the files and run the application.

You should find that the application still works. Additionally, the Component React Developer tools should show:

- The value of state in the App component as an Array of Objects, which when inspected are the 4 todos from the sampleTodos.json file
- The values of props in the AllTodos component – namely todos that is set to the same array as the state in the App component!
    - The props values of the 4 rendered Todo components are still there also.
- The values of state in the TodoForm component, these should update as you enter data in the input, the passage of time and changing the value of the checkbox.
- Run the test specs again and check that all still pass.

Code Snippets

App.jsx

```
import React, { useState } from 'react';                          // 2
import 'bootstrap/dist/css/bootstrap.min.css';
import 'bootstrap';
import 'popper.js';
import 'jquery';
import './Components/css/qa.css';
import sampleTodos from './sampleTodos.json';                     // 3
import Header from './Components/Header';
import Footer from './Components/Footer';
import AllTodos from './Components/AllTodos';
import AddEditTodo from './Components/AddEditTodo';

function App() {
  const [todos, setTodos] = useState(sampleTodos);                // 4
  return (
    <div className="container">
      <Header />
```

```
      <div className="container">
        <AllTodos data={{todos}} />                                  // 5
        <AddEditTodo />
      </div>
      <Footer />
    </div>
  );
}

export default App;
```

Components/AllTodos.jsx

```
import React from 'react';
import './css/AllTodos.css';
import Todo from './Todo';
import TodoModel from './utils/Todo.model';

const AllTodos = ({data}) => {                                       // 3
    const todos = data.todos.map(currentTodo => {                    // 4
        const todo =
          new TodoModel(
            currentTodo.todoDescription,
            currentTodo.todoDateCreated,
            currentTodo.todoCompleted,
            currentTodo._id);
        return <Todo todo={todo} key={todo._id} />
    });
    return (
        <div className="row">
            <h3>Todos List</h3>
            <table className="table table-striped">
                <thead>
                    <tr>
                        <th>Description</th>
                        <th>Date Created</th>
                        <th>Action</th>
                    </tr>
                </thead>
                <tbody>{todos}</tbody>
            </table>
        </div>
    );
};
AllTodos.propTypes = {
  data: PropTypes.exact({
    todos: PropTypes.arrayOf(
      PropTypes.exact({
        _id: PropTypes.string,
        todoDescription: PropTypes.string,
        todoDateCreated: PropTypes.string,
        todoCompleted: PropTypes.bool
      })
    )
  })
};
export default AllTodos;
```

tests/AllTodos.test.js

```
import React from 'react';
import { create } from 'react-test-renderer';
import AllTodos from '../Components/AllTodos';
import sampleTodos from '../sampleTodos.json';

test(`it should render the correct number of Todo components based on the
todo array supplied`, () => {
    const sampleTodosLength = sampleTodos.length;
```

```
    const testRenderer =
      create(<AllTodos data={{todos: sampleTodos}} />);           // 8
    const testInstance = testRenderer.root;
    const tableBody = testInstance.findByType(`tbody`);
    expect(tableBody.children.length).toBe(sampleTodosLength);

});
```

Components/TodoForm.jsx

```
import React, { useState } from 'react';                        // 2
import DateCreated from './utils/DateCreated';

const TodoForm = () => {
  const [todoDescription, setTodoDescription] = useState(``);   // 3
  const [todoDateCreated, setTodoDateCreated] = useState(null); // 3
  const [todoCompleted, setTodoCompleted] = useState(false);    // 3
  return (
    <form>
      <div className="form-group">
        <label htmlFor="todoDescription">Description: </label>
        <input
          type="text"
          name="todoDescription"
          placeholder="Todo description"
          className="form-control"
          value={todoDescription}                               // 4
          onChange={event =>
            setTodoDescription(event.target.value)}
        />
      </div>
      <div className="form-group">
        <label htmlFor="todoDateCreated">
          Created on: 
        </label>
        <DateCreated
          updateDateCreated={dateCreated =>                     // 5
            setTodoDateCreated(dateCreated)
          }
        />
      </div>
      <div className="form-group">
        <label htmlFor="todoCompleted">Completed: </label>
        <input
          type="checkbox"
          name="todoCompleted"
          checked={todoCompleted}                               // 6
          onChange={event =>
            setTodoCompleted(event.target.checked)}
        />
      </div>
      <div className="form-group">
        <input
          type="submit"
          value="Submit"
          className={`btn ${!todoDescription ? `btn-danger` : `btn-
primary`}`}
          disabled={!todoDescription}
        />
      </div>
    </form>
  );
};
export default TodoForm;
```

This is the end of Quick Lab 13a

QuickLab 13b – Thinking In React Part 4 – Testing Event Handlers

Objectives

- To be able to test that event handlers update the component using the act function

Overview

In this QuickLab, you will test that the onChange handlers for the inputs update the state in the TodoForm component. You will need to add to the TodoForm test suite with a suite of onChange event tests. There needs to be a test that checks that the value of the description input changes to the specified value when its onChange is called and a test that checks the value of the checkbox for todoCompleted changes when toggled.

It is worth noting that the todoDateCreated update is not tested here. That is because the change is tested within the DateCreated tests. The updating of state in this component is trusted to the ReactJS library testing.

Continue working in the c-stateful-version/starter folder.

Activity – Test the onChange event on the todoDescription input

1. Open tests/TodoForm.test.js for editing
2. Under the DateCreated render tests suite, add another suite called `onChange event tests`.
3. Add a test with a title of `` `it should render the new value in the input when todoDescription onChange is activated` `` and an arrow function that:
    - Sets a `const` `testValue` to the string `Test`;
    - Sets a `const` `testRenderer` to a call to the `create` function using `<TodoForm />` as an argument;
    - Sets a `const` `testInstance` to be `testRenderer.root`;
    - Uses `findByProps` called on `testInstance`, with an argument of `{name: "todoDescription"}`, to set a `const` `descInput`;
    - Makes an initial assertion that the `value` `prop` of `descInput` is an *empty string* using the `toBe` matcher.
    - Calls `act` with an argument of an arrow function whose body calls `props.onChange` on `descInput` with an argument of an object with a key of `target` and a value of an object with a key of `value` and a value of `testValue`;
    - Make a final assertion that the `value` `prop` of `descInput` is

`testValue`. using the `toBe` matcher.

4. Save the test file and run it to make sure it passes

Create a second test here that changes the value of checked on the `todoCompleted` input.

Finally, add a test that checks that the submit button's disabled property changes from true to false when the description field is populated.

Code Snippets

TodoForm.test.js

```
import React from 'react';
import { create, act } from 'react-test-renderer';

import TodoForm from '../Components/TodoForm';

jest.mock("../Components/utils/DateCreated", () => {
  return function MockDateCreated() {
    return <span testid="dateCreated">Date Created Component</span>
  }
});

describe(`TodoForm test suite`, () => {
  describe(`DateCreated function and render tests`, () => {
    test(`it should render a DateCreated component a date`, () => {
      const testRenderer = create(<TodoForm />);
      const testInstance = testRenderer.root;
      const dateCreated = testInstance.find(
        el => el.type === `span` && el.props.testid === `dateCreated`
      );
      expect(dateCreated).toBeTruthy();
      expect(dateCreated.children).toContain(`Date Created Component`);
    });
  });

  describe(`onChange event tests`, () => {
    test(`it should render the new value in the input when the
todoDescription onChange function is activated`, () => {
      const testValue = `Test`;
      const testRenderer = create(<TodoForm />);
      const testInstance = testRenderer.root;

      const descInput = testInstance.findByProps({ name: "todoDescription"
});

      expect(descInput.props.value).toBe(``);

      act(() => descInput.props.onChange({ target: { value: testValue }
}));

      expect(descInput.props.value).toBe(testValue);
    });

    test(`it should render the new value in the checkbox when the
todoCompleted onChange function is activated`, () => {
      const testValue = true;
      const testRenderer = create(<TodoForm />);
      const testInstance = testRenderer.root;
      const completedInput = testInstance.findByProps({ name:
"todoCompleted" });

      expect(completedInput.props.checked).toEqual(false);

      act(() => completedInput.props.onChange({ target: { checked:
```

```
testValue } }));

      expect(completedInput.props.checked).toBe(testValue);
    });
  });

  test(`should enable the submit button when the todo description is
populated`, () => {

    const testValue = `Test`;
    const testRenderer = create(<TodoForm />);
    const testInstance = testRenderer.root;

    const descInput = testInstance.findByProps({name: "todoDescription"});

    const submitBtn = testInstance.findByProps({type: `submit`});

    expect(submitBtn.props.disabled).toBe(true);

    act(() => descInput.props.onChange({ target: { value: testValue } }));

    expect(submitBtn.props.disabled).toBe(false);
    expect(submitBtn.props.className).toContain(`btn-primary`);

  });
});
```

This is the end of Quick Lab 13b

QuickLab 14a – Thinking In React Part 5 – Adding Inverse Data Flow

Objectives

- To be able to add inverse data flow to an application by passing handler functions by props

Overview

In this QuickLab, you will add inverse data flow to the application and ultimately update the array of todos in the App component. To start this process, in TodoForm, you will allow the user to submit the form, triggering a *submit handler function* on it. This will stop the default action from happening, call a function that will be passed in through its props with the todo values and then reset them to *empty values*.

Moving up the chain to the AddEditTodo component, you will create a function to pass to the TodoForm. This function will take the data for a new (or edited) Todo and generate an ID for it (using the provided utils/generateId function). This will create a new todo and call the function this component receives through props with it.

The top of this inverse data flow is the App component. You will define a submit handling function that receives the new Todo and adds it to the array of todos it already has. This will trigger a re-render of all components that depend on this.

Remember to refer back to the Acceptance Criteria for this part of the application too.

You may continue in the c-stateful-version/starter folder or extend the ql13-solution.

Activity – Add a submit handler to TodoForm

1. Open TodoForm.jsx for editing.
2. Import `PropTypes` from `prop-types`.
3. At the bottom of the file, before the export statement, declare `TodoForm.propTypes` to be an *object*, it should have a key of `submitTodo` and a value of `PropTypes.func.isRequired`.
4. Make this component receive `props` by adding it to the arguments of the function.
5. In the TodoForm function, under the state declarations, add an *arrow function* called `handleSubmit` that takes an argument of `event`. The arrow function body should:
    - Call `preventDefault` on `event`;
    - Call `props.submitTodo` with the *3 state values*;

- Set each of the **state** back to the *default values* initially provided.

6. In the `form` tag, add an attribute of `onSubmit` set to `handleSubmit`.
7. Save the file.

Activity – Add a submit handler to AddEditTodo

8. Open **AddEditTodo.jsx** for editing.
9. Import `PropTypes` from `prop-types`.
10. At the bottom of the file, before the **export** statement, declare `AddEditTodo.propTypes` to be an *object*, it should have a **key** of `submitTodo` and a **value** of `PropTypes.func.isRequired`.
11. Import `generateTodoId` from `'./utils/generateId'`.
12. Import `TodoModel` from `'./utils/Todo.model'`.
13. Make this component receive `props` by adding it to the arguments of the function.
14. Add an *arrow function* called `submitTodo` to the component, it should:
    - Receive `todoDescription`, `todoDateCreated` and `todoCompleted`;
    - Define a `const _id` that is the result of a call to `generateTodoId`;
    - Define a `const` called `newTodo` that calls the **constructor** for `TodoModel` with `todoDescription`, `todoDateCreated` converted to an *ISO String IF it exists*, `todoCompleted` and `_id`;
    - Call `props.submitTodo` with the `newTodo`.
15. In the render of the function add an **attribute** of `submitTodo` to the **TodoForm** component with a value of `submitTodo`.
16. Save the file.

Activity – Add a submit handler to App

1. Open **App.js** for editing.
2. Under the **state** declaration, add an *arrow function* called `submitTodo`, it should:
    - Receive `todo`;
    - Declare a `const` called `updatedTodos`, defined as an **array** that has a ***spread*** of `todos.todos` as its first element and `todo` as its second element;
    - Call `setTodos` with `updatedTodos`.
3. In the render of the function, add an **attribute** of `submitTodo` to the **AddEditTodo** component with a value of `submitTodo`.

4. Save the file.

Run the application and check that adding a new **todo** updates the application.

Code Snippets

TodoForm.jsx

```
import React, { useState } from 'react';
import PropTypes from 'prop-types';
import DateCreated from './utils/DateCreated';

const TodoForm = props => {

  const [todoDescription, setTodoDescription] = useState(``);
  const [todoDateCreated, setTodoDateCreated] = useState(null);
  const [todoCompleted, setTodoCompleted] = useState(false);

  const handleSubmit = event => {
    event.preventDefault();
    props.submitTodo(todoDescription, todoDateCreated, todoCompleted);
    setTodoDescription(``);
    setTodoDateCreated(null);
    setTodoCompleted(false);
  }

  return (
    <form onSubmit={handleSubmit}>
      <div className="form-group">
        <label htmlFor="todoDescription">Description: </label>
        <input
          type="text"
          name="todoDescription"
          placeholder="Todo description"
          className="form-control"
          value={todoDescription}
          onChange={event => setTodoDescription(event.target.value)}
        />
      </div>
      <div className="form-group">
        <label htmlFor="todoDateCreated">Created on: </label>
        <DateCreated updateDateCreated={dateCreated =>
           setTodoDateCreated(dateCreated)}
        />
      </div>
      <div className="form-group">
        <label htmlFor="todoCompleted">Completed: </label>
        <input
          type="checkbox"
          name="todoCompleted"
          checked={todoCompleted}
          onChange={event => setTodoCompleted(event.target.checked)}
        />
      </div>
      <div className="form-group">
        <input
          type="submit"
          value="Submit"
          className={`btn ${!todoDescription ? `btn-danger` : `btn-
primary`}`}
          disabled={!todoDescription}
        />
      </div>
    </form>
  );
};

TodoForm.propTypes = {
    submitTodo: PropTypes.func.isRequired
}

export default TodoForm;
```

AddEditTodo.jsx

```
import React from 'react';
import PropTypes from 'prop-types';
import './css/AddEditTodo.css';
import generateTodoId from './utils/generateId';
import TodoForm from './TodoForm';
import TodoModel from './utils/Todo.model';

const AddEditTodo = props => {
  const submitTodo = (todoDescription, todoDateCreated, todoCompleted) =>
{
    const _id = generateTodoId();
    const newTodo = new TodoModel(
      todoDescription,
      todoDateCreated?.toISOString(),
      todoCompleted,
      _id
    );
    props.submitTodo(newTodo);
  }
  return (
    <>
      <div className="addEditTodo row">
        <h3>Add/Edit Todo</h3>
      </div>
      <TodoForm submitTodo={submitTodo} />
    </>
  );
}
AddEditTodo.propTypes = {
  submitTodo: PropTypes.func.isRequired
}
export default AddEditTodo;
```

App.js

```
import React, { useState } from 'react';
import 'bootstrap/dist/css/bootstrap.min.css';
import 'bootstrap';
import 'popper.js';
import 'jquery';
import './Components/css/qa.css';
import sampleTodos from './sampleTodos.json';

import Header from './Components/Header';
import Footer from './Components/Footer';
import AllTodos from './Components/AllTodos';
import AddEditTodo from './Components/AddEditTodo';

function App() {
  const [todos, setTodos] = useState(sampleTodos);

  const submitTodo = todo => {
    const updatedTodos = [...todos, todo];
    setTodos(updatedTodos);
  }
  return (
    <div className="container">
      <Header />
      <div className="container">
        <AllTodos todos={todos} />
        <AddEditTodo submitTodo={submitTodo} />
      </div>
      <Footer />
    </div>
  );
}
```

```
export default App;
```

This is the end of Quick Lab 14a

QuickLab 14b – Thinking In React Part 5 – Testing Form Submission

Objectives

- To be able to add inverse data flow to an application by passing handler functions by props

Overview

In this QuickLab, you will test the submission of the form triggers the submit function that is passed in through props. The function will be mocked and spied on to ensure that it is called. Tests should also check that the state values are reset on submission. This will be achieved by creating an instance of the TodoForm component with an appropriate prop for submitTodo – this will need to be added to all instance of the TodoForm component that are created in this test file to maintain the correct construction for each test. Asynchronous calls will be made to the onChange methods to update values and then the onSubmit event should be fired. The assertions from this test is that the submitTodo function has been called once with the values for description and completed as set in the asynchronous calls and null for the date created. Further assertions should be made to check that the values of description and completed are reset to an empty string and false, respectively.

You should continue working on the same project as you used for QuickLab 14a.

Activity 1 – Adding the submitTodo prop to TodoForm

1. Open TodoForm.test.js for editing.
1. Add a suite variable called `submitTodo`, leaving it `undefined`, to the current TodoForm test suite called `Form submission tests`.
2. Add a `beforeEach` to the suite that sets `submitTodo` to `jest.fn()`.
3. For each `create` function that is currently in the suite, add a `prop` of `submitTodo` set to `submitTodo` to every `TodoForm` component made within the file.
4. Run the tests to verify that that all still pass.

Activity 2 – Testing Form Submission

5. Add a *new test suite* to the `TodoForm test suite` called `Form submission tests`.
6. Add a `test` with a title of `` `it should call submitTodo with a prescribed parameters on submission` `` and an *<u>async</u> arrow function* that:
    - Creates a `testRenderer` using a `TodoForm` component with a

`prop` of `submitTodo` set to `submitTodo`;

- Creates a `testInstance` set to the `root` of the `testRenderer`;
- Obtains the **input** element with a `prop` name of `todoDescription` and sets a `const` `descInput`;
- Obtains the **input** element with a `prop` name of `todoCompleted` and sets a `const` `completedInput`;
- Defines a `const` `descTestValue` set to the **string** `Test`;
- Defines a `const` `compTestValue` set to the **Boolean** `false`;
- Obtains the form element and sets a `const` `form`;
- **Await** a call to `act` which has a parameter of an *async arrow function* that returns a call to `onChange` on the `props` of `completedInput`, passing in an **object** that has a **key** of `target` and a **value** that is an **object** with a **key/value** pair of `checked` and `compTestValue`;
- **Await** a call to `act` which has a parameter of an *async arrow function* that returns a call to `onChange` on the `props` of `descInput`, passing in an **object** that has a **key** of `target` and a **value** that is an **object** with a **key/value** pair of `value` and `descTestValue`;
- **Await** a call to `act` which has a parameter of an *async arrow function* that returns a call to `onSubmit` on the `props` of `form`, passing in **new** `Event` object with a parameter of the string `` `form` ``;
- **Assert** that the `submitTodo` function *has been called once*;
- **Assert** that the `submitTodo` function *has been called with* the values `descTestValue`, `null` and `compTestValue`.

7. Add a `test` that checks that the `completedInput` and `descInput` return to their **default values**, set up the same as this one (refactor if you wish) and **asserting** `descInput.props.value` is an *empty string* and that `completedInput.props.checked` is `false`.

8. Save the file and run the tests.

All tests should pass.

If you have time...

Follow the same mocking call pattern to test the submitTodo in the AddEditTodo component and in the App component.

Code Snippets

TodoForm.test.js

```
import React from 'react';
import { create, act } from 'react-test-renderer';

import TodoForm from '../Components/TodoForm';

jest.mock("../Components/utils/DateCreated", () => {
  return function MockDateCreated() {
    return <span testid="dateCreated">Date Created Component</span>
  }
});

describe(`TodoForm test suite`, () => {
  let submitTodo;                                                    // 1.2

  beforeEach(() => {                                                 // 1.3
    submitTodo = jest.fn();                                          // 1.3
  });                                                                // 1.3

  describe(`DateCreated function and render tests`, () => {
    test(`it should render a DateCreated component a date`, () => {
    const testRenderer = create(<TodoForm
                                 submitTodo={submitTodo} />);    // 1.4

    const testInstance = testRenderer.root;

    const dateCreated = testInstance.find(
      el => el.type === `span` && el.props.testid === `dateCreated`);

    expect(dateCreated).toBeTruthy();
     expect(dateCreated.children).toContain(`Date Created Component`);
  });
});

  describe(`onChange event tests`, () => {
    test(`it should render the new value in the input when the
      todoDescription onChange function is activated`, () => {

      const testValue = `Test`;
      const testRenderer = create(<TodoForm submitTodo={submitTodo} />);
      const testInstance = testRenderer.root;
      const descInput = testInstance.findByProps(
        { name: "todoDescription" }
      );

      expect(descInput.props.value).toBe(``);

      act(() => descInput.props.onChange(
            { target: { value: testValue } }
          )
      );

      expect(descInput.props.value).toBe(testValue);

  });

    test(`it should render the new value in the checkbox when the
          todoCompleted onChange function is activated`, () => {

      const testValue = true;
      const testRenderer = create(<TodoForm submitTodo={submitTodo} />);
      const testInstance = testRenderer.root;
      const completedInput = testInstance.findByProps(
        { name: "todoCompleted" }
      );

      expect(completedInput.props.checked).toEqual(false);

      // continued over
```

```
      act(() => completedInput.props.onChange(
        { target: { checked: testValue } })
      );

      expect(completedInput.props.checked).toBe(testValue);

    });

  });

  describe(`Form submission tests`, () => {
    test(`it should call submitTodo and reset the form on submission`,
          async () => {

      const testRenderer = create(<TodoForm submitTodo={submitTodo} />);
      const testInstance = testRenderer.root;
      const descInput = testInstance.findByProps(
        { name: "todoDescription" });

      const descTestValue = `Test`;

      const compTestValue = true;

      const completedInput = testInstance.findByProps(
        { name: "todoCompleted" });

      const form = testInstance.findByType('form');


      await act(async () => completedInput.props.onChange(
                              { target: { checked: compTestValue } })
      );

      await act(async () => descInput.props.onChange(
                              { target: { value: descTestValue } })
      );

      await act(async () => form.props.onSubmit(new Event(`form`)));

      expect(submitTodo).toHaveBeenCalledTimes(1);
      expect(submitTodo).
        toHaveBeenCalledWith(descTestValue, null, compTestValue);

      expect(descInput.props.value).toBe(``);

      expect(completedInput.props.checked).toBe(false);
    });

  });

});
```

This is the end of Quick Lab 14b

QuickLab 15 – External Data – Using useEffect

Objectives

- To be able to use the useEffect hook

Overview

In this QuickLab, you will transfer the initial setting of state from the useState call to a call on useEffect. Although functionally, this will not add anything to the application, it is a starting point to transferring the obtaining of data from a static file import to a call to an external service. The useEffect hook needs to be imported and then implemented, simply calling setTodos with the imported sampleTodo array.

You should use the d-external-data/starter folder for this QuickLab, remembering to run an npm install before starting.l.

Activity

1. Open App.js for editing.
2. Add `useEffect` to the list of imports from React (it should be placed inside the {}).
3. Change the value of the `useState` call in the Component to an *empty object*.
4. Add a call to `useEffect` – it takes an *arrow function* that executes a call to `setTodos` using the `sampleTodos` as the value for a key of `todos` in *object*.
5. In the render of AllTodos, remove *one of the sets of curly braces* around `todos` in the `data` property.

Save the file and run the application.

You will now get an error as initially, AllTodos receives an empty object that does not have a todos property.

6. Temporarily fix this by making todos and optional chain in the map function int AllTodos.jsx (line 9): `const todos = data.todos?.map(…`

It will load the todos when available but not allow adding.

Code Snippets - App.js

```
import React, { useState, useEffect } from 'react';   // 2
import 'bootstrap/dist/css/bootstrap.min.css';
…

function App() {
  const [todos, setTodos] = useState({});             // 3
  useEffect(() => {                                                // 4
    setTodos({todos: sampleTodos});                   //
  });
```

...

This is the end of Quick Lab 15b

QuickLab 16 - Installing JSON server

Objectives

- To be able to create a mock RESTful data service using a JSON file

Activity

1. Open an additional terminal/command line and enter:

```
npm i -g json-server
```

This installs JSON server globally on your computer - you don't need to install it each time you want to use it in a project.

The data file for this QuickLab has been provided as **todoData.json** in the folder **d-external-data**.

2. Still on the command prompt, start JSON-Server, pointing at this file by navigating to the folder described above and entering the command:

```
json-server --watch todoData.json -p 4000 --id _id
```

The --id _id sets the unique identifier used by JSON-Server to the _id key provided in the JSON file.

The command line running json-server needs to be kept open and running for the duration of your development work. If it needs to be restarted it can be by following instruction 2 above, provided that the command line is pointing at the folder containing the json file when the command is executed.

3. Check that the server is running by navigating to the URL below and checking that adding **/5cc08495bf3fd62d03f2f4c2** to the end of the URL shows the data for the todo with todoDescription of Sample Todo 2:

```
http://localhost:4000/todos
```

This is the end of QuickLab 16

QuickLab 17 - Use an External Service – Getting Data

Objectives

- To be able to fetch data from an external service

Overview

In this QuickLab, you will replace the static array with a call to an external REST API. To do this you will need to use the useEffect hook within App to make a GET call (using axios) to the retrieve the todos from your json-server and then set the todos in the app to this. Try/catch blocks should be used to conditionally render messages if the data is loading or if it fails to be fetched.

As the AllTodos component is dependent on the data, it should display a message to the user in place of the todo data should the data take longer to load than anticipated or the retrieval from the data source be unsuccessful.

You should use the d-external-data/starter folder or extend your previous Todo application.

Activity 1 - Use axios to obtain data from JSON server

1. Ensure that JSON server is running a serving the todoData.json file:

```
json-server --watch todoData.json -p 4000 --id _id
```

2. On the command line, navigate to your project folder and run the following command to install axios for the project:

```
npm i --save axios
```

3. Open App.js for editing.
4. Add an import for `axios` from `axios`:
5. Comment out the import for `sampleTodos`.
6. Under the list of imports, declare a `const` called `TODOSURL` and set it to be `` `http://localhost:4000/todos` ``.
7. Add a new state called `getError`, initially set to an *empty string*.
8. After the useEffect, declare a `const` called `getTodos` and set this to be an `async` arrow function that:
    - *Tries* to:
        - Set a `const` called `res` to `await` the return of an `axios.get` call to `TODOSURL`;
        - Returns an object that has a key of `todos` and a value of `res.data` if res.data has length

        - Return a key of `error` with a value of `` `There are no todos stored` `` *otherwise*
    - Catches an error called `e` and:
        - Calls `setGetError` with `` `Data not available from server: ${e.message}` ``.
        - Returns an object of a key of `error` and a value of `` `Data not available from server: ${e.message}` ``.
9. In the useEffect function:
    - Change the call to `setTodos` so it awaits a call to `getTodos`.
    - Surround the `setTodos` call in an *async arrow function* called `getData`.
    - Call `getData`.
10. Surround the return of the App component in a React Fragment and then *conditionally render* a Modal component (imported from ./Components/utils) dependent on `getError` having a value.
11. Pass the Modal component props of:
    - `handleClose` – set to a callback that sets `getError` to an *empty string*;
    - `message` – set to the value of `getError`.

This component will display a modal box if there is an error retrieving the data.

12. Save the file.

Activity 2 - Add 'no-data' error handling to the AllTodos component

1. Open Components/AllTodos.jsx for editing.
2. Remove or comment out the existing logic in the component.
3. Set a state of `dataStatus` with an initial value of an object that has keys of `name` and `message` and values of `` `loading` `` and `` `Data is loading…` ``.
4. Add a `useEffect` call that has:
    - An `if` statement that checks to see if the `data` prop is available and has an `error` property, setting the `dataStatus` state to `` `error` `` for `name` and `data.error` for `message`;
    - An `else if` that checks to see if the `data` prop is available and has a `todos` property then:
        - Declares a `const` called `ds` and sets this dependent on `data.todos` having a length greater than 0:
            - to an object with a key of `name`, value `` `data` `` and key `message` set to `null` if it does
            - to an object with a key of `name`, value `` `nodata` ``

and key `message` set to `` `There were no todos previously saved` `` if it doesn't ;

- A final `else` clause that resets the `dataStatus` state to its initial value;
- A dependency array that contains `data`.

5. Create a **function** called `populateTable` that:
    - Conditionally checks to see if `data?.todos?` has `length` *greater than 0*, **return** a call to `map` on `data.todos`, passing in the `currentTodo` and:
        - *Destructuring* the `currentTodo` to its 4 properties;
        - Instantiating an *instance* of the `TodoModel` class using the `currentTodo's` properties as `todo`;
        - Returning a `Todo` component that passes in a **prop** of `todo` with a value of `todo` and a `key` of `todo._id`.
    - Returns a *table row* with a *single cell* that:
        - Uses the `dataStatus.name` property to set its `id`
        - Spans 3 columns
        - Has content of `dataStatus.message`.
6. In the `return` of the component, make the content of the *table body* call the function `populateTable`.
7. Ammend the `PropTypes` for this component so data can now be `oneOfType` – which is an **array** that contains the original **exact** object shape for a **todo**, or an **object** with a **key** of `error` and a **value** that is a **string** and finally, simply an ***empty object***.
8. Save the file and run the application.

It should still function as it did before we used the 'External Data' source.

9. Locate the command line that is running JSON Server and stop it (using **CTRL+C**).
10. Refresh the application and you should see the **Modal** and then the ***error message*** in place of the **todos**.
11. Add a ***setTimeout*** that lasts ***5 seconds*** around the ***getData()*** call in ***useEffect*** in **App.js** - you should see the *"loading"* message until the data is returned. You may remove this once you have checked – this should be tested in any tests that you write.
12. Remove the ***todo objects*** from the **todoData.json** file and check that the correct message is displayed (you may need to restart json-server to effect this change).

If You Have Time...

Read up on writing tests using @testing-library/react, @testing-library/jest-dom and its *extended matchers* to test whether the AllTodos component still renders correctly.

The code for this can be found in the ql17-solution/src/tests/AllTodos.test.js file and requires async tests that use screen, query helpers and extended matchers.

Code Snippets

App.js

```
import React, { useState, useEffect } from 'react';
import 'bootstrap/dist/css/bootstrap.min.css';
import 'bootstrap';
import 'popper.js';
import 'jquery';
import './Components/css/qa.css';
// import sampleTodos from './sampleTodos.json';                    // 1.5
import axios from 'axios';                                          // 1.4
import Modal from './Components/utils/Modal';
import Header from './Components/Header';
import Footer from './Components/Footer';
import AllTodos from './Components/AllTodos';
import AddEditTodo from './Components/AddEditTodo';

const TODOSURL = `http://localhost:4000/todos`;                     // 1.6

function App() {
  const [todos, setTodos] = useState({});
  const [getError, setGetError] = useState(``);                     // 1.7

  useEffect(() => {                                                 // 1.9
    const getData = async () => {                                   //
      setTodos(await getTodos());                                   //
    }
    setTimeout(() => {                                              // 2.11
      getData();                                                    //
    }, 5000)                                                        //
  }, []);

  const getTodos = async () => {                                    // 1.8
    try {                                                           //
      const res = await axios.get(TODOSURL);                        //
      return res.data.length ? { todos: res.data } :                //
        { error: `There are no todos stored`} ;                     //
    } catch (e) {                                                   //
      setGetError(`Data not available from server: ${e.message}`);
      return { error: `Data not available from server: ${e.message}` };
    }                                                               //
  };                                                                //

  const submitTodo = todo => {
    const updatedTodos = [...todos.todos, todo];                    //
    setTodos({ todos: updatedTodos });                              //
  }

  return (
    <>
      {getError && <Modal handleClose={()=>setGetError(``)}         //1.10
        message={getError} />}
      <div className="container">
        <Header />
        <div className="container">
          <AllTodos data={todos} />
          <AddEditTodo submitTodo={submitTodo} />
        </div>
        <Footer />
      </div>
    </>
  );
}

export default App;
```

AllTodos.js

```
import React, { useEffect, useState } from 'react';
import PropTypes from 'prop-types';
import './css/AllTodos.css';
import Todo from './Todo';
import TodoModel from './utils/Todo.model';

const AllTodos = ({data}) => {
  const [dataStatus, setDataStatus] = useState({ name: `loading`, message:
`Data is loading...` });

  useEffect(() => {

    if (data?.error) {
      setDataStatus({ name: `error`, message: data.error });
    }

    else if (data?.todos) {
      const ds = data.todos.length > 0 ?
        { name:`data`, message: null }
      :
        { name:`nodata`, message:`There were no todos previously saved` };
      setDataStatus(ds);
    }

    else {
      setDataStatus({
        name: `loading`,
        message: `Data is loading...`
      });
    }
  }, [data]);

  const populateTable = () => {

    if (data?.todos?.length > 0) {
      return data.todos.map(currentTodo => {
        const {
          todoDescription,
          todoDateCreated,
          todoCompleted,
          _id } = currentTodo;
        const todo = new TodoModel(
          todoDescription,
          todoDateCreated?.toISOString(),
          todoCompleted,
          _id
        );
        return <Todo todo={todo} key={todo._id} />
      });
    }

    return (
        <tr>
          <td id={dataStatus.name} colSpan="3">{dataStatus.message}</td>
        </tr>
      );
  }

  return (
    <div className="row">
      <h3>Todos List</h3>
      <table className="table table-striped">
        <thead>
          <tr>
            <th>Description</th>
            <th>Date Created</th>
            <th>Action</th>
          </tr>
```

```
        </thead>
        <tbody>{populateTable()}</tbody>
      </table>
    </div>

    );};

AllTodos.propTypes = {
    data: PropTypes.oneOfType([
        PropTypes.exact({
            todos: PropTypes.arrayOf(
                PropTypes.exact({

                    _id: PropTypes.string,
                    todoDescription: PropTypes.string,
                    todoDateCreated: PropTypes.string,
                    todoCompleted: PropTypes.bool
                })
            )
        }),
        PropTypes.exact({
            error: PropTypes.string
        }),
        PropTypes.exact({})
    ])
};

export default AllTodos;
```

This is the end of QuickLab 17

QuickLab 18 - Use an External Service – Sending Data

Objectives

- To be able to send data to an external service

Overview

In this QuickLab, you will add further functionality to the App component so that it is able to send the new todo submitted on the form to the REST API as POST request. To do this you will modify the submitTodo function in App.js so that rather than setting a new state, it sends the data to the REST service and then calls getTodos to retrieve them again. You should also manage the online status of the application during this process and any errors that may be returned.

You should use the **d-external-data/starter** folder that you used for QuickLab 17, or use the ql17-solution folder as a base for this QuickLab.

Activity 1 - Write a method to POST a new todo

1. Ensure that JSON server is running a serving the **todoData.json** file:

```
json-server --watch todoData.json -p 4000 --id _id
```

2. Open **App.js** for editing.
3. Add a **state** called `postError` with an *initial value* of an *empty string*.
4. Change submitTodo so that it is an **async** *arrow function* that takes `todo` as an argument and replace its body with:
   - A call setting `postError` to an *empty string;*
   - A **try** block that and and `await` an `axios` `post` call providing `TODOSURL` and `todo` as arguments
   - A **catch** block that takes `e` as an argument and sets `postError` to a **string** that explains the error;
   - A **finally** block that calls `setTodos`, `await`ing `getTodos`.for the value.
5. After the condition render for the `getError` modal, add another *conditionally render* a `Modal` component dependent on `postError`. The `Modal` should receive **props** of:
   - `handleClose` which is a **function** call to `setPostError` with an *empty string*;
   - `message`, which receives a value of `postError`.
6. Save the file.

The application should function correctly. Check that you can persist a new todo and also that the error modal is displayed when you turn json-server off.

If you have time...
Write functionality for the application that would allow the editing of a particular todo. See where you may be able to make components more type-safe using PropTypes or have cleaner code through *refactoring* and *destructuring*. Add any additional tests that you feel are required.

The solution for this can be found in the folder ql18-solution-extended. There is a *markdown* file in the project that explains the steps taken.

Code Snippets - App.js

```
import React, { useState, useEffect } from 'react';
import 'bootstrap/dist/css/bootstrap.min.css';
import 'bootstrap';
import 'popper.js';
import 'jquery';
import './Components/css/qa.css';
import axios from 'axios';
import Header from './Components/Header';
import Footer from './Components/Footer';
import AllTodos from './Components/AllTodos';
import AddEditTodo from './Components/AddEditTodo';

const TODOSURL = `http://localhost:4000/todos`;

function App() {
  const [todos, setTodos] = useState([]);
  const [getError, setGetError] = useState(``);
  const [postError, setPostError] = useState(``);
  useEffect(() => {
    const getData = async () => {
      setTodos(await getTodos());
    }
    getData();
  }, []);

  const getTodos = async () => {
    try {
      const res = await axios.get(TODOSURL);
      return res.data.length ? { todos: res.data } :
        { error: `There are no todos stored`} ;
    } catch (e) {
      setGetError(`Data not available from server: ${e.message}`);
      return { error: `Data not available from server: ${e.message}` };
    }
  };

  const submitTodo = async todo => {
    setPostError(``);
    try {
      await axios.post(TODOSURL, todo);
    } catch (e) {
      setPostError(`There was a problem adding the todo: ${e.message}`)
    } finally {
      setTodos(await getTodos());
    }
  }
  return (
    {getError && <Modal handleClose={()=>setGetError(``)}         //1.10
        message={getError} />}
    {postError &&
      <Modal
        handleClose={()=>setPostError(false)} show={postError}
       />
    }
    <div className="container">
      <Header />
```

```
      <div className="container">
        <AllTodos data={todos} />
        <AddEditTodo submitTodo={submitTodo} />
      </div>
      <Footer />
    </div>
  );
}
export default App;
```

This is the end of QuickLab 18

QuickLab 19 – Routing the Application

Objectives

- To be able to use simple routes in an application
- To be able to add internal links to an application

Overview

In this QuickLab, you will add routes to App.js for the default view, which will be the list of AllTodos and an AddTodo view, which will display the AddEditTodo component. To do this, an extra package called react-router-dom will need to be added to the project. A BrowserRouter component will wrap the whole application to enable routing. Routes will be defined in a Switch component to ensure that matches are made.

Once the application routing has been defined. The Header component will be modified to use a Link component rather than the usual <a href> combination to use React's routing system.

You should use the e-routed-app/starter folder as the 'Edit Todo' functionality has been included here.

Activity 1 – Setting up for routing

1. Ensure that JSON server is running a serving the todoData.json file.

```
json-server --watch todoData.json -p 4000 --id _id
```

2. Point a command line at the root of your project and install react-router-dom using the following command:

```
npm i --save react-router-dom
```

3. Open App.js for editing.

1. Under the import for React, import `BrowserRouter as Router`, `Route` and `Switch` from react-router-dom.

2. In the return of App, wrap all of the markup in a `<Router>` component:

Activity 2 - Define the application's Route components

1. Surround the call for the AllTodos and AddTodo components with a `<Switch>`.

2. Surround the call for `<AllTodos…>` with a `<Route>` that has properties:

    - An `exact` `path` set to `'/'`;

3. Surround the call for `<AddEditTodo…>` with a `<Route>` with:
    - a `path` of `'/add'`;
4. Save the file and check that the navigation works as expected when the path is typed into the address bar.

Notice how the application re-renders completely when an address is typed and that the Edit button on AllTodos does not appear to have any affect now .

You will also notice some tests now fail. These are fixed in the solution and changes are documented in /test/TestChanges.md. Tests for Editing a todo are skipped as this functionality is temporarily in an undesired state.

Activity 3 – Add navigation to the Header

1. Open Header.jsx for editing from the Components folder.
2. Import `Link` and `NavLink` from react-router-dom (inside {})
3. Locate the link surrounding the text *Todo App* and:
    - Replace the `<a></a>` with `<Link></Link>`;
    - Replace `href`= with `to=`.
4. Add links for Todos and Add Todo that sit inside a div with classes of `collapse navbar-collapse` and a ul that has classes of `navbar-nav mr-auto`. Each list item should:
    - Have a `className` of `navbar-item`;
    - Have a NavLink that has a `to` attribute of `"/"` or `"/add"`, a `className` of `nav-link` and an `activeClassNames` of `nav-link active`
5. Save the file.

Check that the application now renders without completely refreshing the application - check the network activity in the Developer Tools to make sure!

Start the json-server again.

Try clicking on Edit in the table and then on the Add Todo link. You will notice that the todo is displayed in the form and you can submit it. This will cause the form to rerender and display a blank form. If you navigate to home, then the edited todo has been saved.

Code snippets are on the next page.

Code Snippets

App.js

```
import React, { useState, useEffect } from 'react';
import {                                                          // 1.4
  BrowserRouter as Router,                                        //
  Route,                                                          //
  Switch                                                          //
} from 'react-router-dom';                                        //
… /* Other code omitted from snippet */

function App() {

  /* This function's logic has been omitted from snippet */

  return (
    <Router>                                                      // 1.5
      {/* conditional renders omitted from snippet */}
      <div className="container">
        <Header />
          <Switch>                                                // 2.1
            <Route exact path="/" >                               // 2.2
              <AllTodos data={todos} selectTodo={selectTodo} />
            </Route>
            <Route path="/add">                                   // 2.3
              <AddEditTodo
                  submitTodo={submitTodo}
                  updateTodo={updateTodo}
                  todo={todoToEdit} />
            </Route>
          </Switch>                                               // 2.1
        </div>
        <Footer />
      </div>
    </Router>                                                     // 1.5
}

export default App;
```

Header.js

```
import React from "react";
import { Link, NavLink } from "react-router-dom";                // 3.1
import logo from "./images/qalogo.svg";

const Header = () => {
  return (
    <header>
      <nav className="navbar navbar-expand-sm">
        <a
          href="https://www.qa.com"
          className="navbar-brand"
          target="_blank"
          rel="noopener noreferrer"
        >
          <img src={logo} alt="QA Ltd" style={{ width: '100px' }} />
        </a>
        <Link className="navbar-brand" to="/">                   // 3.3
          <h1>Todo App</h1>                                      //
        </Link>                                                  //
        <div className="collapse navbar-collapse">               // 3.4
          <ul className="navbar-nav mr-auto">                    //
            <li className="navbar-item">                         //
              <NavLink                                           //
                to="/"                                           //
                className="nav-link"                             //
                activeClassName="nav-link active"                //
              >                                                  //
                All todos                                        //
              </NavLink>                                         //
            </li>                                                //
            <li className="navbar-item">                         //
              <NavLink                                           //
                to="/add"                                        //
                className="nav-link"                             //
                activeClassName="nav-link active"                //
              >                                                  //
                Add Todo                                         //
              </NavLink>                                         //
            </li>                                                //
          </ul>                                                  //
        </div>                                                   //
      </nav>
    </header>
  );
};

export default Header;
```

This is the end of QuickLab 19

QuickLab 20 - Use Parameterised Routes

Objectives

- To be able to use parameters to define and evaluate routes
- To be able to use parameter data in view logic

Overview

In this QuickLab, you will modify the Route to be used by an edit link to pass in the Id of the todo that is to be edited, setting :_id as a Route parameter in the path. The current link in the Todo component will be updated to use a Link component along with the Id to dynamically create a link for each todo.

The AddEditTodo component will be modified so that it recognises the todo that has been chosen for editing. This will be done by leveraging the useParams hook that is supplied as part of the react-router-dom package to odenitfy the id of the todo to update.

The id will help find the todo from the array of todo objects and then parse the values into the form. The update function will operate in the same way as before.

We will add a Redirect to show the list of todos when submit is clicked.

You should continue use the e-routed-app/starter folder or the ql-19-solution.

Activity 1 – Add a Route for editing a Todo and a 404 route

1. Ensure that JSON server is running as in previous QuickLabs.
2. Open App.js for editing.
    - In the `<Route>` for `/add`, change the `submitTodo` prop's name to `submitAction` and remove all other props.
    - Under the `<Route>` for `/add` add another `<Route>` with a `path` set to `'/edit/:_id'`;
    - Populate the Route component with an `<AddEditTodo />` with attributes:
        - `submitAction` set to `{updateTodo}`;
        - `data` set to `{todos}`.
3. Add a `Route` component with *no path data* that renders the `NotFound` component included in the folder Components/utils.
4. Remove or comment any code that references `todoToEdit` or `setTodoToUpdate` and `selectTodo` as this is not needed anymore. You will find code in:
    - The declaration of state for `todoToEdit`;

- The `selectTodo` function;
- The setTodoToEdit call in the updateTodo function;
- The `selectTodo` prop passed into the `AllTodos` component.

5. Save the file.

Activity 2 – Link 'Edit' in the table to the Route

1. Open Components/Todo.jsx for editing.
2. Import `Link` from `'react-router-dom'`.
3. Change the `<span></span>` that surrounds 'Edit' for a `<Link></Link>` that has attributes:
   - `to` set to `{`/edit/${todo._id}`}`;
   - `className` set to `link`;
   - As you have removed the `selectTodo` callback from the link, if you have *deconstructed* the **props**, you can remove `selectTodo` from there and the **propTypes**.
4. Save the file.

Return to the browser and check that clicking on one of the Edit links produces a URL that has /edit/ followed by the _id contained in the JSON file for that todo.

Activity 3 – Make the AddEditTodo component recognise the Todo to edit

3.1 – Change the useEffect to use the parameter in the route and make the todo to edit be recognised in the component:

1. Open Components/AddEditTodo.jsx for editing.
2. Change the `submitTodo` prop to `submitAction` – replace props with a deconstructed object of `submitAction` and `data`.
3. Add an **import** of `useParams` from **react-router-dom**.
4. Declare a **const** `{ _id }` to be a call to the `useParams` hook.
5. Add a **useEffect** that:
   - Sets `todo` to `null` *if there is no* `_id` from the **useParams** hook.
   - Checks *if there is an* `_id` *and* that *there is not an* `error` property on `todo` and then:
     - Sets a `todoToEdit` by *finding* a `todo` in the `todos` array whose `_id` **matches** the `_id` from the **useParams** hook;
       - *If one is found*, **set** `todo` with it;
       - *If not*, **set** `todo` to an **object** with a **key** of `error` and

a value of a *string* of `Todo could not be found`;

- Has a *dependency array* that contains `_id`, `data` and `todo`.

6. In the **render**, add a **conditionally rendered** `<Modal>`, displayed if `todo` is set and has an `error` property. The `handleClose` property should set `todo` to `null` and the `message` property should be `todo.error`.
7. In the `<h3>` make the *condition* `_id`.

3.2 – Make the submitTodo function handle both adding and updating

1. Add a *new state* to the component called `submitted`, initially `false`.
2. Comment out the `updateTodo` function.
3. Add an additional argument of `todoId` to the `submitTodo` function.
4. Remove the setting of `_id` in `submitTodo`.
5. Declare a `const` called `id` and set this to either the `todoId` or a call to `generateTodoId` dependent on whether `todoId` was present.
6. Change the name of `newTodo` to `todoToSubmit` and change:
   - The `todoDateCreated?.toISOString()` call to create a `new Date` from `todoDateCreated` and *then convert it to an ISO String*;
   - Ihe final argument to `id`.
7. Change the passed value in `submitAction` to `todoToSubmit`
8. Set `submitted` to `true`;

3.3 – Make the submitting the form return to the homepage when submitted or no todo to edit exists:

1. Add an **import** of `{ Redirect }` from **react-router-dom**.
2. In the **AddEditTodo** function, add a state of `submitted` with an initial value of `false`.
3. Add a `return` to the `useEffect` that is an *arrow function* that takes *no parameters* and calls `setTodo` with an *empty object* and `setSubmitted` to `false` – this will ensure correct navigation is allowed when the component is unmounted.
4. In the `submitTodo` function, *just before the function closes* and after all other blocks inside it are closed make a call to **set** the `submitted` state to `true`.
5. In the Component's *return* insert a *conditionally rendered* `<Redirect />` with a `to` attribute set to `'/'` that is dependent on `submitted` OR `props.todos` not having a `length`, returning the *existing markup* when `submitted` is `false`.

Change the action to recognise the parameter in the route:

1. Find the declaration of `action` and replace the *condition in the ternary statement* with `todo && props.match`.
2. Save the file.

Check the navigation in the browser, you should see that the 'Edit' link now populates the form with the selected Todo and when submitted, the updated Todo is shown in the AllTodos table.

You should also find that clicking Submit after using the 'Add Todo' main navigation link returns to the list of Todos showing the newly added Todo in it.

Code Snippets

App.js

```
import React, { useState, useEffect } from 'react';
import { BrowserRouter as Router, Route, Switch } from 'react-router-dom';

import NotFound from './Components/utils/NotFound';

/* Other imports omitted */

function App() {

  /* Function logic omitted */

  return (
    <Router>
      {/* Conditional modal renders omitted here */}
      <div className="container">
        <Header />
        <div className="container">
          <Switch>
            <Route exact path="/">
              <AllTodos data={todos} />
            </Route>
            <Route path="/add">
              <AddEditTodo submitAction={submitTodo />
            </Route>
            <Route path="/edit/:_id">
              <AddEditTodo submitAction={updateTodo} data={todos}/>
            </Route>
            <Route>
              <NotFound />
            </Route>
          </Switch>
        </div>
        <Footer />
      </div>
    </Router>
  )

default export App;
```

Todo.jsx

```
import PropTypes from 'prop-types';
import { Link } from 'react-router-dom';
import TodoModel from './utils/Todo.model';

// selectTodo removed from props as no longer needed
const Todo = ({ todo }) => {
```

```
  const { todoDescription, todoDateCreated, todoCompleted } = todo;
  const dateCreated =
    new Date(Date.parse(todoDateCreated)).toUTCString();
    const completedClassName = todoCompleted ? `completed` : ``;
  let completed;

  if(todoCompleted) {
    completed = `N/A`;
  } else {
    completed = <Link to={`/edit/${_id}`} className="link">Edit</Link>
  }

  return (
    <tr>
      <td className={completedClassName}>{todoDescription}</td>
      <td className={completedClassName}>{dateCreated}</td>
      <td>{completed}</td>
    </tr>
  );
};

Todo.propTypes = {
    todo: PropTypes.instanceOf(TodoModel)
}

export default Todo;
```

AddEditTodo.jsx

```
import React, { useState, useEffect } from 'react';
import PropTypes from 'prop-types';
import { Redirect, useParams } from "react-router-dom";
import './css/AddEditTodo.css';
import TodoForm from './TodoForm';
import TodoModel from './utils/TodoModel';
import Modal from './utils/Modal';

const AddEditTodo = ({submitAction , data}) => {
  const [todo, setTodo] = useState({});
  const [submitted, setSubmitted] = useState(false);
  const { _id } = useParams();

  useEffect(() => {
    if(!_id) setTodo(null);
    if (todo?.error && _id) {
      const todoToEdit = data?.todos?.find(
        currentTodo => currentTodo._id === _id
      );
      if(todoToEdit) {
        setTodo(todoToEdit);
      } else {
        setTodo({ error: `Todo could not be found` });
      }
    }
  }, [data, todo, _id]);

  const submitTodo = (
    todoDescription,
    todoDateCreated,
    todoCompleted,
    todoId
  ) => {

    const id = todoId ? todoId : generateTodoId();
    const todoToSubmit = new TodoModel(
      todoDescription,
      new Date(todoDateCreated).toISOString(),
      todoCompleted,
      todoId
    );
```

```
    submitAction(todoToSubmit);
    setSubmitted(true);
  };

  return (
    <>
      {submitted && <Redirect to="/" />
      {todo?.error &&
        <Modal handleClose={() => setTodo(null) message={todo.error} />
      }
      <div className="addEditTodo row">
          <h3>{_id ? `Edit` : `Add`} Todo</h3>
            <TodoForm todo={todo} submitAction={submitTodo} />
        </div>
      }
    </>
  );
}

AddEditTodo.propTypes = {
  submitTodo: PropTypes.func.isRequired,

  todo: PropTypes.arrayOf(
    PropTypes.exact({
      _id: PropTypes.string,
      todoDescription: PropTypes.string,
      todoDateCreated: PropTypes.string,
      todoCompleted: PropTypes.bool
    })
  )
}

export default AddEditTodo;
```

This is the end of QuickLab 20

QuickLab 21 – State Management – Getting Data

Objectives

- To be able to use hooks for context to abstract application state away from application components

In this QuickLab, you will remove the management of application away from the view components. You will create a TodosProvider that will serve as a wrapper to the relevant parts of the application and use context to provide data.

A StateManagement folder will be created with a file for the TodosProvider. React's createContext will be used to create a TodosStateProvider and a useTodosState context hook. This is a *function* and it will be *exported* to allow the *todos state* to be used and this will set a context to a call to the useContext hook – passing in TodosStateContext. The context *should be returned* as long as it is not undefined, in which case an *error* should be returned.

The component created will use a State Hook to get the data from our external service. It will receive its children as props – this is because the Provider does not know what its children will be. This is good for Provider reuse!

The return of the component will render the TodosStateContext.Provider setting a value based on the *state*.

The AllTodos component will call the useTodosState hook to get the value from the Provider and use this in place of the array of todos previously obtained from its parent via props. Some changes to the logic to render based on this will be needed.

All logic will be removed from the App component and the Routes will be altered. This reflects the fact that they do not need to pass data into components as they will now use Context. The AllTodos component will be modified to use the data.

You should use the f-state-management/starter folder or extend your previous Todo application.

Activity 1 – Prepare for State Management

1. In src, create a new folder called StateManagement.
2. Within it, create a file called TodosProvider.jsx.
3. Import `React` and `{ createContext, useContext, useEffect, useState }` from `react` and `axios` from `axios`.
4. Declare an exported constant `TodosStateContext` set to a call to `createContext()`.(This is exported for use in testing).
5. Set a `const` `baseUrl` to `` `http://localhost:4000/todos` ``.

Activity 2 – The useTodosState function

1. Export a function called `useTodosState` that:
    - Declares an `const` called `context` and sets it to a call to `useContext`, passing in `TodosStateContext`.
    - If `context` is `undefined`, throw a *new Error* with the *message* `` `useTodosState must be used within a TodosProvider` ``.
    - Returns `context`.
2. Declare an async *arrow function* called `getAllTodos` that tries to:
    - Returns the `data` property of *awaiting* an `axios` `get` call to `baseUrl`;
    - Catches an `error` and returns an object with a key of `errorMessage` and a value of `` `Data not available from server: ${error.message}` ``.
      Note: you could forward the status and conditionally render depending on it.

Activity 3 – The TodosProvider component

1. Create a *functional component* called `TodosProvider` that receives a prop of an object containing `children`.
2. Add a call to `useState`, passing in *an empty object*, *deconstructed* to *array constants* `todos` and `setTodos`.
3. Add a `useEffect` call that has a callback arrow function that:
    - Declares an async arrow function called `getTodos` that:
        - Declares a `const` `payload` that `awaits` `getAllTodos()`;
        - Calls `setTodos` passing in `payload`;
    - Calls `getTodos`.

   The useEffect dependency array should be empty.

4. The component should return a `TodosStateContext.Provider` with a property of `value` set to `{todos}`.

   (GOTCHA: make sure there are 2 sets of curly braces around todos!)

5. `TodosStateContext.Provider` wraps a `children` object.
6. Make sure that the `TodosProvider` is *exported as default*.
7. Save the file.

Activity 4 – Remove state and prop passing from App.js

1. Open App.js for editing.
2. Remove *all traces of logic* from the component class.
3. Remove the *conditional renders for the modals*

4. Replace the *all of the routes* (apart from 'NotFound') with ***single route*** that has an `exact` `path` of `"/"` and a `component` `AllTodos` which has no props – this is temporarily removes the Add/Edit functionality.
5. Wrap the `Switch` component in a `TodosProvider` component, importing it from the file created earlier.
6. Remove any ***unused imports and const declarations*** (i.e. useState, useEffect, axios and TODOSURL – You can leave AddEditTodo as we'll reuse it later).
7. Save the file.

Activity 5 – Make AllTodos use the new state management

1. Open AllTodos.jsx for editing.
2. Remove **props** from **AllTodos** and their **propTypes**.
3. Add a `const` of `{ todos }` set to a call to `useTodosState` (imported from **TodosProvider – remember it is NOT a default import so needs {}**).
4. In the `useEffect` function, modify this to use the state provided:
   - Change `data?.error` to `todos?.error` in the `if` statement, also modifying the `message` value in the `setDataStatus` call to `todos.errorMessage`;
   - In the `else if` condition, change this to just `todos`
   - Change the value in the dependency array from `data` to `todos`.
5. In the `populateTable` function:
   - Remove all references to `data` (leave just `todos`.);
   - Remove the `selectTodo` prop from the `Todo` component.
6. In the `return` for the component:
   - Wrap the code into a `React.Fragment`
   - Add a conditional render if `dataStatus.name` is the string `` `error` ``:
     - Render a `Modal` whose `handleClose` callback sets `dataStatus`' `name` to `` `confirmedError` ``, using the existing `message`.
7. Save all files and ensure that **json-server** is running with the **todoData.json** file as its data source.

The application should display the todos in the table as before. Check the React DevTools to see the affect this has had on the way the application is rendered behind the scenes. You should particularly look for the use of State and Context hooks in the App and AllTodos components.

Note: The ***Add/Edit*** functionality is temporarily unavailable as we have removed their routes.

If you have time...

Review the testing files. Refer to the TestChanges markdown file in the testing folder to see what changes have been made to the tests.

Code Snippets

TodosProvider.jsx

```
import React, { createContext, useContext, useEffect, useState } from
'react';

import axios from 'axios';

const TodosStateContext = createContext();
const baseUrl = `http://localhost:4000/todos`;

export const useTodosState = () => {
    const context = useContext(TodosStateContext);

    if (context === undefined) {
        throw new Error(`useTodosState must be used within a
TodosProvider`);
    }

    return context;
}

const getAllTodos = async () => {
    try {
        return (await axios.get(baseUrl)).data;
    }
    catch (error) {
      return {
        errorMessage: `Data not available from server: ${error.message}`
      };
    }
}

const TodosProvider = ({ children }) => {
    const [todos, setTodos] = useState({});

    useEffect(() => {
        const getTodos = async () => {
            const payload = await getAllTodos();
            setTodos(payload);
        };

        getTodos();

    }, []);

    return (
        <TodosStateContext.Provider value={{ todos }}>
            {children}
        </TodosStateContext.Provider>
    );
};

export default TodosProvider;
```

App.jsx

```
import React from 'react';
import { BrowserRouter as Router, Route, Switch } from 'react-router-dom';
import 'bootstrap/dist/css/bootstrap.min.css';
import 'bootstrap';
import 'popper.js';
import 'jquery';
import './Components/css/qa.css';
import Header from './Components/Header';
import Footer from './Components/Footer';
import AllTodos from './Components/AllTodos';
import NotFound from './Components/utils/NotFound'
import TodosProvider from './StateManagement/TodosProvider';

function App() {

  return (
    <Router>
      <div className="container">
        <Header />
        <div className="container">
          <TodosProvider>                                          // 4.6
            <Switch>
              <Route exact path="/">                               // 4.4
                <AllTodos />
              </Route>
              <Route>
                <NotFound />
              </Route>
            </Switch>
          </TodosProvider>                                         // 4.6
        </div>
        <Footer />
      </div>
    </Router>
  );
}

export default App;
```

AllTodos.jsx – required code only

```
import React, { useState, useEffect} from 'react';
import './css/AllTodos.css';
import Todo from './Todo';
import TodoModel from './utils/Todo.model';
import { useTodosState } from '../StateManagement/TodosProvider';
import Modal from './utils/Modal';

const AllTodos = () => {

  const [dataStatus, setDataStatus] = useState({
    name: `loading`,
    message: `Data is loading…`
  });

  const { todos } = useTodosState();

  useEffect(() => {
    if (todos?.errorMessage) {
      setDataStatus({name: `error`, message: todos.errorMessage});
    } else if (todos) {
      const ds = todos.length > 0 ?
        { name: `data`, message: null }
      : {
          name: `nodata`,
          message: `There were no todos previously saved`
        };

      setDataStatus(ds);
    } else {
      setDataStatus({
        name: `loading`,
        message: `Data is loading`
      });
    }
  }, [todos]);

  const populateTableBody = () => {

    if (todos?.length > 0) {
      return todos.map(currentTodo => {
        const {
          todoDescription,
          todoDateCreated,
          todoCompleted,
          _id } = currentTodo;

        const todo = new TodoModel(
          todoDescription,
          new Date(todoDateCreated).toISOString(),
          todoCompleted,
          _id);

        return <Todo todo={todo} key={todo._id} />
      });
    }

    return (
        <tr>
          <td id={dataStatus.name} colSpan="3">
            {dataStatus.message}
          </td>
        </tr>
      );
  };

  return (
    <>
      {dataStatus.name === `error` &&
```

```
          <Modal handleClose={() => setDataStatus({
            name: `confirmedError`,
            message: dataStatus.message})}
            message={dataStatus.message} />
      }

      <div className="row">
        <h3>Todos List</h3>
        <table className="table table-striped">
          <thead>
            <tr>
              <th>Description</th>
              <th>Date Created</th>
              <th>Action</th>
            </tr>
          </thead>
          <tbody>{populateTableBody()}</tbody>
        </table>
      </div>
    </>
  );
};

export default AllTodos;
```

This is the end of QuickLab 21

QuickLab 22 – State management – Dispatching Data

Objectives

- To be able to use hooks for context and reducers to abstract application state management away from application components

Overview

In this QuickLab, you will create a todosReducer. You will define actions to get all todos, add a todo and to edit a todo.

In the first instance, you will replace the use of useState in the Provider with a useReducer hook. This will utilise the reducer to update state. However, since reducer functions should be pure, we need to wrap the reducer in an asynchronous handler so that we can perform server calls before changing the application's state. We will extend the application state to deal with errors.

To allow the passing of data from the TodoForm to update the state and send data to the server, you will define a useTodosDispatch function. This will use a TodosDispatchContext pass the function to components lower in the tree. The rendering of this context will be added to the TodosProvider component and it will pass in dispatch as a value – wrapped in our asynchronous handler.

A custom useTodosDispatch hook will be created to give access to this context in any component. It will be used in the TodoForm component, modified so that the dispatch function deals with the submitting of the form, rather than passing down handler functions from parent components. Dispatch will also help to clear error states to the components.

The submit functionality partly lives in the AddEditTodo component currently, so you will need to modify this component to remove logic for submitting. The todo state will be used in this component (to find specific todos to edit), so it will need utilise useTodosState and filter this instead. The App component will need routes to allow adding and editing of todos with the new functionality.

You should use the ql21-solution as a base for this QuickLab or extend your previous Todo application.

Part 1 – Getting data using the reducer

Activity 1 – Adding a pure reducer function

1. In the TodosProvider.jsx file add an arrow function called `todosReducer`, it should:
    - Take *arguments* of `state` and `action`;
    - `switch` on the `action.type` and for:
        - A case of `` `allTodos` `` it should:

- Check to see *if* `action.payload?` has an `errorMessage` property and **return** an *object* with a **key** of `todos` with value `null` and a **key** of `errors` that is an **object** that *collects* the previous state's `errors` in an object and adds or resets a key of `get` to the `errorMessage` in `action`'s `payload`.
- **Return** an *object* that has a **key** of `todos` set to `action.payload`. and a **key** `errors` set to `null`.

Activity 2 – Create the Dispatch Context

1. Export a `const` of `TodosDispatchContext` set to `createContext` under the declaration of the `TodosStateContext`.
2. Under the `useTodosState` declaration, **export** a new function called `useTodosDispatch`, the function body should:
    - Set a `const` called `context` set to a call to the `useContext` hook with `TodosDispatchContext` as an argument;
    - **If** `context` is `undefined`, **throw** a new `Error` with the message `` `useTodosDispatch must be used within a TodosProvider` `` otherwise **return** context.

Activity 3 – Use the useReducer hook in the TodosProvider

1. In the `TodosProvider`, change the `useState` hook for a `useReducer` hook that desconstructs `state` and `dispatch` to a call to `useReducer`. Swap the import at the top of the file.
2. Pass the `useReducer` hook arguments of `todosReducer` and an *object* with **keys** of `todos` set to `null` and `errors` set to an *empty object*.
3. In the `useEffect`, change the call to `setTodos` to a call to `dispatch` and pass it an **object** a **key** of `type` set to `` `allTodos` `` and `payload` as `payload`.
4. In the `TodosStateContext.Provider`, change the value to `state` – *remove one set of curly braces*
5. Save the file.

Activity 4 – Modify the state use in the AllTodos component

1. Add a *second argument* of `errors` to the deconstruction of the `useTodosState`.
2. In `useEffect`, change the condition of the if statement to be `errors?.get` and the setting of `message` in the `setDataStatus` call to `errors.get`.
3. Add *second dependency* of `errors` to the `useEffect` dependency array.

4. Ensure that all files are saved and run the application (be sure to make sure that json-server and the correct project are running!)

If you return to the browser, you should find that the list of todos still displays. If you turn off json-server and/or change the baseUrl, you can enforce the error conditions.

Note that the Add and Edit Todo functionality are still missing!

Part 2

Activity 5 – Use the reducer to deal with adding and editing a new todo

5.1 Add an async method to add the todo to the endpoint:

1. Open TodoProviders.jsx for editing.
2. Add an async arrow function called `addTodo` under the `getAllTodos` function. It should:
    - Take an argument of `newTodo`;
    - try to:
        - `await` a `post` call to `axios` providing the `baseUrl` and the `newTodo` as arguments
        - Return an *object* with a key of `successMessage` and a value of `` `The todo was successfully added` ``;
    - Or catch an *error object* and return an *object* with a key of `errorMessage` and a value of `` `There was an problem adding the todo: ` ``, appending the *error object*'s `message`.

5.2 Add an `addTodo` action to the todosReducer function:

1. Add a `case` for `` `addTodo` ``, it should:
    - Check to see if `action`'s `payload` has an `errorMessage` and return an object with:
        - A *collection* of the *previous state* (`...state`);
        - Key `errors` *set* to an object to *collect* the *previous* `state`'s `errors` and an additional key of `post` *set* to `action payload`'s `errorMessage`;
        - Key `success` set to `null`.
    - Otherwise return an *object* that has:
        - Key `todos` *set* to `action.payload.todos`;
        - Key `errors` *set* toan object to *collect* the *previous* `state`'s `errors` and and an additional key of `post` *set* to `null`
        - Key `success` *set* to `action payload`'s `addTodoResult`.

5.3 Add an `editTodo` action to the todosReducer function:

1. Add a `case` for `` `editTodo` ``, it should:
    - Check to see if `action`'s `payload` has an `errorMessage` and **return** an object with:
        - A *collection* of the *previous state* (`...state`);
        - Key `errors` *set* to an object to *collect* the *previous* `state`'s `errors` and an **additional key** of `put` *set* to `action payload`'s `errorMessage`;
        - Key `success` **set** to `null`.
    - Otherwise **return** an *object* that has:
        - Key `todos` *set* to `action.payload.todos`;
        - Key `errors` *set* toan object to *collect* the *previous* `state`'s `errors` and and an **additional key** of `put` *set* to `null`
        - Key `success` *set* to `action payload`'s `editTodoResult`.

5.4 Add a `clearAddEditMessages` action to the todosReducer function:

1. Add a `case` for `` `clearAddEditMessages` ``, it should return an object that is:
        - A *collection* of the *previous state* (`...state`);
        - Key `errors` *set* to an object to *collect* the *previous* `state`'s `errors`, an **additional key** of `post` *set* to `null` and an **additional key** of `put` *set* to `null`;
        - Key `success` **set** to `null`.

5.5 Add an asynchronous wrapper function for todosReducer

As reducer functions should be pure, we shouldn't make asynchronous calls in them. To combat this, we will wrap the reducer in an asynchronous handler function called todosReducerAsyncHander.

1. Create an *async arrow function* above `todosReducer` called `todosReducerAsyncHander`. It should receive `dispatch` as an argument and **return**:
    - An *async arrow function* that receives `action` as an argument and a function body that:
        - Switches on `action.type`;
        - For a `case` of `` `addTodo` `` it should:
            - Set a `const` called `addTodoResult` which is to `await` a **call** to `addTodo` with `action.payload` as an argument;
            - Set a `const` of `payload` set to an *object* to collect `addTodoResult` and an **additional key** of `todos`

set to `await` a call to `getAllTodos`;

- Call `dispatch` with a `type` key set to `action.type` and a `payload` key/value pair.

- For a `case` of `` `editTodo` `` it should:
  - Set a `const` called `editTodoResult` which is to `await` a call to `editTodo` with `action.payload` as an argument;
  - Set a `const` of `payload` set to an *object* to collect `editTodoResult` and an additional key of `todos` set to `await` a call to `getAllTodos`;
  - Call `dispatch` with a `type` key set to `action.type` and a `payload` key/value pair.
- For a `case` of `` `clearAddEditMessages` `` it should `dispatch` an *object* that sets `type` to `action.type`.
- For the `default` case, it should `dispatch` an `object` with:
  - `type` set to `` `appError` ``;
  - `payload` set to an *object* with a key of `errorMessage` set to `` `There was an application error` ``.

5.6 – Add the new dispatch context to the application

1. Add a further key/value pair of `success` set to `null` in the *object* passed to as the *second argument* to the `useReducer` hook.
2. Insert a `TodosDispatchContext.Provider` between the render of the `{children}` and the `TodosStateContext.Provider` in the return of the `TodoProvider`.
   - Add a `value` of a *call* to `todosReducerAsyncHandler`, passing in `dispatch`.

5.7 – Reinstate the `/add` and `/edit/:_id` routes in App

1. In App.js, add a route above the AllTodos route that has a `path` of `/add` that renders an AddEditTodo component.
2. Follow this with a route that has a `path` of `/edit/:_id` that renders an AddEditTodo component.

58. – Modify the AddEditTodo component to use the state context

1. Remove all props and propTypes from the component.
2. Remove the `submitted` state as this will be dealt with in the TodoForm component in the future.
3. Import `useTodosState` and in the component, setting a `const` to *deconstruct* the returned object, accessing only `todos`.

4. In `useEffect`:
    - *Change* the condition of the second `if` statement so it looks for `_id` *AND* if `todos` is present and has a `find` property;
        - Modify the `find` call to just be on `todos`;
    - *Change* the ***dependency array*** so it relies on `_id` and `todos`.
5. Remove the `submitTodo` function.
6. Remove the `Redirect` dependent on `submitted`.
7. In the render of `TodoForm`, remove the property of `submitAction`.
8. Save the file

5.9 – Modify the TodoForm so it uses the dispatch context

1. Change the *signature* of the component so it only receives a `todo` property and *match up its propTypes*.
2. Add a **state** of `submitResult` with no initial value.
3. Set a `const` `dispatch` set to `useTodosDispatch()`.
4. Set a `const` which deconstructs the return of `useTodosState` to access the `errors` and `success` properties.
5. Add a second `useEffect` function that:
    - Checks to see if there is an `errors` object and then checks to see if it has a `post` OR a `put` property:
        - If it does call `setSubmitResult` with an object that has a key of `message` set to `errors.post` *null-coalesced* with `errors.put` (this will have the action of setting it to either `errors.post` or `errors.put` dependent on which was present);
    - Checks to see if there is `success`:
        - If it does call `setSubmitResult` with an object that has a key of `message` set to `success`
    - **Returns** a *callback* that sets `submitResult` to `null`.
    - Has a ***dependency array*** of `errors` and `success`
6. Declare a *new arrow function* called `createTodo` that:
    - Sets a `const` called `todoId` based on whether a `todo` was *supplied by props*, using its **value** if it was and the `generateTodoId` from the **utils** file if it wasn't;
    - **Returns** a *new instance* of `TodoModel`, using the **state** and **id**.
7. Modify the `handleSubmit` function so that it:
    - *Prevents* the ***default action*** of the **event**;

- Sets a `const` called `type` dependent on whether `todo` was *supplied via props*, using `` `editTodo` `` if it was and `` `addTodo` `` if it wasn't;
- Sets a `const` called `payload` set to a call to `createTodo`;
- Calls `dispatch` with an *object* of key/values pairs `type` and `payload`.

8. Add a *new arrow function* called `closeModalHandler`:
    - It should take no arguments;
    - Set `submitResult` to an *object* that *collects the previous state* and adds a key of `read` to `true`;
    - Call `dispatch` with an *object* with a key of `type` set to `` `clearAddEditMessages` ``.
9. At the top of the component's return add *conditional renders*:
    - Based on `submitResult` *being present* and having a `read` property:
        - This should render a `Redirect` to `` `/` ``.
    - Based on `submitResult` and `submitResult` not having a `read` property:
        - This should render a Modal that has a `handleClose` property set to `closeModalHandler` and a `message` set to `submitResult`'s `message`.
10. Save the component.

Run the application, ensuring you have json-server running – you should find that it works fully as it did before.

Try switching json-server off at various points in the application's life cycle. Ensure that the expected modals and data display.

If you have time, examine the testing in the tests folder.

Code Snippets

TodosProvider.jsx – todosReducer function:

```
const todosReducer = (state, action) => {
    switch (action.type) {
        case `allTodos`:
            if (action.payload?.errorMessage) {
                return {
                    todos: null,
                    errors: { ...state.errors,
                              get: action.payload.errorMessage }
                };
            }
            return {
                todos: action.payload,
                errors: null
            };

        case `addTodo`:
            if (action.payload.errorMessage) {
                return {
                    ...state,
                    errors: { ...state.errors,
                              post: action.payload.errorMessage },
                    success: null
                };
            }
            return {

                todos: action.payload.todos,
                errors: { ...state.errors, post: null },
                success: action.payload.successMessage
            };

        case `editTodo`:
            if (action.payload.errorMessage) {
                return {
                    ...state,
                    errors: { ...state.errors,
                              put: action.payload.errorMessage },
                    success: null
                };
            }
            return {
                todos: action.payload.todos,
                errors: { ...state.errors, put: null },
                success: action.payload.successMessage
            };

        case `clearAddEditMessages`:
            return {
                ...state,
                errors: { ...state.errors, put: null, post: null },
                success: null
            };

        default:

            return {
                ...state,
                errors: { ...state.errors,
                          [action.type]: action.payload.errorMessage },
                success: null
            };
    }
};
```

TodosProvider.jsx – addTodoFunction function:

```
const addTodo = async newTodo => {
    try {
        await axios.post(baseUrl, newTodo);
        return { successMessage: `The todo was successfully added` };
    }
    catch (e) {
        return { errorMessage:
                    `There was a problem adding the todo: ${e.message}` };
    }
};
```

TodosProvider.jsx – editTodo function

```
const editTodo = async editedTodo => {
  try {
    await axios.put(`${baseUrl}/${editedTodo._id}`, editedTodo)
    return { successMessage: `The todo was successfully updated` };
  }
  catch (e) {
    return { errorMessage:
                `There was a problem updating the todo: ${e.message}` };
  }
}
```

TodosProvider.jsx – useTodosDispatch function:

```
export const useTodosDispatch = () => {
  const context = useContext(TodosDispatchContext);
  if (context === undefined) {
    throw new Error(
      `useTodosDispatch must be used within a TodosProvider`
    );
  }
  return context;
}
```

TodosProvider.jsx – todosReducerAsyncHandler function:

```
const todosReducerAsyncHandler = dispatch => {
    let payload = {};
    return async action => {
        switch (action.type) {
            case `addTodo`:
              const addTodoResult = await addTodo(action.payload);
              payload = { ...addTodoResult, todos: await getAllTodos() }
              dispatch({ type: action.type, payload });
              break;
            case `editTodo`:
              const editTodoResult = await editTodo(action.payload);
              payload = { ...editTodoResult, todos: await getAllTodos() }
              dispatch({ type: action.type, payload });
              break;
            case `clearAddEditMessages`:
              dispatch({ type: action.type });
              break;
            default:
              dispatch({
                type: `appError`,
                payload: { errorMessage:
                            `There was an application error` } });
        }
    }
}
```

TodoProvider Component– complete after all updates

```
const TodosProvider = ({ children }) => {

    const [state, dispatch] = useReducer(todosReducer, {});

    useEffect(() => {
        const getTodos = async () => {
            const payload = await getAllTodos();
            dispatch({ type: `allTodos`, payload });
        };

        getTodos();

    }, []);

    return (
        <TodosStateContext.Provider value={state}>
          <TodosDispatchContext.Provider value={dispatch}>
            {children}                                          //
          </TodosDispatchContext.Provider>                          //
        </TodosStateContext.Provider>
    );

}
```

AddEditTodo.jsx – needed code only

```
import React, { useState, useEffect } from 'react';
import {  useParams } from "react-router-dom";
import './css/AddEditTodo.css';
import TodoForm from './TodoForm';
import { useTodosState } from '../StateManagement/TodosProvider';

const AddEditTodo = ()=> {
  const [todo, setTodo] = useState({});
  const { id } = useParams();
  const { todos } = useTodosState();

  useEffect(() => {
    if(!_id) setTodo(null);
    if (todos?.find && _id) {
      const todoEditing = todos.find(
        currentTodo => currentTodo._id === _id
      );
      if(todoToEdit) {
        setTodo(todoToEdit);
      } else {
        setTodo({error: `Todo could not be found`});
      }
    }
  }, [todos, _id]);

  return (
    <>
      {todo?.error &&
        <Modal handleClose={() => setTodo(null)} message={todo.error} />}
      <div className="addEditTodo row">
        <h3>{_id ? `Edit` : `Add`} Todo</h3>
        <TodoForm todo={todo} />
      </div>
    </>
  );
}

export default AddEditTodo;
```

TodoForm Component – needed code only:

```
import React, { useState, useEffect } from 'react';
import { Redirect } from 'react-router-dom';
import PropTypes from 'prop-types';
import DateCreated from './utils/DateCreated';
import generateTodoId from './utils/generateId';
import { useTodosState, useTodosDispatch }
  from '../StateManagement/TodosProvider';
import TodoModel from './utils/Todo.model';
import Modal from './utils/Modal';
import generateTodoId from './utils/generateId';

const TodoForm = ({todo }) => {

  const [todoDescription, setTodoDescription] = useState(``);
  const [todoDateCreated, setTodoDateCreated] = useState(null);
  const [todoCompleted, setTodoCompleted] = useState(false);
  const [submitResult, setSubmitResult] = useState();

  const dispatch = useTodosDispatch();
  const { errors, success } = useTodosState();

  useEffect(() => {
    if (todo) {
      setTodoDescription(todoDescription);
      setTodoDateCreated(todoDateCreated);
      setTodoCompleted(todoCompleted);
    }

    return (() => {
      setTodoDescription(``);
      setTodoDateCreated(new Date());
      setTodoCompleted(false);
      setEditingTodo(false);
    })
  }, [todo]);

  useEffect(() => {
    if (errors?.post || errors?.put) {
      setSubmitResult({ message: errors.post ?? errors.put });
    }

    if (success) {
            setSubmitResult({ message: success });
    }

    return (() => setSubmitResult(null));

  }, [errors, success]);

    const createTodo = () => {

        const todoId = todo ? todo._id : generateTodoId();

        return new TodoModel(
          todoDescription,
          new Date(todoDateCreated).toISOString(),
          todoCompleted,
          todoId);
    }

  const handleSubmit = event => {
    event.preventDefault();
    const type = todo ? `editTodo` : `addTodo`;
    const payload = createTodo();
    dispatch({ type, payload});
  };

  return (
    <>
```

```
      { submitResult?.read && <Redirect to="/" />}

      { submitResult && !submitResult?.read &&
        <Modal
           handleClose={closeModalHandler}
           message={submitResult.message} />}
      <div className="container">
        <form onSubmit={handleSubmit}>
          <div className="form-group">
            <label htmlFor="todoDescription">
              Description: 
            </label>
            <input
              type="text"
              name="todoDescription"
              placeholder="Todo description"
              className="form-control"
              value={todoDescription || ``}
              onChange={event =>
                setTodoDescription(event.target.value)}
            />
          </div>
          <div className="form-group">
            <label htmlFor="todoDateCreated">
              Created on: 
            </label>
            {editingTodo ? (
              `${new Date(
                props.todo.todoDateCreated
              ).toLocaleDateString()} @ ${new Date(
                props.todo.todoDateCreated
              ).toLocaleTimeString()}`
            ) : (
                <DateCreated
                  dateCreated={props.todo ?
                    props.todo.todoDateCreated : null}
                  updateDateCreated={dateCreated =>
                    setTodoDateCreated(dateCreated)}
                />
              )}
          </div>
          {Object.getOwnPropertyNames(props.todo).length > 0 ? (
            <div className="form-group">
              <label htmlFor="todoCompleted">Completed: </label>
              <input
                type="checkbox"
                name="todoCompleted"
                checked={todoCompleted || false}
                onChange={e => setTodoCompleted(e.target.checked)}
              />
            </div>
          ) : null}
          <div className="form-group">
            <input
              type="submit"
              value="Submit"
              className={`btn ${!todoDescription ? `btn-danger` : `btn-
primary`}`}
          disabled={!todoDescription}
            />
          </div>
        </form>
      </div>
    </>
  );
```

```
TodoForm.propTypes = {
  todo: PropTypes.exact({
      todoDescription: PropTypes.string,
      todoDateCreated: PropTypes.string,
      todoCompleted: PropTypes.bool,
      _id: PropTypes.string
  }),
};

export default TodoForm;
```

App.jsx – just the TodosProvider and its children:

```
<TodosProvider>
  <Switch>
    <Route exact path="/"><AllTodos /></Route>
    <Route path="/add"><AddEditTodo /></Route>
    <Route path="/edit/:_id"><AddEditTodo /></Route>
    <Route><NotFound />
  </Switch>
</TodosProvider>
```

This is the end of QuickLab 22

and also the end of all of the QuickLabs.