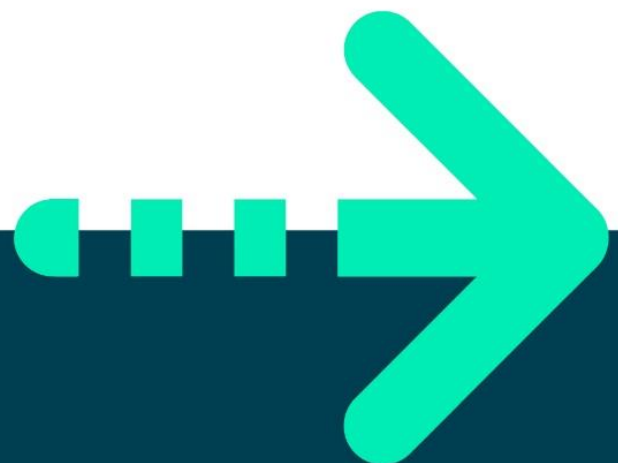




Building Web Applications using ReactJS

QuickLabs Guide





CONTENTS

QuickLabs Environment Set-Up	8
Code Editing	8
NodeJS	8
Do This Before Each QuickLab	8
Code Snippets	8
Quick Lab 1 – Get a ReactJS App Up and Running	10
Objectives	10
Overview	10
Activity	10
Quick Lab 2 – Build and Run the Application	11
Objectives	11
Overview	11
Activity	11
Quick Lab 3 – Create a Function Component	12
Objectives	12
Overview	12
Activity – Part 1 – MyComponent	12
Activity – Part 2 – AnotherComponent	12
Code Snippets	13
Quick Lab 4 - Creating a Class Component	15
Objectives	15
Overview	15
Activity	15
Code Snippets	16
Quick Lab 6 – Thinking in React Part 1 – Component Hierarchy	17
Objectives	17
Overview	17
Activity	17
Wireframes/Mock Ups	20
QuickLab 6 – Thinking in React - Part 1, Component Hierarchy - Sample Solution	21



All Todos	21
Add/Edit Todo	21
Quick Lab 7a – Create Common Header and Footer Components.....	22
Objectives	22
Overview	22
Activity – Header and Footer Acceptance Criteria	22
Desired Outcome	23
Activity – Header Stepped Instructions	23
Activity – Footer Stepped Instructions	23
Code Snippets	25
Quick Lab 8 – Exploring Props.....	26
Objectives	26
Overview	26
Activity – ComponentWithProps - Step-by-step	26
Activity – Using PropTypes – Step-by-step	27
Activity – Using defaultProps – Step-by-step	27
Activity – Supplying props from the Parent – Step-by-step	28
Code Snippets	30
Quick Lab 10a – Thinking in React Part 2 – A Static Version – Components with static data	31
Objectives	31
Overview	31
Desired Outcome	32
Activity 1 – The Todo component	32
Activity 2 – The AllTodos component	32
Activity 3 – Render the AllTodos component	33
Code Snippets	33
Quick Lab 10c – Thinking in React Part 2 – A Static Version – Adding a Form	36
Objectives	36
Overview	36
Desired Outcome	37
Activity 1 – Create the TodoForm Component	37



Activity 2 – Create the AddEditTodo Component	38
Activity 3 – Add the new components to the app	38
Code Snippets	39
Quick Lab 11 - Thinking in React Part 3, Identifying State in the Todo Application	41
Objectives	41
Activity	41
Quick Lab 11 – Thinking in React Part 3 – Identifying State in the Todo Application	42
EXAMPLE SOLUTION	42
Quick Lab 12 – Thinking in React Part 4 – Identifying where State should live in the Todo Application	43
Objectives	43
Activity – List of todos	43
Activity – All properties of a ‘new’ or ‘updated’ Todo	43
Quick Lab 12 – Thinking in React Part 4 – Identifying where State should live in the Todo Application	44
Example Solution	44
QuickLab 13a – Thinking In React Part 4 – Adding State	45
Objectives	45
Overview	45
Activity – Add state to the App Component	45
Activity – Make AllTodos use the todos from props	45
Activity – Add state to the TodoForm Component	46
Code Snippets	47
QuickLab 14a – Thinking In React Part 5 – Adding Inverse Data Flow	50
Objectives	50
Overview	50
Activity – Add a submit handler to TodoForm	50
Activity – Add a submit handler to AddEditTodo	51
Activity – Add a submit handler to App	51
Code Snippets	53
QuickLab 15 – External Data – Using useEffect.....	56
Objectives	56



Overview	56
Activity	56
Code Snippets - App.js	56
QuickLab 16 - Installing JSON server.....	57
Objectives	57
Activity	57
QuickLab 17a - Use an External Service – Getting Data	58
Objectives	58
Overview	58
Code Snippets	60
QuickLab 18a - Use an External Service – Sending Data	63
Objectives	63
Overview	63
Code Snippets	65
QuickLab 19a – Routing the Application.....	67
Objectives	67
Overview	67
Code Snippets	69
QuickLab 20 - Use parameterised routes	71
Objectives	71
Overview	71
Code Snippets	74
QuickLab 21 – State management – Getting Data	77
Objectives	77
Overview	77
QuickLab 22 – State management – Dispatching Data	84
Objectives	84
Overview	84
Activity 2 – Editing actions in the Provider	85
Activity 3 – Adding and Editing actions in the AddEditTodo Component	85
Activity 4 – Adding and Editing actions in the TodoForm Component	86
Activity 5 – Adding and Editing routes in the App Component.	87



Code Snippets	89
Quick Lab 5 - Jest	95
Quick Lab 7b – Write Snapshot Tests.....	96
Objectives	96
Overview	96
Activity – Test the Header	96
Activity – Test the Footer	96
Activity – Ensure the tests fail	97
Code Snippets	97
Quick Lab 9 – Testing Props	98
Objectives	98
Overview	98
Activity – Tests Set Up	98
Activity 1 – Test the header prop	98
Activity 2 – Test the content prop	99
Activity 3 – Test the number prop	99
Code Snippet	100
Quick Lab 10b – Thinking in React Part 2 – A Static Version – Testing Components with static data	101
Objectives	101
Overview	101
Activity 1 – Write Tests for the AllTodos Component	101
Activity 2 – Write Tests for the Todo Component – className	102
Activity – Write Tests for the Todo Component – Action render	102
Code Snippets	103
Quick Lab 10d – Thinking in React Part 2 – A Static Version – Test the Form rendering.....	105
Objectives	105
Overview	105
Activity	105
Code Snippets	106
QuickLab 13a – Thinking In React Part 4 – Adding State - Testing Sections That Were Removed.....	107



Activity – Make AllTodos use the todos from props	107
Activity – Add state to the TodoForm Component	107
QuickLab 13b – Thinking In React Part 4 – Testing Event Handlers.....	109
Objectives	109
Overview	109
Activity – Test the onChange event on the todoDescription input	109
Code Snippets	111
QuickLab 14b – Thinking In React Part 5 – Testing Form Submission	112
Objectives	112
Overview	112
Activity 1 – Adding the submitTodo prop to TodoForm	112
Activity 2 – Testing Form Submission	112
If you have time...	113
Code Snippets	114



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:

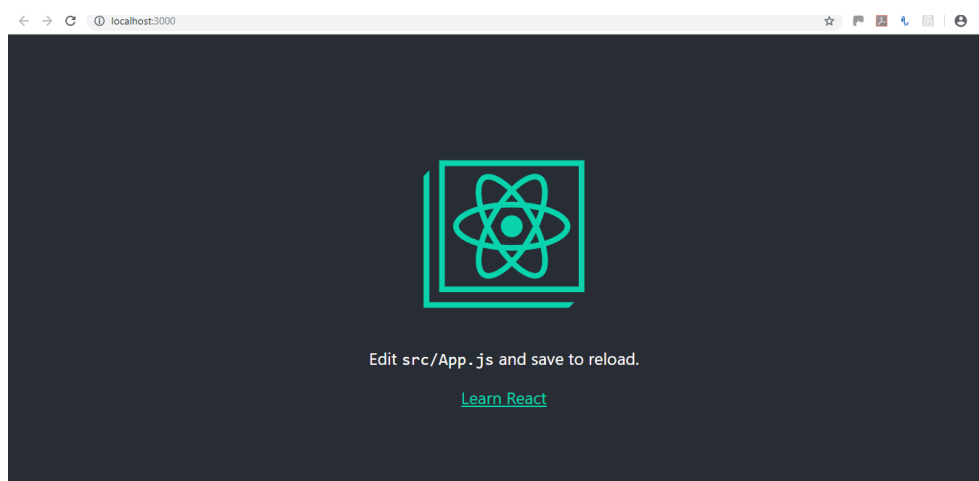
```
npx create-react-app starter
```

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:



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 = () => { // 6
  return (
    <>
      <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 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.

- 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
<div><div><input checked="" type="checkbox"/> Acceptance Criteria Delete</div><div><div>0%</div><div><div><div><input type="checkbox"/> The UI should be created using ReactJS and JSX</div><div><input type="checkbox"/> Every UI should have a <footer></div><div><input type="checkbox"/> The <footer> section should use the Bootstrap CSS classes to automatically set the top margin and left and right padding to 1rem</div><div><input type="checkbox"/> The text in the footer should be centered using the "container" and "align-center" Bootstrap CSS classes</div></div><div>You have unsaved edits on this field. View edits - Discard</div><div><input type="checkbox"/> The 'footer' should contain the text '© QA Ltd 2019.'</div></div></div></div>	<div><div><input checked="" type="checkbox"/> Acceptance Criteria Delete</div><div><div>0%</div><div><div><div><input type="checkbox"/> The UI should be created using ReactJS and JSX</div><div><input type="checkbox"/> 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</div><div><input type="checkbox"/> The <nav> should contain the QA Logo having alt text of "QA Ltd" and a width of 100</div><div><input type="checkbox"/> 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.</div><div><input type="checkbox"/> 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</div></div></div></div></div>



Acceptance Criteria – Specific UIs

All Todos	Add/Edit Todos
<div><div><input checked="" type="checkbox"/> Acceptance Criteria Delete</div><div><div>0%</div><ul style="list-style-type: none"><input type="checkbox"/> The UI should be created using ReactJS and JSX<input type="checkbox"/> All Todos UI should display a title of 'Todos List'<input type="checkbox"/> Todos List should be presented in a striped table.<input type="checkbox"/> Todos List table should have 3 columns: "Description", "Date Created" and "Action".<input type="checkbox"/> Each Todo in the list should be presented as a row in the table<input type="checkbox"/> Completed Todos should be struck through and an action of "N/A"<input type="checkbox"/> Todos not completed should have an action of "Edit" which is a link</div></div>	<div><div><input checked="" type="checkbox"/> Acceptance Criteria Delete</div><div><div>0%</div><ul style="list-style-type: none"><input type="checkbox"/> The UI should be created using ReactJS and JSX<input type="checkbox"/> Add/Edit UI should be wrapped in a <div> with a class of addEditTodo and display a title of a <h3> with text 'Add/Edit Todo'.<input type="checkbox"/> Todo to add or edit should be presented in a <form>.<input type="checkbox"/> The todoDescription should be wrapped in a <div> with a Bootstrap CSS class of "form-group"<input type="checkbox"/> The todoDescription should have a <label> with the text 'Description:'<input type="checkbox"/> The todoDescription should be an <input> with a 'type' of "text", a 'name' of "todoDescription" and a Bootstrap CSS class of "form-control".<input type="checkbox"/> The todoDateCreated should be wrapped in a <div> with a Bootstrap CSS class of "form-group".<input type="checkbox"/> The todoDateCreated should have a <label> with the text 'Created on:' and display the current date and time in a next to it.<input type="checkbox"/> The submit button should be wrapped in a <div> with a Bootstrap CSS class of "form-group".<input type="checkbox"/> The todoCompleted should be wrapped in a <div> with a Bootstrap CSS class of "form-group".<input type="checkbox"/> The todoCompleted should have a <label> with the text 'Completed:'<input type="checkbox"/> The todoCompleted should be an <input> with a 'type' of "checkbox", a 'name' of "todoCompleted" and a Bootstrap CSS class of "form-control"<input type="checkbox"/> 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".</div></div>



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


 **Todo App**

Todos List

Description	Date Created	Action
Sample Todo 1	Sat, 04 May 2019 15:00:00 GMT	N/A
Sample Todo 2	Sat, 04 May 2019 15:30:00 GMT	N/A
Sample Todo 3	Sat, 04 May 2019 15:45:00 GMT	Edit
Sample Todo 4	Sat, 04 May 2019 16:00:00 GMT	Edit

© QA Ltd 2019-

Add/Edit Todo

 **Todo App**

Add/Edit Todo

Description:

Todo description

Created on: 09/12/2019 @ 12:22:23

Completed: ☐

Submit

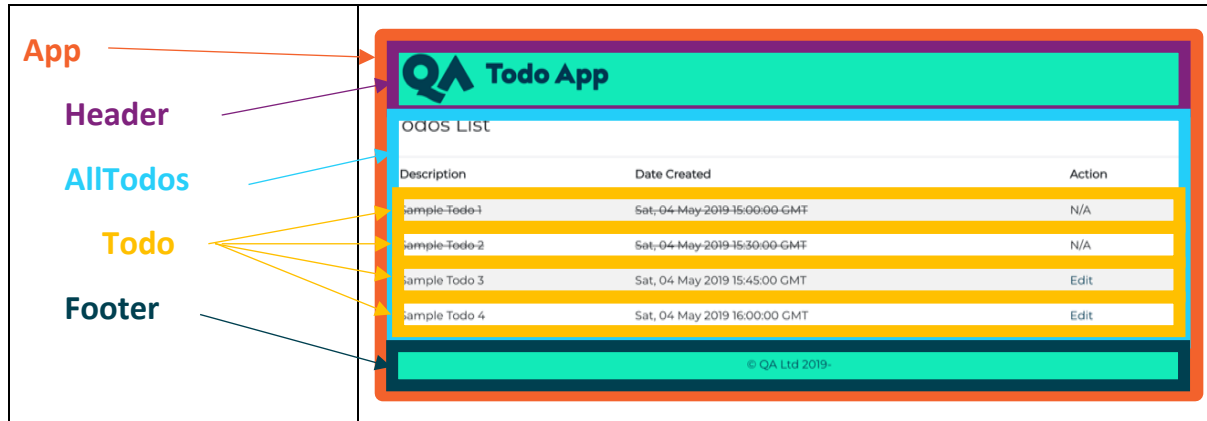
© QA Ltd 2019-

This is the end of Quick Lab 6

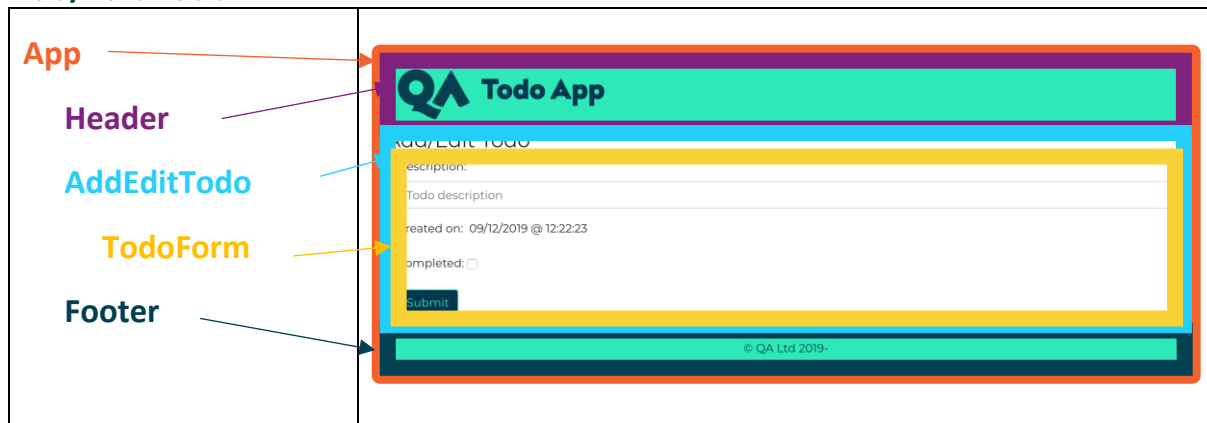


QuickLab 6 – Thinking in React – Part 1 – Component Hierarchy – Sample Solution

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.

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

Footer	Header
<div><div><input checked="" type="checkbox"/> Acceptance Criteria Delete</div><div><div>0%</div><div><div><input type="checkbox"/> The UI should be created using ReactJS and JSX</div><div><input type="checkbox"/> Every UI should have a <footer></div><div><input type="checkbox"/> The <footer> section should use the Bootstrap CSS classes to automatically set the top margin and left and right padding to 1rem</div><div><input type="checkbox"/> The text in the footer should be centered using the "container" and "align-center" Bootstrap CSS classes</div></div><div>You have unsaved edits on this field. View edits - Discard</div><div><input type="checkbox"/> The 'footer' should contain the text '© QA Ltd 2019-'</div></div></div>	<div><div><input checked="" type="checkbox"/> Acceptance Criteria Delete</div><div><div>0%</div><div><div><input type="checkbox"/> The UI should be created using ReactJS and JSX</div><div><input type="checkbox"/> 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</div><div><input type="checkbox"/> The <nav> should contain the QA Logo having alt text of "QA Ltd" and a width of 100</div><div><input type="checkbox"/> 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.</div><div><input type="checkbox"/> 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</div></div></div></div>

- Import these into the App component to display them.



Desired Outcome



Other UIs to go here

© QA Ltd 2019-

Activity – Header Stepped Instructions

1. In `b-static-version/starter/src/Components` create a new file called `Header.jsx`.
2. Add an `import` for `React` from `react`.
3. Add an `import` of `logo` from `'./images/qaLogo.svg'`.
4. Create a **Function component** called `Header` that has *no parameters*.
5. 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`.
6. `export Header` as `default`.
7. Save the file.
8. Open `App.js` from `b-static-version/src` for editing.
9. Add an `import` for `Header` from `./Components/Header`.
10. Within the outer `<div>`, add a child of `<Header />` as an older sibling of the inner `<div>`.
11. 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/qa/logo.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 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 were 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.
1. 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 required in `ComponentWithProps`, but its value is `undefined`. index.js:1375
    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 required in `ComponentWithProps`, but its value is `undefined`. index.js:1375
    in ComponentWithProps (at MyComponent.jsx:8)
    in MyComponent (at App.js:6)
    in App (at src/index.js:7)

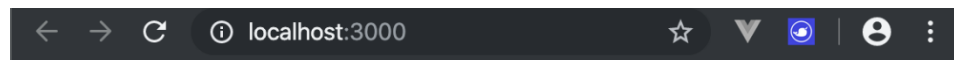
✖ Warning: Failed prop type: The prop `number` is marked as required in `ComponentWithProps`, but its value is `undefined`. index.js:1375
    in ComponentWithProps (at MyComponent.jsx:8)
    in MyComponent (at App.js:6)
    in App (at src/index.js:7)
```

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.



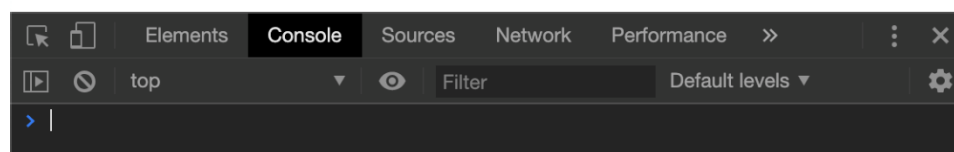
Hello World

Header from defaults

Content from defaults

This is a number from props: 10

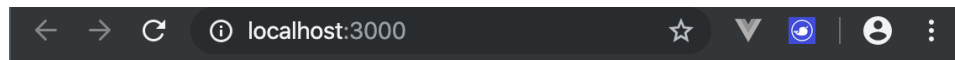
This is a display of a prop that doesn't exist:



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.



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
    <> // CWP3.2
      <h1>{props.header}</h1> // CWP3.3
      <p>{props.content}</p> // CWP3.4
      <p>
        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
    <> // 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 10a – Thinking in React Part 2 – A Static Version – Components with static data

Objectives

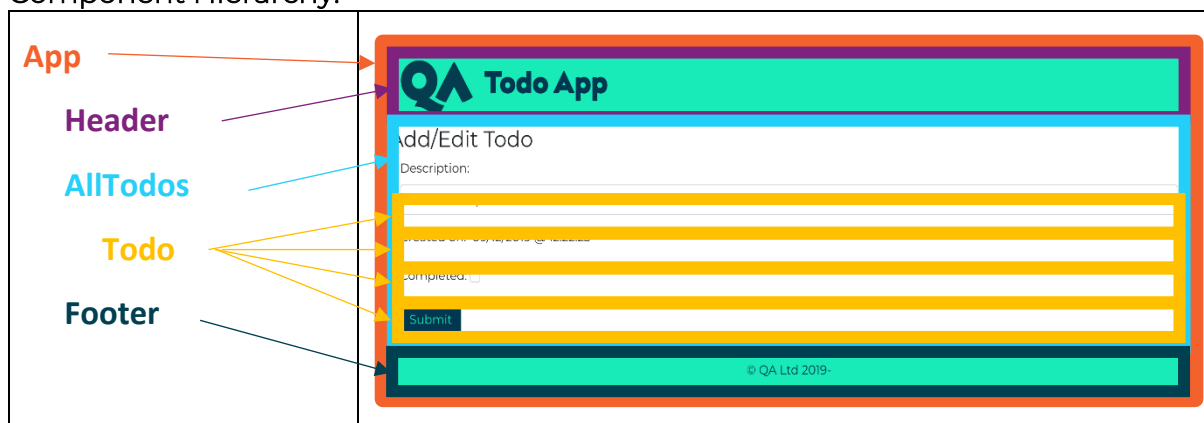
- To be able to use static external data to populate components
- To be able to use the map function to create multiple components
- To be able to conditionally render items dependent on some value

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:



☒ **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



Desired Outcome

QA Todo App		
Todos List		
Description	Date Created	Action
Sample-Todo-1	Sat, 04 May 2019 15:00:00 GMT	N/A
Sample-Todo-2	Sat, 04 May 2019 15:30:00 GMT	N/A
Sample Todo 3	Sat, 04 May 2019 15:45:00 GMT	Edit
Sample Todo 4	Sat, 04 May 2019 16:00:00 GMT	Edit
© QA Ltd 2019		

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}`.
1. Import `PropTypes` from `prop-types`.
2. Import `TodoModel` from `./utils/Todo.model`.
3. Set a `const dateCreated` to be a `new Date` that *parses* the `todo.todoDateCreated` and converts it to a `UTC` string.
4. Set a `const completedClassName` that is *conditionally* set to ``completed`` if `todo.todoCompleted` is `true` and an *empty string* if not.
5. Declare a variable `completed`.
6. Use an `if` statement to set `completed` to the string ``N/A`` if `todo.todoCompleted` is `true` and to the markup `Edit` if not.
7. 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.
8. 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.
9. 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 `classNames` `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 { // 1.9
    completed = <a href="/">Edit</a>
  }
  return (
    <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.todoDescription,
      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 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/utills`), 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.

<input checked="" type="checkbox"/> Acceptance Criteria	Delete
0%	
<input type="checkbox"/> The UI should be created using ReactJS and JSX	
<input type="checkbox"/> Add/Edit UI should be wrapped in a <code><div></code> with a class of <code>addEditTodo</code> and display a title of a <code><h3></code> with text 'Add/Edit Todo'.	
<input type="checkbox"/> Todo to add or edit should be presented in a <code><form></code> .	
<input type="checkbox"/> The <code>todoDescription</code> should be wrapped in a <code><div></code> with a Bootstrap CSS class of <code>"form-group"</code>	
<input type="checkbox"/> The <code>todoDescription</code> should have a <code><label></code> with the text 'Description:'	
<input type="checkbox"/> The <code>todoDescription</code> should be an <code><input></code> with a 'type' of <code>"text"</code> , a 'name' of <code>"todoDescription"</code> and a Bootstrap CSS class of <code>"form-control"</code> .	
<input type="checkbox"/> The <code>todoDateCreated</code> should be wrapped in a <code><div></code> with a Bootstrap CSS class of <code>"form-group"</code> .	
<input type="checkbox"/> The <code>todoDateCreated</code> should have a <code><label></code> with the text 'Created on:' and display the current date and time in a <code></code> next to it.	
<input type="checkbox"/> The submit button should be wrapped in a <code><div></code> with a Bootstrap CSS class of <code>"form-group"</code> .	
<input type="checkbox"/> The <code>todoCompleted</code> should be wrapped in a <code><div></code> with a Bootstrap CSS class of <code>"form-group"</code> .	
<input type="checkbox"/> The <code>todoCompleted</code> should have a <code><label></code> with the text 'Completed:'	
<input type="checkbox"/> The <code>todoCompleted</code> should be an <code><input></code> with a 'type' of <code>"checkbox"</code> , a 'name' of <code>"todoCompleted"</code> and a Bootstrap CSS class of <code>"form-control"</code>	
<input type="checkbox"/> The submit button should be an <code><input></code> with a 'type' of <code>"submit"</code> , have a 'value' of <code>"Submit"</code> and Bootstrap CSS classes of <code>"btn btn-primary"</code> .	



Desired Outcome

The screenshot shows the QA Todo App interface. At the top is a teal header with the QA logo and 'Todo App'. Below it is a 'Todos List' section containing a table with four rows of sample todos. Each row has columns for Description, Date Created, and Action. Below the table is an 'Add/Edit Todo' section with a 'Description:' label, a text input field with placeholder 'Todo description', a 'Date Created:' label with a timestamp, a 'Completed:' checkbox, and a 'Submit' button. At the bottom is a teal footer with '© QA Ltd 2019-'.

Description	Date Created	Action
Sample-Todo-1	Sat, 04 May 2019 15:00:00 GMT	N/A
Sample-Todo-2	Sat, 04 May 2019 15:30:00 GMT	N/A
Sample-Todo-3	Sat, 04 May 2019 15:45:00 GMT	Edit
Sample-Todo-4	Sat, 04 May 2019 16:00:00 GMT	Edit

Add/Edit Todo

Description:

Date Created: 09/12/2019 @ 11:51:42

Completed: ☐

© QA Ltd 2019-

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.

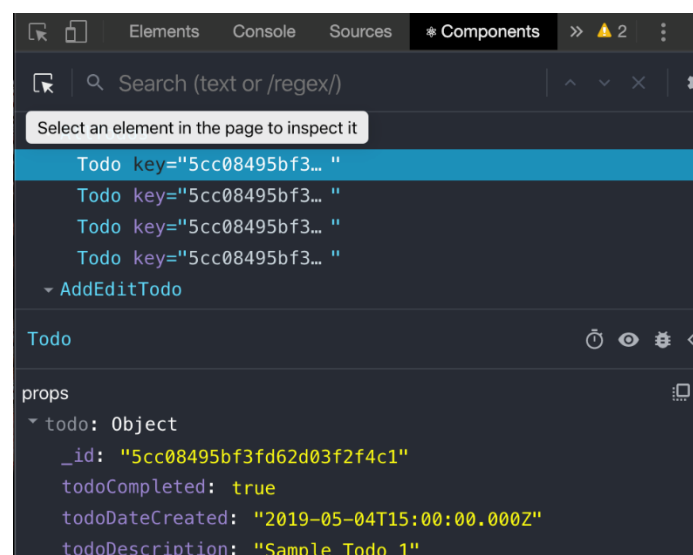
Activity 2 – Create the AddEditTodo Component

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:&nbsp; //
        </label> //
        <input type="text" name="todoDescription" //
          placeholder="Todo description" //
          className="form-control" /> //
        </div> //
        <div className="form-group"> //
          <label htmlFor="todoDateCreated"> //
            Date Created:&nbsp; //
          </label> //
          <DateCreated /> //
        </div> //
        <div className="form-group"> //
          <label htmlFor="todoCompleted"> //
            Completed:&nbsp; //
          </label> //
          <input type="checkbox" name="todoCompleted" /> //
        </div> //
        <div className="form-group"> //
          <input type="submit" value="Submit" //
            className="btn btn-primary" /> //
          </div> //
        </form> //
      </div> //
    ); //
  }; // 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 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:

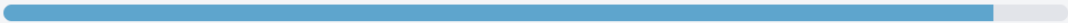
- The List of Todos;
- The Description property of a NEW todo;
- The Date Created property of a NEW todo;
- The Completed property of a NEW todo.

The **Acceptance Criteria** should be considered:


 **Description** Edit

As the PRESENTATIONAL LAYER, I want to be able to display all of the Todos in the same UI so that the user is able to see all Todos in a list

☒ **Acceptance Criteria** Show checked items (13) Delete


93% 

☐ If a new todo is added within the application, this should be displayed in an update list of todos when the All Todos UI is shown

 **Description** Edit

As the PRESENTATIONAL LAYER, I want to be able to display an empty form so that a user can add a new todo

☒ **Acceptance Criteria** Show checked items (10) Delete

91% 

☐ Submitting the form should update the list of todos held in the application

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.

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 **sampleTodos**.
5. In the **return**, add a **prop** to the **AllTodos** component called **todos** 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 `props` by adding this to the *arrow function* arguments.
4. Modify the `map` function to use `props.todos` rather than `sampleTodos`.
5. Save the file.

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. 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.



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 todos={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 = props => {                                     // 3
  const todos = props.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>
  );
};
export default AllTodos;
```

Components/AllTodos.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

  const disabled = !todoDescription;

  return (
    <form>
      <div className="form-group">
        <label htmlFor="todoDescription">Description:&nbsp;</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:&nbsp;</label>
        <DateCreated

```



```
        updateDateCreated={dateCreated =>
          setTodoDateCreated(dateCreated) // 5
        }
      </div>
      <div className="form-group">
        <label htmlFor="todoCompleted">Completed:&nbsp;</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 btn-primary"
          disabled={disabled}
        />
      </div>
    </form>
  );
};

export default TodoForm;
```

This is the end of Quick Lab 13a



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`, `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` 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);
  }

  const disabled = !todoDescription;

  return (
    <form onSubmit={handleSubmit}>
      <div className="form-group">
        <label htmlFor="todoDescription">Description:&nbsp;</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:&nbsp;</label>
        <DateCreated updateDateCreated={dateCreated =>
          setTodoDateCreated(dateCreated)}
        />
      </div>
      <div className="form-group">
        <label htmlFor="todoCompleted">Completed:&nbsp;</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 btn-danger"
          disabled={disabled}
        />
      </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,
      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



Labs Check to this point (12:27, 23/04)



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 `sampleTodos` array.

Activity

1. Open `App.js` for editing.
1. Add `useEffect` to the list of imports from `React` (it should be placed inside the `{}`).
2. Change the value of the `useState` call in the Component to an *empty array*.
3. Add a call to `useEffect` – it takes an *arrow function* that executes a call to `setTodos` using the `sampleTodos` and an *empty array* as a second argument.

Save the file and run the application. It will load the todos but not allow adding.

Code Snippets - App.js

```
import React, { useState, useEffect } 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';
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([]);              // 3
  useEffect(() => {                                     // 4
    console.log(`useEffect called`);
    setTodos(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 download the 0 version of the JSON server globally:

```
npm -g install json-server@0
```

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 17a - 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. After the `useEffect`, declare a `const` called `getTodos`.
8. 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`;
- Set a `const` called `todos` to `await` the `return` of `res.data`;
- Calls `setTodos` with `todos`;



- Catches an error called `e` and:
 - Calls `setTodos` with `e.message`.
9. In the `useEffect` function, remove the call to `setTodos` and replace it with a call to `getTodos`.
 10. Save the file.

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

1. In `App.js`, add 2 further `states`:
 - `onlineStatus` to confirm that data was retrieved from the network, initially `false`;
 - `loading` to indicate if the data has been requested before a response is received, initially `true`.
2. Before the `try` block in `getTodos()`, change the value of `loading` to `true`.
3. In the `try` block, after the `setTodo()` call, change the value of `loading` to `false` and the value of `onlineStatus` to `true`.
4. In the `catch` block, change the value of `loading` to `false` and `onlineStatus` to `false`.
5. In the return of the `App` function, in the `<div>` under the `<Header />` add a conditional render of a `<h3>` with content of "The data server may be offline, changes will not be saved" dependent on whether `onlineStatus` AND `loading` are both `false`, `null` should be rendered if this is NOT the case.
6. Add an attribute of `loading` with a value of `{loading}` to the `AllTodos` component call.
7. Save the file.

Activity 3 - Add 'no-data' error handling to the AllTodos component

1. Open `Components/AllTodos.jsx` for editing.
2. Add a declaration of a variable called `todos` as the first line of the arrow function for the component set to an empty array.
3. Remove the `const` declaration on `todos`.
4. Wrap the setting of `todos` in a conditional statement on whether `props.todos` has a `length` property AND that the type of the first entry in the array is not a string (if `todos` are present, the first element in the array should be an object).
 - If these conditions return `true`, use the existing map function on `todos` to set `todos` to a `Todo` component that has attributes of `todo` set to the `current todo` and the `key` set to the `_id` of the `current todo`;



- Add an **else if** clause that checks the status of **props.loading** and if **true**:
 - Push a **<tr>** with a **key** of **loading** that contains a **<td>** that has a **colspan** of **3** and content of **"Please wait - retrieving the todos"**;
- Add an **else** that:
 - Pushes a **table row** with **key** of **error** and a child **<td>** that has a **colspan** of **3** and the content **"There has been an error retrieving the todos array"**.

5. Save the file and run the application.

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

6. Locate the command line that is running JSON Server and stop it (using CTRL+C).
7. Refresh the application and you should see the **error message** in place of the **todos**.
8. Add a **setTimeout** that lasts **5 seconds** around the **getTodos()** call in **useEffect** in **App.js** - you should see the **"loading"** message until the data is returned.

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 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 [onlineStatus, setOnlineStatus] = useState(false); // 2.1
  const [loading, setLoading] = useState(true); //

  useEffect(() => { // 1.9
    setTimeout(() => { // 3.8
      getTodos(); // 1.9
    }, 5000) // 3.8
  }, []);

  const getTodos = async () => { // 1.7
    setLoading(true); // 1.8, 2.2
    try { //
      const res = await axios.get(TODOSURL); //
      const todos = await res.data; //
      setTodos(todos); //
    } catch { //
    }
  }
}
```

```

        setLoading(false); // 2.3
        setOnlineStatus(true); // 2.3
      } catch (e) { //
        setTodos(e.message); //
        setLoading(false); // 2.4
        setOnlineStatus(false); // 2.4
      } //
    }; //

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

    // Continued on next page

    return (
      <div className="container">
        <Header />
        <div className="container">
          {!onlineStatus && !loading ? ( // 2.5
            <h3>The data server may be offline, changes will not be //
              saved</h3> //
          ) : null}
          <AllTodos todos={todos} loading={loading} /> // 2.6
          <AddEditTodo submitTodo={submitTodo} />
        </div>
        <Footer />
      </div>
    );
  }

  export default App;

```

AllTodos.js

```

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

const AllTodos = props => { // 3.1
  let todos = []; //

  if (props.todos.length && typeof props.todos[0] !== "string") { // 3.2
    todos = props.todos.map(currentTodo => {
      const todo = new TodoModel(
        currentTodo.todoDescription,
        currentTodo.todoDateCreated,
        currentTodo.todoCompleted,
        currentTodo._id);

      return <Todo todo={todo} key={todo._id} />
    });
  } else if (props.loading) {
    todos.push(
      <tr key="loading">
        <td colSpan="3">Please wait - retrieving the todos</td>
      </tr>
    );
  } else {
    todos.push(
      <tr key="error">
        <td colSpan="3">There has been an error retrieving the todos</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>{todos}</tbody>
    </table>
  </div>
);
};
export default AllTodos;
```

This is the end of QuickLab 17a



QuickLab 18a - 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 or extend your previous Todo application.

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. Under the `getTodos` declaration, declare `postTodo` as an `async arrow function` that takes `todo` as an argument with:
 - A try block that:
 - `await` an `axios post` call providing `TODOSURL` and `todo` as arguments;
 - Calls `setOnlineStatus` with `true`;
 - Calls `getTodos()`;
 - A catch block that takes `e` as an argument and sets `onlineStatus` to `false`.
4. Save the file.
- 5.

Activity 2 - Make the submitTodo function use the post calls

1. Replace the body of the `submitTodo` function in `App.js` with a call to `postTodo`.
2. Save the file and return to the browser.
3. Try to add a new `todo` ensuring that they are persisted when the



application is reloaded.



If you have time...

Write functionality for the application that would allow the editing of a particular **todo**. The solution for this can be found in the folder **ql18-solution-extended**. Comments in the files here explain why each step has been taken.

See where you may be able to make components more type-safe using **PropTypes** or have cleaner code through *refactoring* and *destructuring*.

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';
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 [onlineStatus, setOnlineStatus] = useState(false);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    setTimeout(() => {
      getTodos();
    }, 5000)
  }, []);

  const getTodos = async () => {
    setLoading(true);
    try {
      const res = await axios.get(TODOSURL);
      const todos = await res.data;
      setTodos(todos);
      setLoading(false); 2.3
      setOnlineStatus(true);
    } catch (e) {
      setTodos(e.message);
      setLoading(false);
      setOnlineStatus(false);
    }
  };

  const postTodo = async todo => {                                     // 1.3
    try {                                                            //
      await axios.post(TODOSURL, todo);                             //
      setOnlineStatus(true);                                         //
      getTodos();                                                    //
    } catch (e) {                                                    //
      setOnlineStatus(false);                                        //
    }                                                                //
  };                                                                //

  // Continued on next page
```

```
const submitTodo = todo => {  
  postTodo(todo); // 2.1  
}  
  
return (  
  <div className="container">  
    <Header />  
    <div className="container">  
      {!onlineStatus && !loading ? (  
        <h3>The data server may be offline, changes will not be  
        saved</h3>  
      ) : null}  
      <AllTodos todos={todos} loading={loading} />  
      <AddEditTodo submitTodo={submitTodo} />  
    </div>  
    <Footer />  
  </div>  
>);  
}  
  
export default App;
```

This is the end of QuickLab 18a



QuickLab 19a – 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 or extend your previous Todo application.

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@5.2.0
```

Note: The latest version of `react-router-dom` (6.8.1 at the time of writing) is incompatible with the step by step instructions given below. Consequently, you will need to install the earlier version of 5.2.0. This kind of issue is a common occurrence and locking versions of software to specific versions of packages is an entirely normal workaround. However, you can be sure we will investigate the differences between the incompatible router versions when you attend the live event.

3. Open `App.js` for editing.
4. Under the `import` for `React`, import **BrowserRouter as Router**, **Route** and **Switch** from `react-router-dom`.
5. 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. Replace the call for the `<AllTodos...>` component with a `<Route>` instead, it should have properties:
 - An `exact path` set to `'/'`;
 - A `render` whose callback *returns* the `AllTodos` component with the same attributes as the component being replaced.
3. Replace the call for `<AddEditTodo...>` with a `<Route>` with:
 - a `path` of `'/add'`;
 - A `render` whose callback *returns* the `AddEditTodo` component with attributes of a `submitTodo`, `todos` and `todoToUpdate`.
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.

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>` 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-item` and an `activeClassName`s 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!

Code snippets are on the next page



Code Snippets

App.js

```
import React, { useState, useEffect } from 'react';
import {
  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> // 1.4
      <div className="container"> // 1.5
        <Header />
        <div className="container">
          {!onlineStatus && !loading ? (
            <h3>The data server may be offline, changes will not be
              saved</h3>
            ) : null}
          <Switch> // 2.1
            <Route // 2.2
              exact
              path="/"
              render={() =>
                <AllTodos
                  todos={todos}
                  loading={loading}
                  selectTodo={selectTodo} />
              }
            />
            <Route // 2.3
              path="/add"
              render={() =>
                <AddEditTodo
                  submitTodo={submitTodo}
                  todos={todos}
                  todoToUpdate={todoToUpdate} />
              }
            />
          </Switch> // 2.1
        </div>
        <Footer />
      </div> // 1.5
    </Router>
  );
}

export default App;
```



Header.js

```
import React from "react";
import { Link, NavLink } from "react-router-dom"; // 3.1
import logo from "../images/qa/qa-logo.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" //
              > //
                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 19a



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. The **match** object supplied as a **prop** will be examined and the `_id` **param** will be used to display the todo in the form. The **useEffect** hook will be modified to check for the **match** object and the `params._id` value and this will also form the basis of the argument in the **find** function. So that the application navigates correctly, a **return** will be added to the **useEffect** hook to reset the state of **todo** to an empty object and set a new state of **submitted** to **false**. The **submitTodo** function will have to set the **submitted** state to **true** as part of its call. A **Redirect** will be conditionally rendered *if the form has been submitted*. So that the correct word is shown in the render, the declaration of **action** will be changed to use the **todo** and the **match** object to set **Add** or **Edit** in the title.

You should use the **e-routed-app/starter** folder or extend your previous Todo application.

Activity 1 – Add a Route for editing a Todo

1. Ensure that JSON server is running as in previous QuickLabs.
2. Open **App.js** for editing.
3. Under the **<Route>** for **'/add'** add another **<Route>** with:
 - A **path** set to **'/edit/:_id'**;
 - A **render** method that receives **props** as an argument and returns **<AddEditTodo />** with attributes:
 - **{...props}**;
 - **submitTodo** set to **{submitTodo}**;
 - **todos** set to **{todos}**.
4. Remove any line that references **todoToUpdate** or **setTodoToUpdate** as this is not needed anymore.
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 `` that surrounds 'Edit' for a `<Link></Link>` that has attributes:
 - `to` set to `{`/edit/${todo._id}`}`;
 - `className` set to `link`;
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

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. Add an import of `useParams` from `react-router-dom`.
3. Declare a `const { _id }` to be a call to the `useParams` hook.
4. In `useEffect` change the initial check logic to check for `props.todos` and `_id`.
5. In the `find` function for `todoEditing`, replace the parameter `props.todoToUpdate._id` with `_id`.
6. In the array of dependencies for `useEffect`, replace `props.todoToUpdate` with `_id`.

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

7. Add an import of `{ Redirect }` from `react-router-dom`.
8. In the `AddEditTodo` function, add a state of `submitted` with an initial value of `false`.
9. 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.
10. 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`.
11. 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:

12. Find the declaration of `action` and replace the *condition in the ternary statement* with `todo && props.match`.
13. 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';

/* Other imports omitted */

function App() {
  /* Function logic omitted */

  return (
    <Router>
      <div className="container">
        <Header />
        <div className="container">
          {!onlineStatus && !loading ? (
            <h3>The data server may be offline, changes will not be
              saved</h3>
          ) : null}
          <Switch>
            <Route
              exact path="/"
              render={() =>
                <AllTodos todos={todos} loading={loading} />
              }
            />
            <Route
              path="/add"
              render={() =>
                <AddEditTodo submitTodo={submitTodo} todos={todos} />
              }
            />
            <Route
              path="/edit/:_id"
              render={props =>
                <AddEditTodo
                  {...props}
                  submitTodo={submitTodo}
                  todos={todos}
                />
              }
            />
          </Switch>
        </div>
        <Footer />
      </div>
    </Router>
  )
}

default export App;
```



Todo.jsx

```
import PropTypes from 'prop-types';
import { Link } from 'react-router-dom'; // 2.2
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` : ``;
  return (
    <tr>
      <td className={completedClassName}>{todoDescription}</td>
      <td className={completedClassName}>{dateCreated}</td>
      <td>
        {todo.todoCompleted ? (
          `N/A`
        ) : (
          <Link className="link" to={`/edit/${todo._id}`}> // 2.3
            Edit //
          </Link> //
        )}
      </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"; // 3.2, 7
// 3.6
import './css/AddEditTodo.css';
import TodoForm from './TodoForm';

const AddEditTodo = props => {
  const [todo, setTodo] = useState({});
  const [submitted, setSubmitted] = useState(false); // 3.8
  const { _id } = useParams(); // 3.3

  useEffect(() => {
    // Edited to use the parameter from the url
    if (props.todos && _id) { // 3.4
      const todoEditing = props.todos.find( // 3.5
        todoToCheck => todoToCheck._id === _id //
      );
      setTodo(todoEditing);
    }

    // Add for correct navigation // 3.9
    return () => { //
      setTodo({}); //
      setSubmitted(false); //
    }; //
  }, [props.todos, _id]); // 3.6

  const submitTodo = submittedTodo => {
    props.submitTodo(submittedTodo);
    setTodo({});
    setSubmitted(true); // 3.10
  };
};
```

```
};  
// const action = props.todoToUpdate &&  
//   Object.getOwnPropertyNames(props.todoToUpdate).length  
const action = todo && props.match ? `Edit` : `Add`; // 3.12  
return (  
  <  
    {submitted || !props.todos.length ? // 3.11  
      <Redirect to="/" /> //  
      :  
      <div className="addEditTodo row">  
        <h3>{action} Todo</h3>  
        <TodoForm todo={todo} submitTodo={submitTodo} />  
      </div>  
    }  
  </>  
>);  
}  
AddEditTodo.propTypes = {  
  submitTodo: PropTypes.func.isRequired  
}  
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 and reducers to abstract application state management away from application components

Overview

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 and reducers 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 **TodosDispatchProvider**. A *reducer function* should be created that will take current **state** and an **action** to *return a new state*, based on the *action*. In the first instance, you will create an **allTodos** action that returns the *previous state* along with the *action payload*. Unknown *actions* should be dealt with by returning an *error object* with an appropriate message.

An *async* function will be used to make a call to the RESTful service, returning the *todos* data or an appropriate *error object* if this does not succeed.

The actual **TodosProvider** component will receive its **children** as **props**. It will need to use React's **useReducer** hook to define **state** and **dispatch**, passing in the *reducer function* and an *empty object*. The **useEffect** hook will be used to make an **async** call to *get the todos*, generating a **payload** from a call to the *async* function that *gets all of the todos* and calling **dispatch** with an *object* that has key-value pairs of **type** of **allTodos** and **payload**.

The component will be returned as its **children**, wrapped in a **TodosStateContext.Provider** component that has a property **value** set to **{state}**.

A *function* 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 **TodoStateContext**. The **context** *should be returned* as long as it is not **undefined**, in which case an *error* should be returned.

All logic will be removed from the **App** component and the **Routes** will be altered to display the **component** without the need for the **render** function as *no props will be passed in*. The **Switch** component will be wrapped in a **TodosProvider** component.

The **AllTodos** component will be modified to call the **useTodosState** function from the provider file and the logic modified so that *props are not used*. Other state and logic will be removed from this component as what it needs can be deduced from the provided context.

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 `TodoProvider.jsx`.
3. Import `React` and `{ createContext, useContext, useEffect, useReducer, }` from `react` and `axios` from `axios`.
4. Declare a constant `TodosStateContext` set to a call to `createContext()`.
5. Set a `const baseUrl` to ``http://localhost:4000/todos``.

Activity 2 – The `todosReducer` and `useTodosState` functions

1. Export a function called `useTodosState` that:
 - Declares a `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:
 - Set a variable called `todosData` to await an `axios.get` call to `baseUrl`;
 - Sets `todosData` to `todosData.data`;
 - Returns `todosData`;
 - Catches an `error` and returns an *object* with a *key* of `error` and a *value* of `"Data not available from server"`.
3. Declare an arrow function called `todosReducer` that takes arguments of `state` and `action` and has a body that has:
 - A `switch` on `action.type`:
 - For a case of `"allTodos"` return an *object* that has a *spread of state* and a *key* `todos` set to `action.payload`;
 - For the `default` case, throw a new `Error` with a *message* of `"Unhandled action type:"` along with the *actual action type*;



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 `useReducer`, passing in `todosReducer` and *an empty object, deconstructed to array constants* `state` and `dispatch`.
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 `dispatch` passing in an object with a key `type` set to `allTodos` and a `payload` key/value pair;
 - Calls `getTodos`.
4. The component should return a `TodosStateContext.Provider` with a property of `value` set to `state`.
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 render for server offline*.
4. Replace the *all of the routes* with *single route* that has an `exact path` of `"/` and a `component` `AllTodos`
5. Remove any *unused imports and const declarations* (i.e. `TODOSURL`).
6. Wrap the `Switch` component in a `TodosProvider` component, importing it from the file created earlier.
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`).
4. Change array of `todosToDisplay` to `todoComponents` (in all instances).



5. Remove current `if` statement and replace with one that:
 - Checks to see if `todos` is present and has a property called `error` and sets `todoComponents` to be an array containing a *single element* that is a **table row** with a **key** of `error` that wraps a **cell** with a `colSpan` of `3` and content of `todos.error`;
 - Alternatively, check to see if `todos` is present AND if its `length` has a value and set `todoComponents` to map the `todos` array:
 - The `currentTodo` passed in should be used to create an instance of the `TodoModel`.
 - Each instance should return a `Todo` element passing the `todo` in as a `todo` property and using its `_id` as a **key**;
 - Otherwise make `todoComponents` a *single element array* that is a **table row** with a **key** of `temporary` wrapping a **cell** that *spans 3 columns* with text of `Data is loading...`
6. Make the render use `todoComponents` instead of `todosToDisplay`.
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 tools to see the affect this has had on the way the application is rendered behind the scenes.



Code Snippets

TodosProvider.jsx

```
import React, { createContext, useContext, useEffect, useReducer } // 1.3
from 'react';
import axios from 'axios'; // 1.3

const TodosStateContext = createContext(); // 1.4
const TodosDispatchContext = createContext(); //
const baseUrl = `http://localhost:4000/todos`; // 1.5

const todosReducer = (state, action) => { // 2.3
  switch (action.type) {
    case `allTodos`: {
      return { ...state, todos: action.payload };
    }
    default: {
      throw new Error(`Unhandled action type: ${action.type}`);
    }
  }
};

const getAllTodos = async () => { // 2.2
  try {
    let todosData = await axios.get(baseUrl);
    todosData = todosData.data;
    return todosData;
  } catch (e) {
    return { error: `Data not available from server` };
  }
}

const TodosProvider = ({ children }) => { // 3.1
  const [state, dispatch] = useReducer(todosReducer, {}); // 3.2

  useEffect(() => { // 3.3
    const getTodos = async () => {
      const payload = await getAllTodos();
      dispatch({ type: `allTodos`, payload });
    };
    getTodos();
  }, []);

  return (
    <TodosStateContext.Provider value={state}> // 3.4
      {children} // 3.5
    </TodosStateContext.Provider>
  );
}

export function useTodosState() { // 2.1
  const context = useContext(TodosStateContext); //
  if (context === undefined) { //
    throw new Error(`useTodosState must be used within a TodosProvider`); //
  } //
  return context; //
}

export default TodosProvider; // 3.6
```



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 TodosProvider from './StateManagement/TodosProvider';

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

export default App;
```



AllTodos.jsx

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

const AllTodos = () => { // 5.2
  const { todos } = useTodosState(); // 5.3
  let todoComponents = []; // 5.4

  if (todos && todos.hasOwnProperty(`error`)) { // 5.5
    todoComponents = [
      <tr key="error"><td colspan="3">{todos.error}</td></tr>
    ];
  } else if (todos && todos.length) {
    todoComponents = todos.map(currentTodo => {
      const todo = new TodoModel(currentTodo.todoDescription,
        currentTodo.todoDateCreated, currentTodo.todoCompleted,
        currentTodo._id);
      return <Todo todo={todo} key={todo._id} />
    });
  } else {
    todoComponents = [
      <tr key="temporary"><td colspan="3">Data is loading...</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>{todoComponents}</tbody> // 5.6
      </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 add further actions to the todosReducer to add a todo and to edit a todo. You will define a useTodosDispatch function that will use the TodosDispatchContext. This will be used in the TodoForm component and this will be modified so that it deals with the submitting of the form, rather than passing down handler functions from parent components. As that functionality partly lives in the AddEditTodo component currently, you will need to modify this component to remove logic for submitting. The todo state will be handled in this form, so it will need utilise useTodosState. The App component will need routes to allow adding and editing of todos with the new functionality.

You should use the **f-state-management/starter** folder, **ql21-solution** or extend your previous Todo application.

Activity 1 – Adding actions in the Provider

1. Open TodosProvider.jsx for editing.
2. Add a constant **TodosDispatchContext** and set it to a call to **createContext**;
3. In the **todosReducer**, add another case for an **`addTodo`** action it should:
 - Call a function called **addTodo** with **action's payload** (this function is yet to be written);
 - Return an *object* that has **todos** as a **key** and a **value** that is an *array* that has a **spread of state.todos** and **action's payload**.
4. Create an **async** arrow function called **addTodo**, it should:
 - Receive a **newTodo**;
 - Try to set a variable called **addTodoSuccess** by *awaiting* a call to **axios.post** to the **baseUrl** with the **newTodo** as the body;
 - Returns **addTodoSuccess.data**;
 - Catch an **error** returning an *object* with **error** as a **key** and a **value** of **`Todo not added on server`**.
5. Export a new function called **useTodoDispatch**, 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 `useTodoDispatch must be used within a TodosProvider` otherwise return context.
6. Insert a `TodosDispatchContext.Provider` between the render of the `{children}` and the `TodosStateContext.Provider` in the `return` of the `TodosProvider`.
 7. Save the file.

Activity 2 – Editing actions in the Provider

1. Add another `case` to the `switch` in `todosReducer`, it should be for ``editTodo`` and:
 - Call a function called `editTodo` with `action`'s `payload` (yet to be written);
 - Declare a variable called `todos`, set to *collect the current state of todos* in an array;
 - Set a variable called `updatedTodos` to a `map` of `todos`, passing in the `currentTodo` and if the `currentTodo`'s `_id` matches the `_id` of `action.payload` it returns the `action.payload` otherwise returning the `currentTodo`;
 - Return an *object* who's `todos` key is set to `updatedTodos`.
2. Add an arrow function called `editTodo`, that is `async` and receives an `editedTodo`.
3. The body of the function should:
 - Try to set a variable called `editTodoSuccess` by awaiting the result of an `axios.put` call to the `baseURL/editedTodo's _id` and a body of `editedTodo` and then return `editTodoSuccess`'s `data`;
 - Catch an `error` and return an *object* with `error` as a `key` and a value of ``Todo not updated``.
4. Save the file.

Activity 3 – Adding and Editing actions in the AddEditTodo Component

1. Open `AddEditTodo.jsx` for editing.
2. Remove the imports for `PropTypes` and `{ Redirect }`.
3. Add an import for `{ useTodosState }` from `TodosProvider`.
4. Replace the `submitted` state with an `action` state initially set to ``Add``.
5. Destructure an object to `todos` by calling `useTodosState`.

6. In the `if` block in `useEffect`:
 - Remove the `props.` from `todos` and add a further AND condition of `todos.find` to the `if` statement.
 - Call `setAction` with the string ``Edit``;
 - Remove `props` from the `todos.find` call;
 - Change the `setSubmitted` state call to a call to `setAction` with the string ``Add`` in the `return`.
7. In the *dependency array* for `useEffect`, remove `props` from the `todos` whilst leaving `props.match` intact.
8. Remove the `submitTodo` function and the declaration and setting of the `const action`.
9. Clear the mark up between the *Fragment* and replace it with:
 - A `<div>` with `classNames addEditTodo` and `row`;
 - An inner `<h3>` component that renders `{action}` and the text `Todo`;
 - A sibling `<TodoForm>` component with a property of `todo` set to `todo` and a property of `action` set to the `action` state run over with `toLowerCase` and the string `Todo`.
10. Remove the `propTypes` from the component.
11. Save the file.

Activity 4 – Adding and Editing actions in the TodoForm Component

1. Open `TodoForm.jsx` for editing.
2. Add imports for `{ Redirect }` from `react-router-dom`, `{ useTodosDispatch }` from `TodosProvider` and `TodoModel` from `Todo.model`.
3. In the *component code*, remove the state for `todoId` and add state for:
 - `editingTodo`, initially set to `false`;
 - `submitted`, initially set to `false`.
4. Add a `const` to set `dispatch` as a call to `useTodosDispatch`.
5. In the `useEffect`:
 - Remove `_id` from the *destructuring* of `props.todo`;
 - Remove the setting of `todoId` in the initial `if` block;
 - Add a call to `setEditingTodo` to make it `true`.
6. Transfer the `else` block into a `return` callback and add calls to set the state of `editingTodo` and `submitted` to `false`, removing the setting of

`todoId`.

7. Add an arrow function called `createTodo` that:
 - Sets `todoId` conditionally on whether `props.action` is `'addTodo'`, using `generateTodoId()` if it is and `props.todo._id` if it isn't;
 - Returns a new `TodoModel`, using the state for description, dateCreated and completed as arguments along with the `todoId` set in the last step.
8. Modify the `handleSubmit` function so that in its entirety it:
 - Stops the *default* action;
 - Sets a `const todo` to a call to `createTodo`;
 - Calls `dispatch` with an *object* that has a *key* of `type` set to `props.action` and a `payload` set to `todo`;
 - Calls `setSubmitted` with `true`.
9. Add an arrow function called `redirectAfterSubmit` that uses `submitted` as a condition to return a `Redirect` component with a `to` property of `"/` or `null`.
10. In the `return`, wrap the current mark up in a `React.Fragment` and add a call to `redirectAfterSubmit`.
11. In the `value` of `todoDescription` add an *OR clause* of an *empty string*.
12. After the `label` for `todoDateCreated`, change the condition from `props.todo.todoDateCreated` to `editingTodo`.
13. In the `value` of `todoCompleted` add an *OR clause* of `false`.
14. In the `propTypes`, remove the need for `submitTodo`.
15. Modify the `todo` type so it is *wrapped* with a `oneOfType` array, containing the exact object as the *first* element in the array and a `PropTypes.bool` as the *second*.
16. Save the file.

Activity 5 – Adding and Editing routes in the App Component.

1. Open `App.js` for editing.
2. In the `<Switch>`, after the existing `<Route>`:
 - Add a `Route` with a `path` of `/add` that has a `component` of `AddEditTodo`;
 - Add a `Route` with a `path` of `/edit/:_id` that has a `component` of `AddEditTodo`.
3. Save the file.



Run the application – you should find that it works fully as it did before.

Code Snippets

TodosProvider.jsx – updated todosReducer function:

```
const todosReducer = (state, action) => {
  switch (action.type) {
    case `allTodos`: {
      return { ...state, todos: action.payload };
    }
    case `addTodo`: { // 1.3
      addTodo(action.payload); //
      return { todos: [...state.todos, action.payload] }; //
    }
    case `editTodo`: { // 2.1
      editTodo(action.payload); //
      let todos = [...state.todos]; //
      let updatedTodos = todos.map(currentTodo => //
        currentTodo._id === action.payload._id //
        ? action.payload : currentTodo //
      ); //
      return { todos: updatedTodos }; //
    }
    default: {
      throw new Error(`Unhandled action type: ${action.type}`);
    }
  }
};
```

TodosProvider.jsx – addTodoFunction function:

```
const addTodo = async newTodo => { // 1.4
  try { //
    let addTodoSuccess = await axios.post(baseUrl, newTodo); //
    return addTodoSuccess.data; //
  } catch (e) { //
    return { error: `Todo not added on server` }; //
  } //
}
```

TodosProvider.jsx – useTodosDispatch function:

```
export function useTodosDispatch() { // 1.5
  const context = useContext(TodosDispatchContext); //
  if (context === undefined) { //
    throw new Error( //
      `useTodosDispatch must be used within a TodosProvider` //
    ); //
  } //
  return context; //
}
```

TodosProvider.jsx – editTodo function

```
const editTodo = async editedTodo => { // 2.3
  try { //
    let editTodoSuccess = //
      await axios.put(`${baseUrl}/${editedTodo._id}`); //
    return editTodoSuccess.data; //
  } catch (e) { //
    return { error: `Todo not updated` }; //
  } //
}
```

TodoProvider – component return

```
return (
  <TodosStateContext.Provider value={state}>
    <TodosDispatchContext.Provider value={dispatch}> // 1.6
      {children} //
    </TodosDispatchContext.Provider> //
  </TodosStateContext.Provider>
);
```

AddEditTodo.jsx – needed code only

```
import React, { useState, useEffect } from 'react';
import './css/AddEditTodo.css';
import TodoForm from './TodoForm';
import { useTodosState } from '../StateManagement/TodosProvider'; // 3.3

const AddEditTodo = props => {
  const [todo, setTodo] = useState({});
  const [action, setAction] = useState(`Add`); // 3.4
  const { todos } = useTodosState(); // 3.5

  useEffect(() => {
    if (todos // 3.6
      && todos.find
      && props.match && props.match.params._id
    ) {
      setAction(`Edit`)
      const todoEditing = todos.find(
        todoToCheck => todoToCheck._id === props.match.params._id //
      );
      setTodo(todoEditing); //
    } //

    return () => { //
      setTodo({}); //
      setAction(`Add`); //
    }; //
  }, [todos, props.match]); // 3.7

  return (
    <> //
      <div className="addEditTodo row"> // 3.9
        <h3>{action} Todo</h3> //
        <TodoForm //
          todo={todo} //
          action={` ${action.toLowerCase()}Todo`} //
        </> //
      </div> //
    </> //
  );
};

export default AddEditTodo;
```

TodoForm Component – needed code only:

```
import React, { useState, useEffect } from 'react';
import { Redirect } from 'react-router-dom'; // 4.2
import PropTypes from 'prop-types';
import DateCreated from './utils/DateCreated';
import generateTodoId from './utils/generateId';
import { useTodosDispatch } //
  from '../StateManagement/TodosProvider'; // 4.2
import TodoModel from './utils/Todo.model'; // 4.2
```

```

const TodoForm = props => {
  const [todoDescription, setTodoDescription] = useState('');
  const [todoDateCreated, setTodoDateCreated] = useState(null);
  const [todoCompleted, setTodoCompleted] = useState(false);
  const [editingTodo, setEditingTodo] = useState(false); // 4.3
  const [submitted, setSubmitted] = useState(false); //

  const dispatch = useTodosDispatch(); // 4.4

  useEffect(() => { // 4.5
    const { todoDescription, todoDateCreated, todoCompleted } =
      props.todo;
    if (Object.getOwnPropertyNames(props.todo).length > 0) {
      setTodoDescription(todoDescription); //
      setTodoDateCreated(todoDateCreated); //
      setTodoCompleted(todoCompleted); //
      setEditingTodo(true) //
    }

    return () => { // 4.6
      setTodoDescription(''); //
      setTodoDateCreated(new Date()); //
      setTodoCompleted(false); //
      setEditingTodo(false); //
      setSubmitted(false); //
    }
  }, [props.todo]);

  const createTodo = () => { // 4.7
    const todoId = props.action === `addTodo`
      ? generateTodoId() : props.todo._id; //

    return new TodoModel( //
      todoDescription, //
      todoDateCreated, //
      todoCompleted, //
      todoId //
    ); //
  }

  const handleSubmit = event => { // 4.8
    event.preventDefault(); //
    const todo = createTodo(); //
    dispatch({ //
      type: props.action, //
      payload: todo //
    }); //
    setSubmitted(true); //
  }; //

  const redirectAfterSubmit = () => submitted ? // 4.9
    <Redirect to="/" /> : null; //

  const disabled = !todoDescription;

  return (
    <> // 4.10
      {redirectAfterSubmit()} //
      <div className="container">
        <form onSubmit={handleSubmit}>
          <div className="form-group">
            <label htmlFor="todoDescription">
              Description:&nbsp;
            </label>
            <input
              type="text"
              name="todoDescription"
              placeholder="Todo description"
              className="form-control"
            />
          </div>
        </form>
      </div>
    </>
  );
}

```

```

        value={todoDescription || ``} // 4.11
        onChange={event =>
          setTodoDescription(event.target.value)}
      />
    </div>
    <div className="form-group">
      <label htmlFor="todoDateCreated">
        Created on:&nbsp;
      </label>
      {editingTodo ? ( // 4.12
        `${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} // 4.13
          onChange={e => setTodoCompleted(e.target.checked)}
        />
      </div>
    ) : null}
    <div className="form-group">
      <input
        type="submit"
        value="Submit"
        className={disabled ?
          `btn btn-danger` : `btn btn-primary`}
        disabled={disabled}
      />
    </div>
  </form>
</div>
</div> // 4.10
);
};

TodoForm.propTypes = {
  todo: PropTypes.oneOfType([ // 4.15
    PropTypes.exact({
      todoDescription: PropTypes.string,
      todoDateCreated: PropTypes.string,
      todoCompleted: PropTypes.bool,
      _id: PropTypes.string
    }),
    PropTypes.bool])
};

export default TodoForm;

```

App.jsx – just the TodosProvider and its children:

```

<TodosProvider>
  <Switch>
    <Route exact path="/" component={AllTodos} />
    <Route path="/add" component={AddEditTodo} /> // 5.2
  </Switch>
</TodosProvider>

```



```
      <Route path="/edit/:_id" component={AddEditTodo} />          //  
    </Switch>  
  </TodosProvider>
```

This is the end of QuickLab 22.



QuickLabs to end of Module 12 video above.

Checked to here: 09:58, 04/05/2020

QuickLabs Not Used Follow:



Quick Lab 5 - Jest

There are no activities in this QuickLab



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 once 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 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 `findAllByType` 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

4. 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.

5. 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 `findAllByType` 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.

6. 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 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`.
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 `findAllByType` 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', () => {
  const testTodo = new TodoModel();
  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();
  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 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 `` 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 `e1`;
 - Returns if the `type` property of `e1` is `span` AND `e1.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 // 6
    </span> // 7
  }; // 8
});

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 />); // 8
      const testInstance = testRenderer.root; // 9
      const dateCreated = testInstance.find( // 10
        el => el.type === `span` && // 11
        el.props.testid === `dateCreated` // 12
      ); // 13
      expect(dateCreated).toBeTruthy(); // 14
      expect(dateCreated.children). // 15
        toContain(`Date Created Component`) // 16
    }); // 17
  }); // 18
}); // 19
```

This is the end of Quick Lab 10d



QuickLab 13a – Thinking In React Part 4 – Adding State - Testing Sections That Were Removed

Activity – Make AllTodos use the todos from props

...

6. 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 **todos** and the **create** function doesn't provide this.

7. Open **tests/AllTodos.test.js** for editing.
8. In the **create** function that renders the **AllTodo** component, add a property of **todos** set to **sampleTodos**.
9. 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

...

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

...

- Run the test specs again and check that all still pass.

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




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*;
 - 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`.



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.



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);
    });
  });
});
```

This is the end of Quick Lab 13b



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.
2. Add a `suite` variable called `submitTodo`, leaving it `undefined`, to the current `TodoForm` test suite called `Form submission tests`.
3. Add a `beforeEach` to the suite that sets `submitTodo` to `jest.fn()`.
4. 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.
5. Run the tests to verify that that all still pass.

Activity 2 – Testing Form Submission

6. Add a `new test suite` to the `TodoForm test suite` called `Form submission tests`.
7. Add a `test` with a `title` of `it should call submitTodo with a prescribed parameters on submission` and an `async` 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`.
8. 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`.
9. 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

