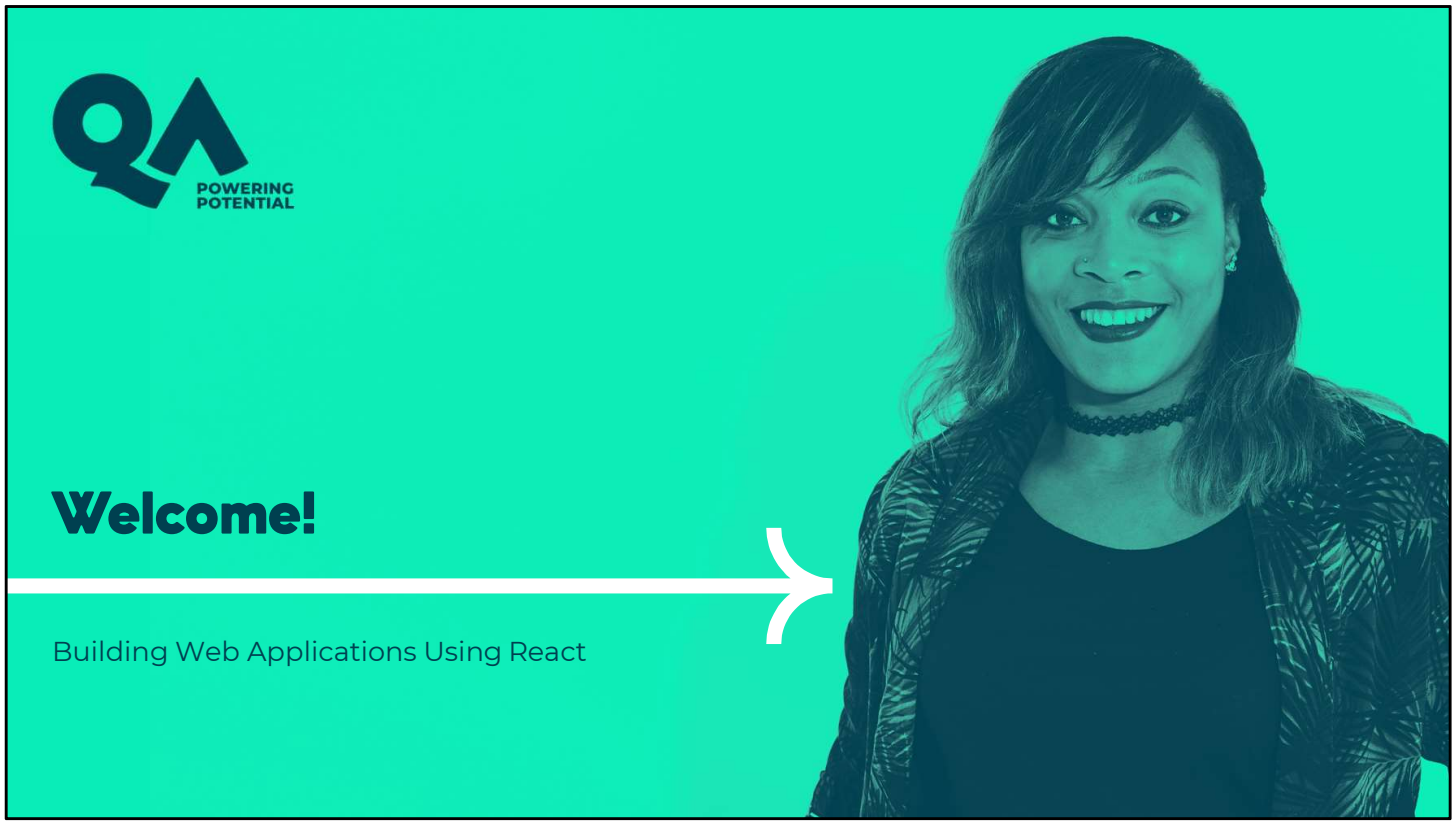

POWERING
POTENTIAL
Welcome!
Building Web Applications Using React

Overview

Objectives

- To explain the aims and objectives of the course

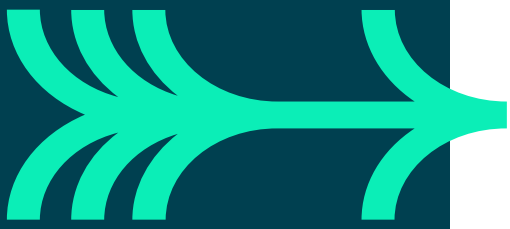
Contents

- Course administration
- Course objectives and assumptions
- Introductions
- Any questions?

Exercises

- Locate the exercises
- Locate the help files

Administration



Front door security
Name card
Chairs

Fire exits
Toilets
Coffee Room

Timing
Breaks
Lunch

Downloads and viruses
Admin. support
Messages

Taxis
Trains/Coaches
Hotels

First Aid

Telephones/Mobiles

We need to deal with practical matters right at the beginning.

Above all, please ask if you have any problems regarding the course or practical arrangements. If we know early on that something is wrong, we have the chance to fix it. If you tell us after the course, it's too late! We ask you to fill in an evaluation form at the end of the course. If you alert us to a problem for the first time on the feedback form at the end of the course, we won't have had the opportunity to put it right.

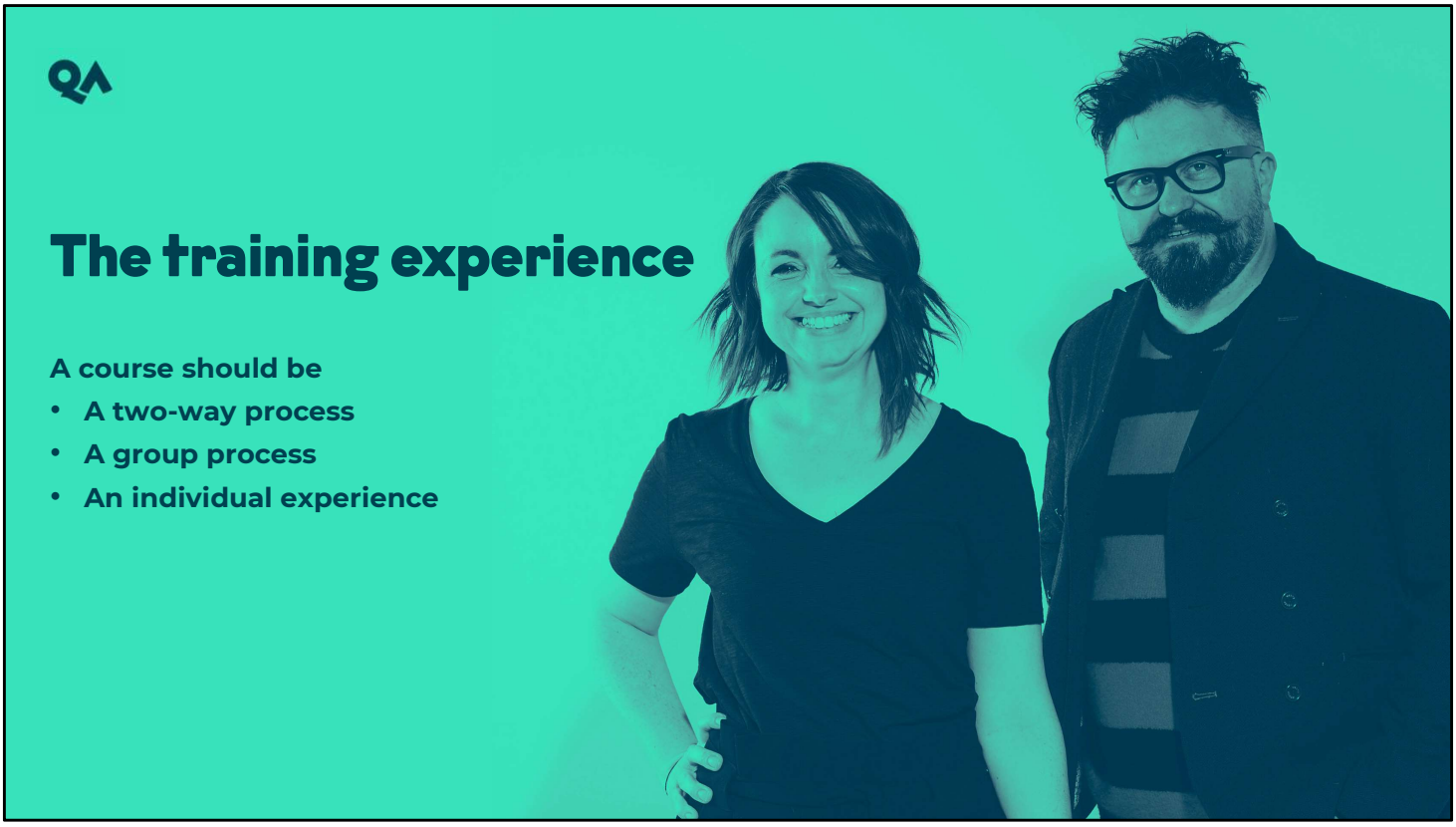
If this course is being held at your company's site, much of this will not apply or will be outside our control.

The course will be made up of lecture material coupled with the course workbook, informal questions and exercises, and structured practical sessions. Together, these different teaching techniques will help you absorb and understand the material in the most effective way.

The course notebooks contain all the overhead foils that will be shown, so you do not need to copy them. In addition, there are extra textual comments (like these) below the foils, which are there to amplify the foils and provide further information. Hopefully, these notes mean you will not need to write too much and can listen and observe during the lectures. There is, however, space to make your own annotations too.

The appendices cover material that is beyond the scope of the course, together with some help and guidelines. There are also appendices on bibliography and Internet resources to help you find more information after the course.

In the practical exercise sessions, you will be given the opportunity to experiment and consolidate what has been taught during the lecture sessions. Please tell the instructor if you are having difficulty in these sessions. It is sometimes difficult to see that someone is struggling, so please be direct.



The best courses are not those where the instructor spends all of his/her time pontificating at the front of the class. Things get more interesting if there is dialogue, so please feel free to make comments or ask questions. At the same time, the instructor has to think of the whole group, so if you have many queries, you might be asked to deal with them off-line.

Work with other people during practical exercise sessions. The person next to you may have the answer, or you may know the remedy for them. Obviously do not simply 'copy from' or 'jump-in on' your neighbour, but group collaboration can help with the enjoyment of a course.

We are also individuals. We work at different paces and may have special interests in particular topics. The aim of the course is to provide a broad picture for all. Do not be dismayed if you do not appear to complete exercises as fast as the next person. The practical exercises are there to give plenty of practical opportunities; they do not have to be finished and you may even choose to focus for a long period on the topic that most interests you. Indeed, there will be parts labelled 'if time allows' that you may wish to save until later to give yourself time to read and absorb the course notes. If you have finished early, there is a great deal to investigate. Such “’hacking’ time is valuable. You may not get the opportunity to do it back in the office!

Course outcomes

By the end of the course, you will be able to:

- Create React components
- Perform some simple tests
- Think in React
- Add state and props to an application
- Add inverse data flow to an application
- Use some common React hooks
- Use external services to provide data
- Set up a single page application
- Use context and reducers

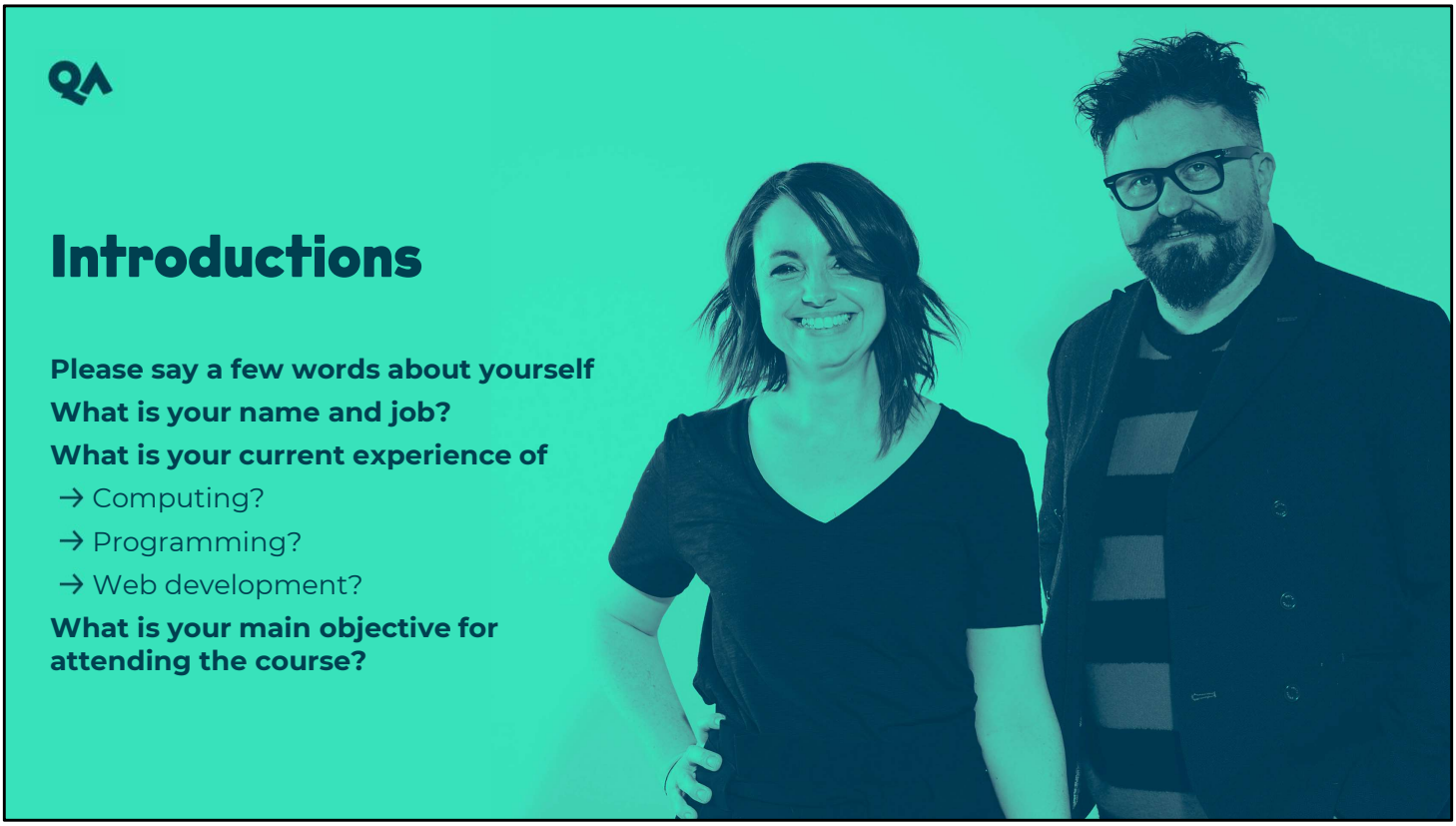
Assumptions

This course assumes the following prerequisites

- HTML and CSS skills equivalent to those provided by the Web Development Fundamentals – HTML and CSS course
- JavaScript skills equivalent to those provided by the Web Development Fundamentals – JavaScript course

If there are any issues, please tell your instructor now

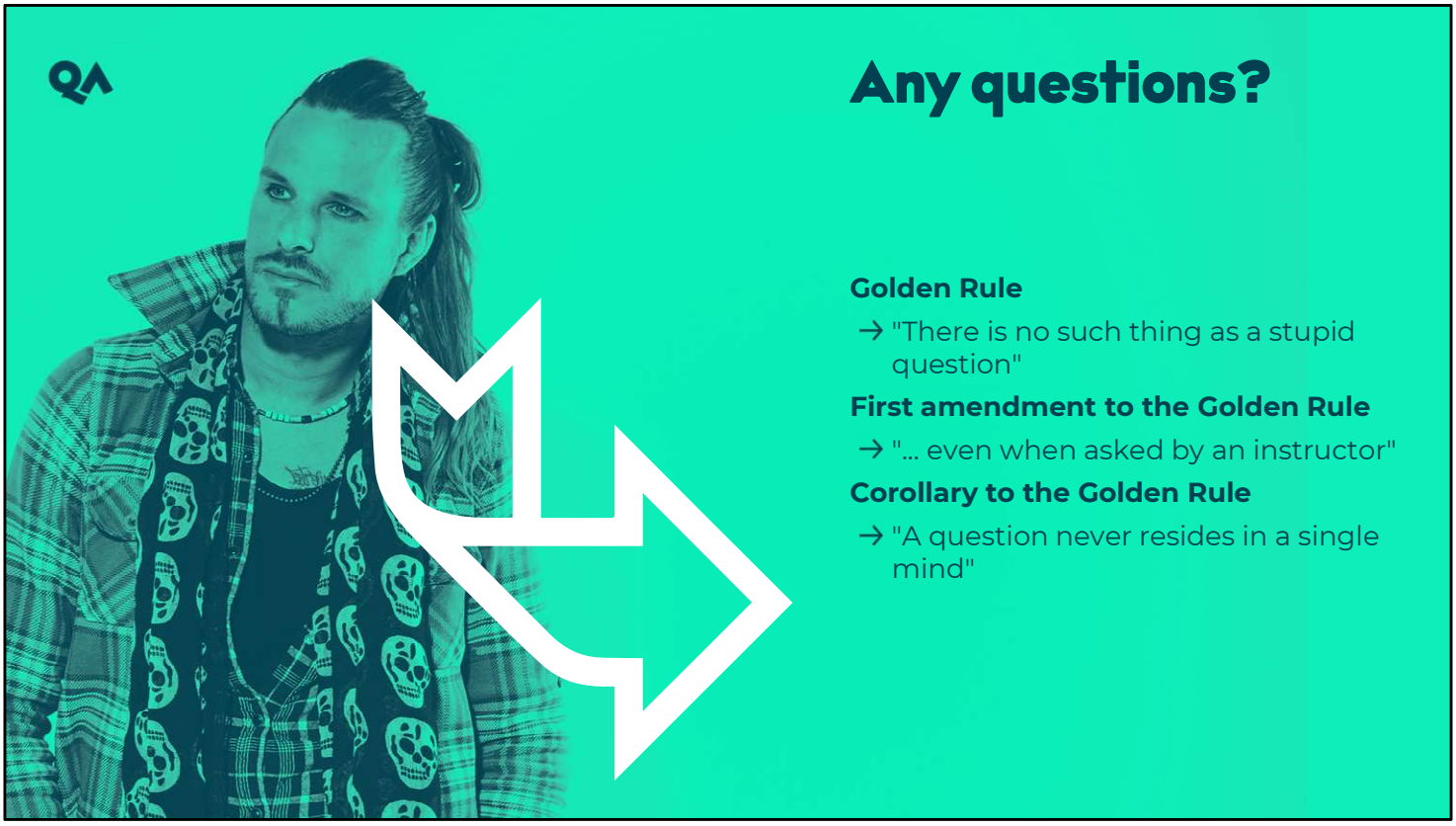
If you are not sure of any of these, please inform the instructor as soon as you can and they will do their best to help you.



One of the great benefits of courses is meeting other people. They may have similar interests, have encountered similar problems and may even have found the solution to yours. The contacts made on the course can be very useful.

It is useful for us all to be aware of levels of experience. It will help the instructor judge the level of depth to go into and the analogies to make to help you understand a topic. People in the group may have specialised experience that will be helpful to others.

It is worth highlighting particular interests, as we may be able to address them during the course. However, it is a general course that aims to cover a broad range of topics, so the instructor may have to deal with some areas during a coffee break or over lunch.



Please feel free to ask questions.

Teaching is a much more enjoyable and productive process if it is interactive. You will no doubt think of questions during the course. If so, ask them!

CLIENT REQUEST FOR AN APPLICATION

As the **user**,

I want to be able to *access an* ***online application*** *that allows me to* ***view***, ***add*** *and* ***edit*** *a 'To Do' list*

so that *I can keep myself organised*

DISCUSSION:

How should the 'team' go about creating a solution for this request?

What architecture structure would it take?

Classic N-TIER Architecture

N-Tier Architecture is the idea that the application is split into distinct areas of code running on different computers.

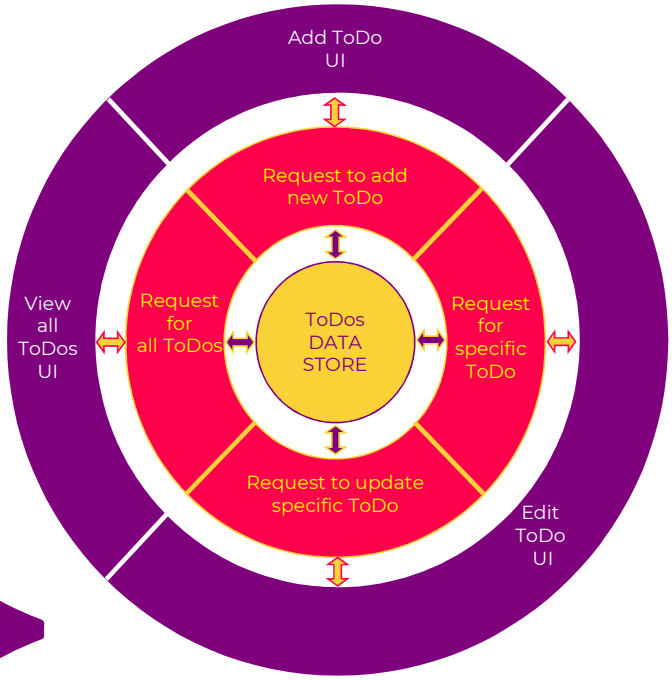
Makes applications more maintainable and upgradable.

Makes code more reusable.

Presentational or UI Layer

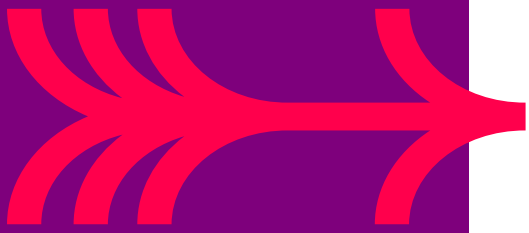
Business Logic or Application Layer

Persistence or Data Layer



11

EPIC BREAKDOWN



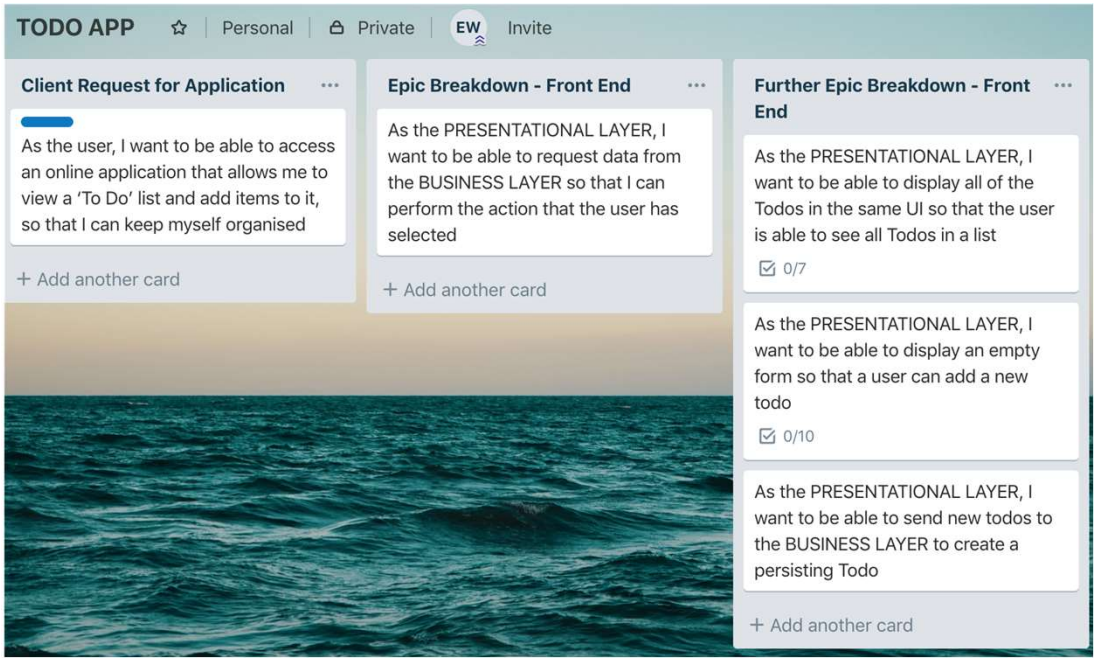
As the **PRESENTATIONAL LAYER**,
I want to be able to ***request data from*** or ***send data to*** the **BUSINESS LAYER**
so that *I can perform the action that the user has selected*

DISCUSSION:

What needs to be done to produce a solution for this story?

Is it still a bit of an epic?

Epic breakdowns



Further refinement may yield 3 user stories.

They will have some common dependencies such as requiring common headers and footers for each view.

13

Front end development options

	'Vanilla' Development	JavaScript Libraries	JavaScript Frameworks
TECHNIQUE:	Use HTML, CSS and JavaScript to create the user interface 	Use a JavaScript library to build and create the user interface E.g. ReactJS 	Use a JavaScript framework to create the user interface E.g. Angular 
PROS:	• No specialist knowledge of libraries or frameworks needed	• Fast user interface • Reusable code • Can create single page applications • Only need to use parts of library needed in project	• Built-in features and functionality • Reusable code • Can create single page applications
CONS:	• Duplication of code to create multiple pages • Higher maintenance costs	• Requires specialist knowledge to use effectively • Unable to perform 'business logic' actions within the application	• Requires specialist knowledge to use effectively • Restrictive rules of the framework • Can create large overheads for download as whole framework is included

14

Acceptance criteria

All of these user stories have shared dependencies

- Therefore some shared criteria for acceptance:

Acceptance Criteria Delete

0%

- [ ] The UI should be created using ReactJS and JSX
- [ ] Every UI should have a 'header' section
- [ ] The 'header' section should contain the QA Logo that is linked to https://www.qa.com
- [ ] The 'header' section should contain the title 'QA Todo App'
- [ ] Every UI should have a 'footer' section
- [ ] The 'footer' section should contain '© QA Ltd 2019-'

Further Epic Breakdown - Front End

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

0/7

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

0/10

As the PRESENTATIONAL LAYER, I want to be able to send new todos to the BUSINESS LAYER to create a persisting Todo

15


Introduction
Building Web Applications Using React

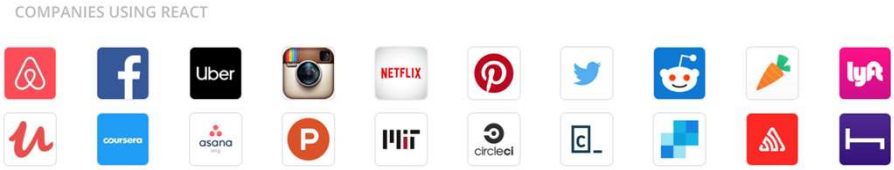
Objectives

- To become aware of what React is
- To be aware of developer tools available for React
- To be able to set up the developer environment and a skeleton React application
- To be aware of the security concerns with React

ReactJS – another JS Framework, right?

WRONG!

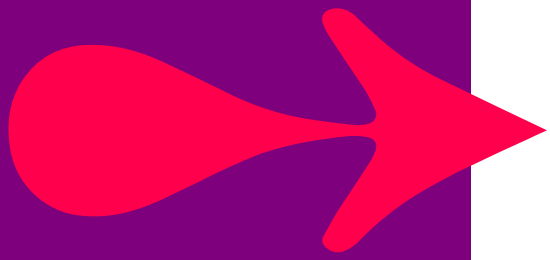
- ReactJS is not a framework, it's a UI Library
- Developed at Facebook
- Purpose is to facilitate the creation of interactive, stateful and reusable UI components
- Facebook have used React for Instagram and in Facebook itself!
- Instagram's UI is completely made with React
- Facebook's commenting UI is React-based
- Parts of Netflix use React
- Often used as the "V" part of MVC – no assumptions made about rest of stack
- React is there to take data and display it!



What does React do?

- Performs work on client side
- Can be rendered on server side
- Both client and server side can work together
- Uses a concept called Virtual DOM
- It performs the least amount of DOM manipulation possible for a fast, fluid interface, yet still up to date

A REALLY SIMPLE REACTJS APPLICATION EXAMPLE



The smallest possible ReactJS example would be:

```
import React from `react`;
import ReactDOM from `react-dom`;

ReactDOM.render(
  <h1>Hello World</h1>,
  document.querySelector(`#root`)
);
```

Essentially, this places the ReactJS component (the first argument to **`ReactDOM.render`**) in the element with an **`id`** of **`root`** in **index.html**

Virtual DOMs

A copy of the actual DOM that React uses to decide what needs to updated.

If a component is changed as a result of code being executed, the page has to update.

Rather than update the whole page, React does two things:

- First, it runs a diffing algorithm, which finds out what properties have changed in that object in the Virtual DOM
- Second, reconciliation happens, where it updates the parts of the DOM identified by the diffing algorithm

This is really good for multiple reasons:

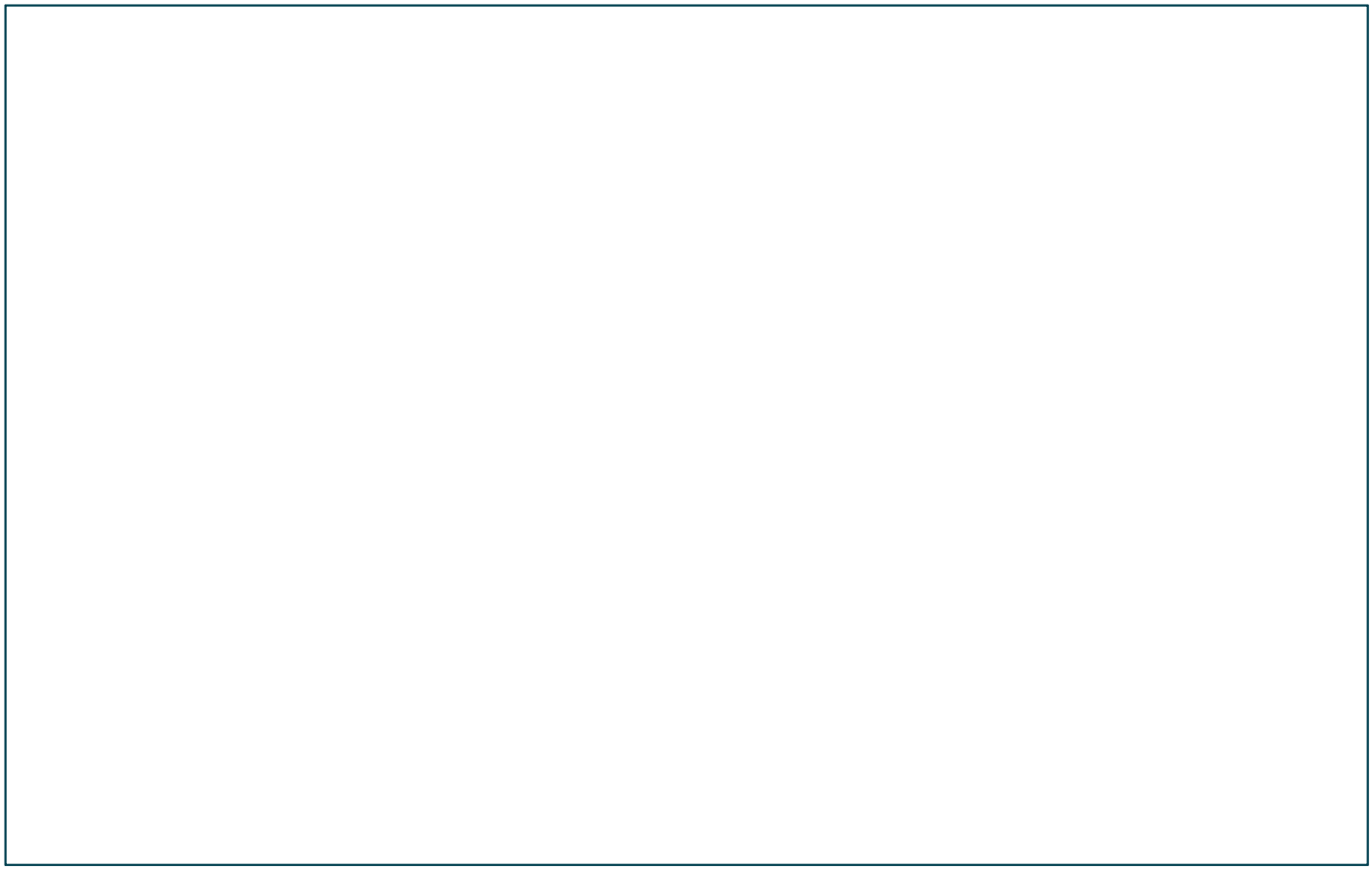
- Memory: since it's only changing what it needs to change, there's less overhead overall.
- Fluidity: the entire object (or page!) doesn't need to refresh whenever anything changes, only what needs to change.
- Server-side DOMs can be rendered to enable server-side React views
- Even less overheads!

The developer environment

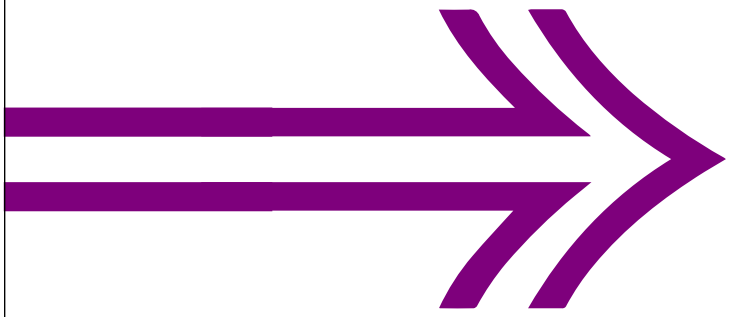
As with all modern web development it relies heavily on NodeJS and npm.

Application tooling can be:

- ***Zero-config*** – use the **create-react-app npx** package runner to produce a skeleton ReactJS application and install all of the required tooling (including testing)
- ***Manual*** – use **Webpack** and **Babel** to transpile and bundle JavaScript and JSX modules



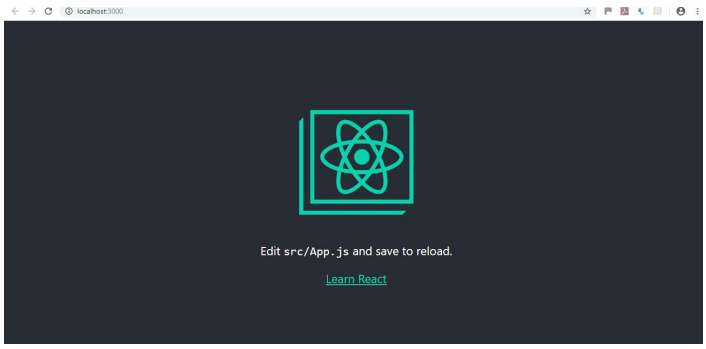
QuickLabs Environment Set Up – Install NodeJS (if not already done)



In this activity, you will:

- Visit http://nodejs.org/en
- Download and install the latest Long Term Support version
- Verify the installation

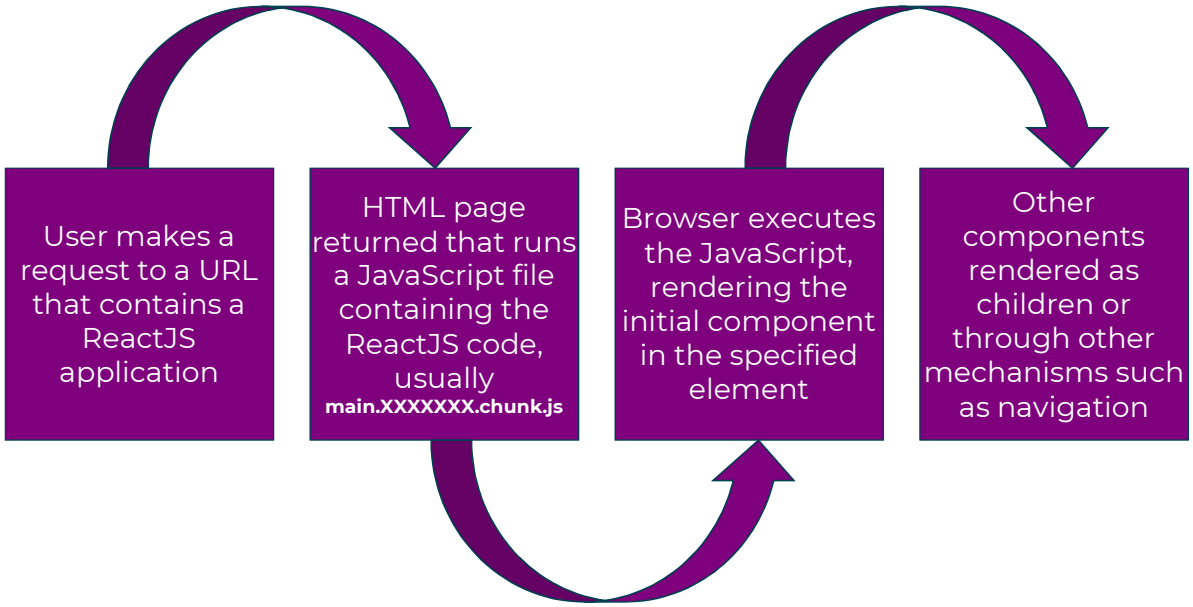
QuickLab 1 – Get a ReactJS App Up and Running



In this QuickLab, you will:

- Use a pre-built package to scaffold a ReactJS application
- Run the application and view it in the browser

How a ReactJS app gets into the browser



25

ReactJS and security

The most obvious of the OWASP Top 10 for ReactJS to be susceptible to is:
A9:2017-Using Components with Known Vulnerabilities

- ReactJS uses a lot of other packages

npm keeps track of packages with know vulnerabilities – use `npm audit` to check your application.

If the solution does not involve breaking changes, `npm audit fix` can be used to update packages to later versions.

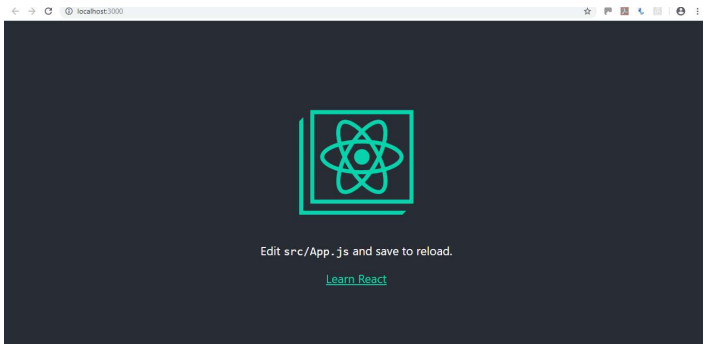
If there are breaking changes, then a bigger rebuild of the application may be necessary.

No vulnerabilities are reported in the core code for **A7:2017- Cross-Site Scripting (XSS)**.

Developers should always be aware of the dangers of incoming data (**A3:2017- Sensitive Data Exposure**) and whether it should be trusted (**A2:2017-Broken Authentication**).

26

QuickLab 2 – Build and Run the Application

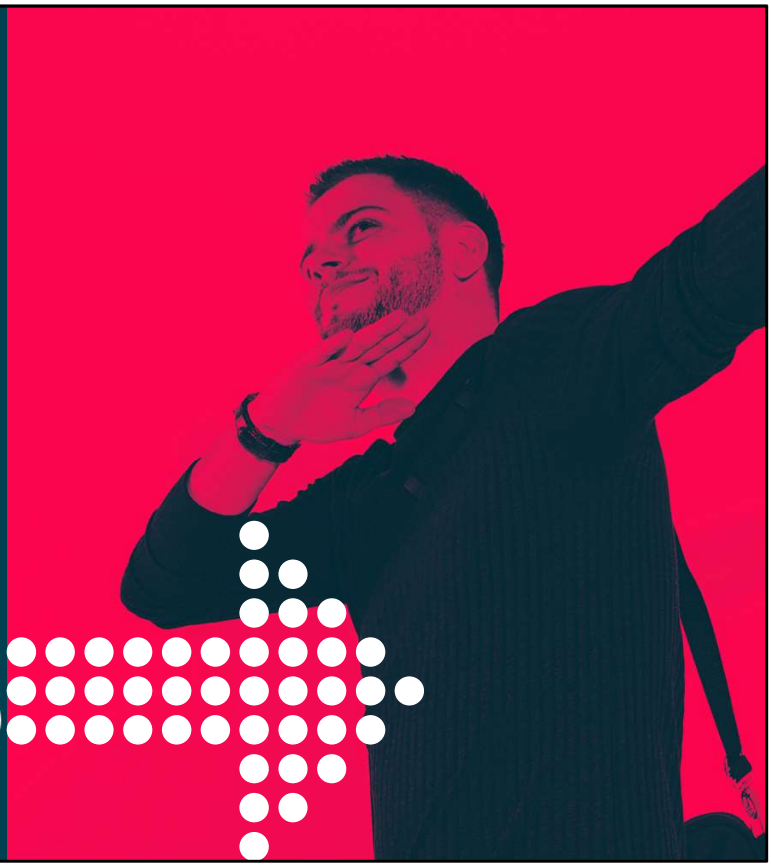


In this QuickLab, you will:

- Examine the build tools supplied with create-react-app
- Examine the built files
- Serve a built application and examine

Objectives

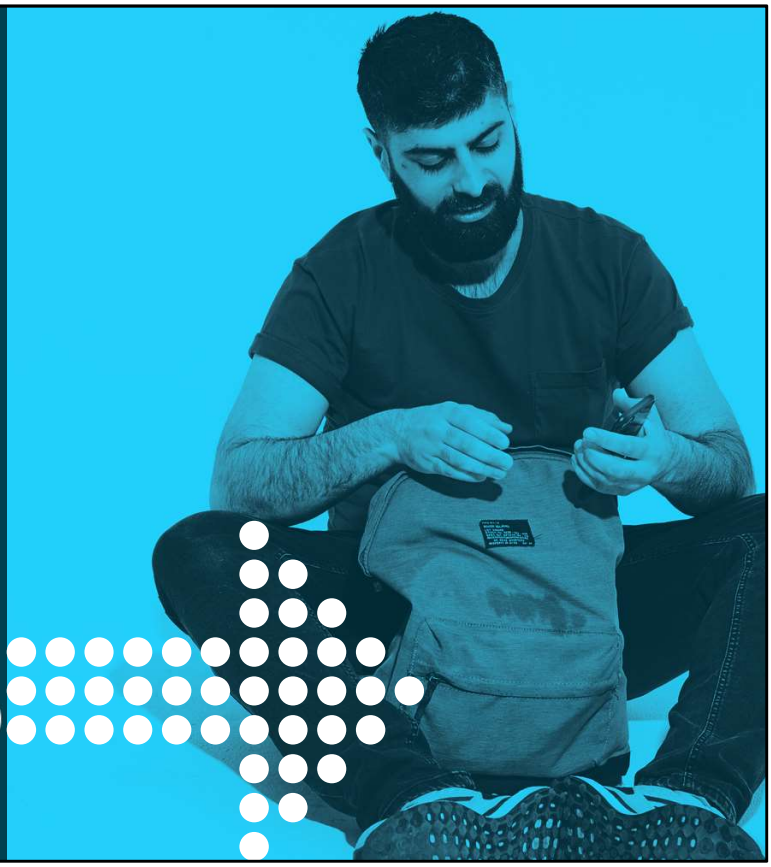
- To become aware of what React is
- To be aware of developer tools available for React
- To be able to set up the developer environment and a skeleton React application
- To be aware of the security concerns with React



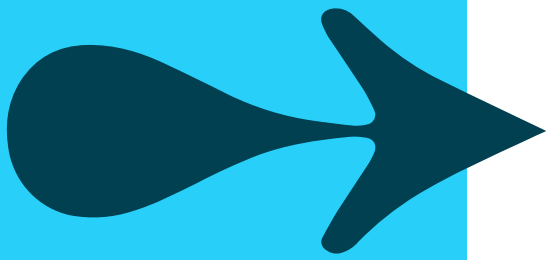

Components and JSX
Building Web Applications Using React

Objectives

- To understand what components are
- To know what JSX is and why it is used in React JS
- To be able to create functional and class components
- To be able to add multiple components
- To know how to inspect components in the browser
- To understand how the in-browser tools work



REACT COMPONENTS



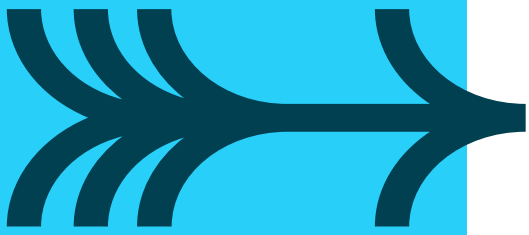
React is fundamentally about components

- Developers need to think in components when working with ReactJS!

According to the official documentation:

"React is a library for building composable user interfaces. It encourages the creation of reusable UI components which present data that changes over time."

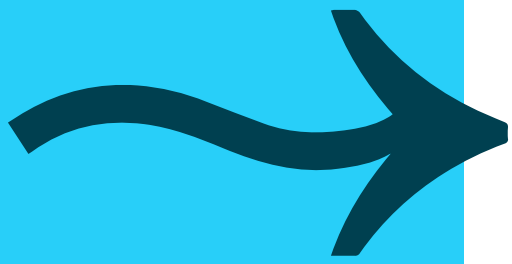
REACT COMPONENTS



Components:

- Should be created as a function
  - Traditionally applications were a mixture of function and class components
  - Since React v16, all components can be functions
  - Class components will NEVER be deprecated
- Should be written using JSX
  - Can be written in raw JavaScript – readability suffers
  - JSX helps integrate HTML and JavaScript
  - Written specifically for ReactJS

JSX – THE LANGUAGE FOR REACT



Short for JavaScript Syntax Extension.

Provides syntactic sugar for component code.

- Helps clean up HTML and JavaScript
- Makes it easier and quicker for developers
- Faster optimisation occurs when compiling to JavaScript
- Helps prevents XSS attacks

Essentially, it converts the HTML in the return of a component into a React element which in turn creates the actual element to be added to the DOM.

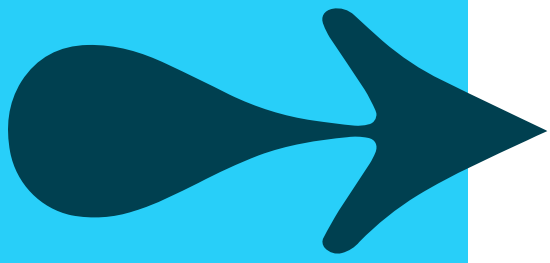
```
<div className="header">
  <h1>My React App</h1>
<div>
```

Becomes:

```
React.createElement("div", {className:
"header"}, React.createElement("h1", null,
"My React App"));
```

From https://reactjs.org/docs/introducing-jsx.html: By default, React DOM escapes any values embedded in JSX before rendering them. Thus it ensures that you can never inject anything that's not explicitly written in your application. Everything is converted to a string before being rendered. This helps prevent XSS attacks.

JSX



JSX can be used as an expression.

- After compilation they become regular JS function calls that evaluate to JS objects
- Can assign variables to it
- JS Expressions can be embedded into JSX using curly braces:

```
import React from `react`;

const Title = () => {
  const title = `My React App`;
  const el = <h1>Welcome to {title}</h1>;

  return (
    <>
      {el}
    </>
  );
}
export default Title;
```

This is shorthand for `<React.Fragment>`

Would render: **Welcome to My React App**

JSX is an expression too

JSX expressions get compiled to regular JavaScript function calls that evaluate to JavaScript objects.

This means that JSX can also be...

...used as expressions inside **if** statements and **for** loops

...accepted as an argument in expressions and function calls

...returned from functions

```
function makeWelcomeMessage(user) {
  if (user) {
    return <h1>Hello, {getDisplayName(user)}!</h1>;
  }
  return <h1>Hello, Stranger.</h1>;
}
```

35

Components as Functions in JSX

Where possible, components should be functions.

Functions must return:

- a representation of a single native DOM component (e.g. **`<div />`**) **OR**
- a custom composite component defined by the developer **OR**
- null or false to indicate that the component should not be rendered

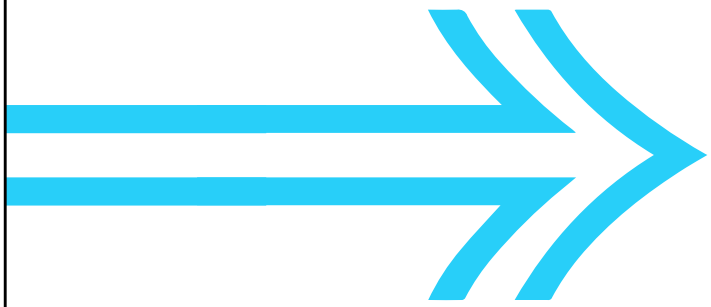
This return should be **PURE**, it:

- Does not modify component's state
- Returns same result each time it is invoked
- Does not directly interact with the browser

```
// Component with single native DOM
// element

import React from `react`;
const MyComponent = props => {
       return (
              <h1>Hello World</h1>
       );
};

export default MyComponent;
```

36

QuickLab 3 – Create function components



In this QuickLab, you will:

- Add function components to your application
- Render the new components as a children of other components

Components can be classes

Components can be created as a class.

- The release of React version 16.8 introduced functionality to the library that means they are no longer necessary in most cases
- Classes will always be supported in React though!

Class components must have a render method that must return:

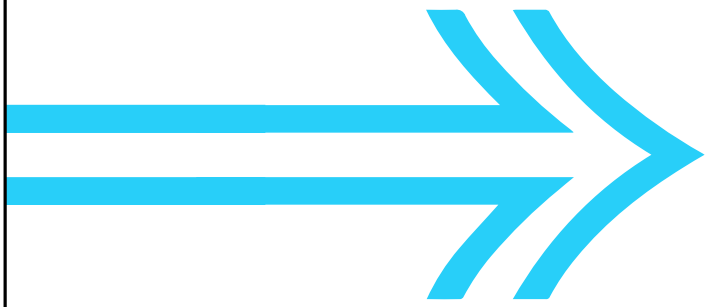
- a representation of a single native DOM component (e.g. **`<div />`**) **OR**
- a custom composite component defined by the developer **OR**
- null or false to indicate that the component should not be rendered

This return should be **PURE.**

```
// Component as a class with single
// native DOM element
class MyClassComponent extends Component
{
  render() {
    return (
      <div>
        <h2>I am a class component</h2>
        <p>I work in almost the same way
        as Function components</p>
      </div>
    );
  }
};
export default MyClassComponent;
```

38

QuickLab 4 – Create a Class Component



In this QuickLab, you will:

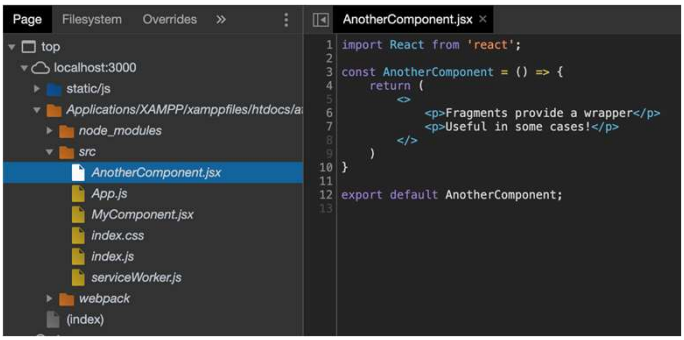
- Add a class component to your application
- Render the new component as a child of another component


React Developer Tools

DEBUGGING IN THE BROWSER

Because a development server is used, access is granted (via source-maps) to the component code.

They can be found on the Sources **tab**, accessing the *project folder* on **localhost:3000** and then the **src** folder:

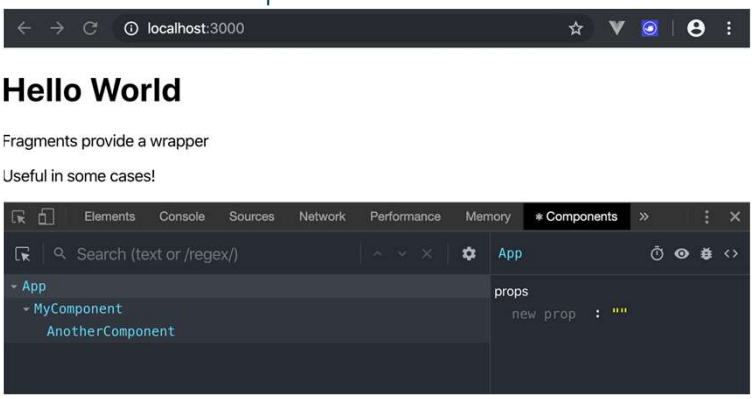


Breakpoints, watches and all the usual JS debugging features can be used here.

REACT DEVELOPER TOOLS

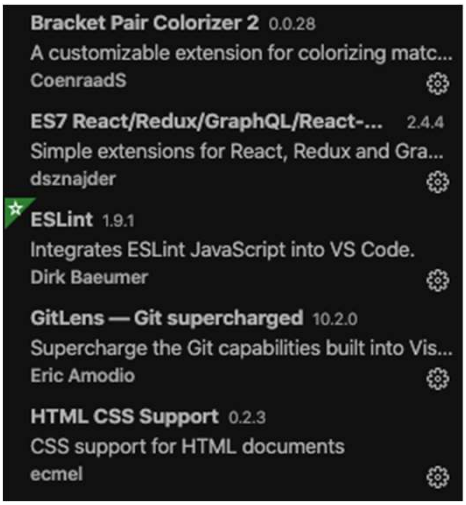
Facebook provide an extension for Chrome and Firefox.

- Components view allows overview of rendered components
  - The component tree is displayed
  - Selecting individual component reveals more information
  - Additional tools will display the matching DOM element, log the component to the console & show the component's source code



The DevTools also have a Profiler which allows measurements to be taken whilst the application is rendering – such as render time for components and other measurements needed for performance optimisation.

USEFUL VSCODE EXTENSIONS



These extensions for VSCode can help in making development more efficient and robust.

The ES7 React extension provides Emmet style shortcuts for many ReactJS constructs:

`rafce [Tab]` – gives boilerplate code for a Functional component.

There is a list of the shortcuts and the boilerplate they produce in the extension's documentation.

Bracket Pair Colorizer 2 – This extension makes matching bracket pairs different colours for easy visual identification

ESLint – This will lint your JavaScript code and suggest how it can be made 'better' in terms of format, line termination, etc – rules can be edited through the extension's config

GitLens – Provides information within the editor window for files that have commits against them, revealing who, when and giving access to what

HTML CSS Support – Tracks class names and ids within your CSS files in the project and suggests these when writing HTML

Objectives

- To understand what components are
- To know what JSX is and why it is used in React JS
- To be able to create functional and class components
- To be able to add multiple components
- To know how to inspect components in the browser
- To understand how the in-browser tools work


Introduction to Testing
Building Web Applications Using React

Objectives

- To understand how tests are set up and run in a create-react-app React application
- To have an overview of the Jest testing library
- To be able to run and access code-coverage reports for tests

Testing

Integral part of software development.

create-react-app projects come ready made to test:

- Jest – “Jest is a delightful JavaScript Testing Framework with a focus on simplicity. It works with projects using: Babel, TypeScript, Node, React, Angular, Vue and more!”

The scripts supplied mean that `npm test` can be used to run the tests.

Tests are run from files that contain **.test.** in their name.

- These need not be in the same folder as the component files – Jest is set up to look for test files
- Tests are also run from any **.js** file within a folder called **__tests__**

47

Jest overview

Open Source JavaScript framework capable of testing ***any kind*** of JavaScript application.
Follows Behaviour-Driven Development to ensure that each line of JS is properly unit tested.
Provides small syntax to test smallest unit of entire application, instead of testing it as a whole.
Features:

- Doesn't depend on any other JavaScript Framework
- Doesn’t require any DOM
- Clean and obvious syntax

48

Jest test building blocks

SUITES

Basic building block of Jest Framework.

Collections of similar type test cases written for a specific file or function.

Can be nested.

Usually contain a **`describe`** function and at least 1 call to an **`it`** or **`test`** function.

- The describe function is not necessarily required though

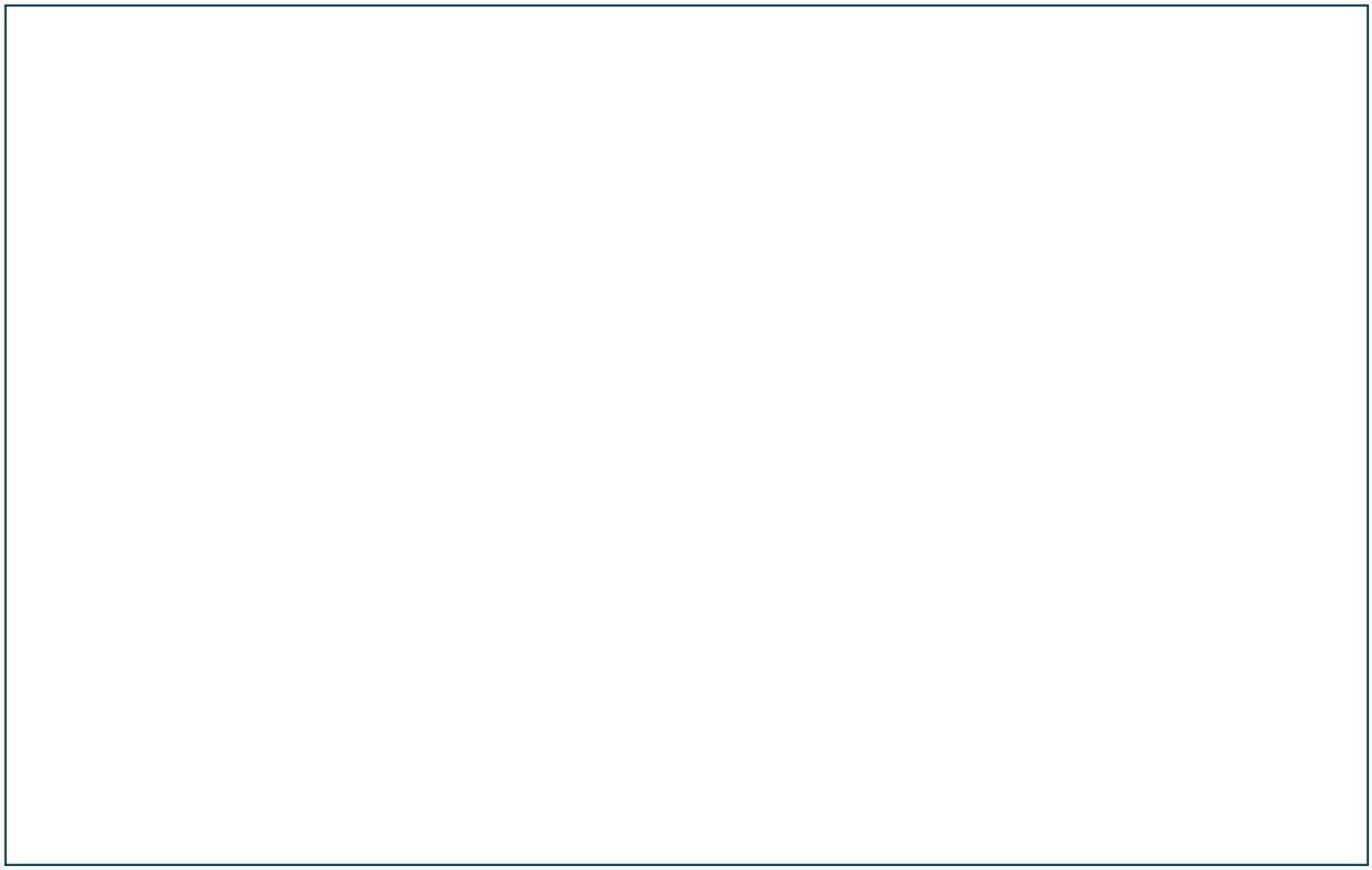
`describe()`

Call to this function requires a string, used to identify the suite and a function that contains calls to the **`it`** or **`test`** functions.

`it()` or `test()`

Calls to these functions require a string, used to identify the test case it represents and a function that defines how the test will be evaluated – through calls to the **`expect`** function.

49



Jest Test Building Blocks

`expect()`

Defines the ***actual value*** obtained by running the test case.

- `it()` and `test()` calls can have more than 1 `expect()` call.

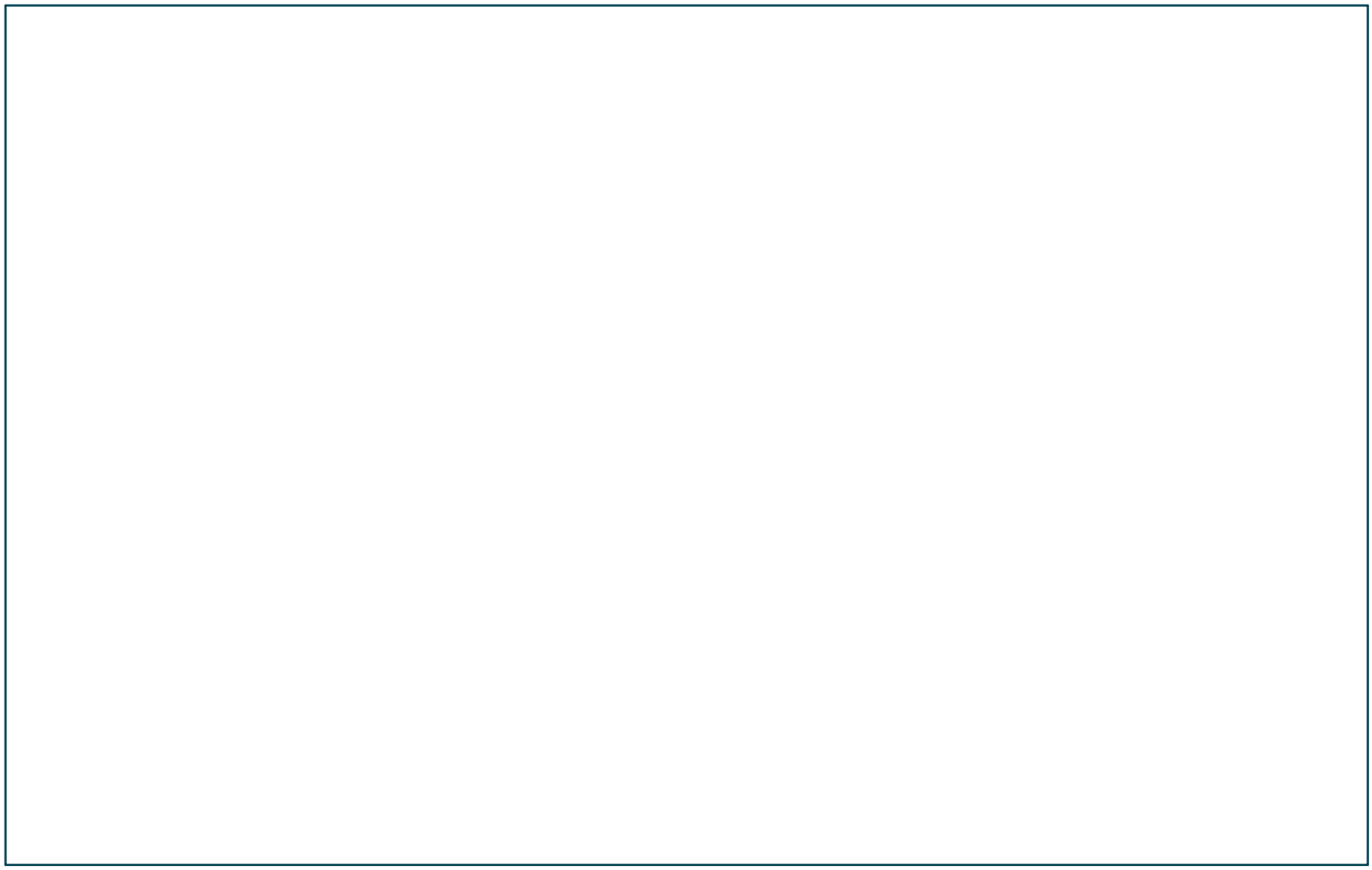
Uses **matchers** to define how to compare the expected result with the actual.

MATCHERS

Jest provides a host of in-built matchers.

Each does a Boolean comparison of the actual and expected outputs.

50



Jest MATCHERS

Matcher	Description
`.anything()`	Match anything except null or undefined
`.toBe(expected), .toEqual(expected)`	Expect the actual value to be === to the expected, for toEqual, objects and their properties can be compared
`.toBeFalsy(), .toBeNull(), .toBeUndefined()`	Expect the actual value to be Falsy, explicitly null or explicitly undefined
`.toContain(expected)`	Expect the actual value to contain the expected value – can be in an array as well as a substring in a string
`.toBeGreaterThan(expected), .toBeGreaterThanOrEqual(exp)`	Expect the actual value to be greater than (or equal to) the expected value
`.toBeLessThan(expected), .toBeLessThanOrEqual(exp)`	Expect the actual value to be less than (or equal to) the expected value
`.not.`	Invert the result of the matcher (placed after expect and before matcher)

51

Example Spec

```
describe(`A suite is just a function`, () => {
    let a;
    let b;
    it(`and so is a spec`, () => {
      a = true;
      b = `Another string`;
      expect(a).toBe(true);
      expect(a).not.toBeFalsy();
      expect(b).toContain(`her`);
      expect(b).not.toMatch(/foobar$/);
    });
});
```

For this test to report as **PASSED**, all `expect` calls must return true.

NOTE: This is not an ideal way to test these situations!

52

Custom Matchers

Sometimes the built-in matchers will not fit with the test required.

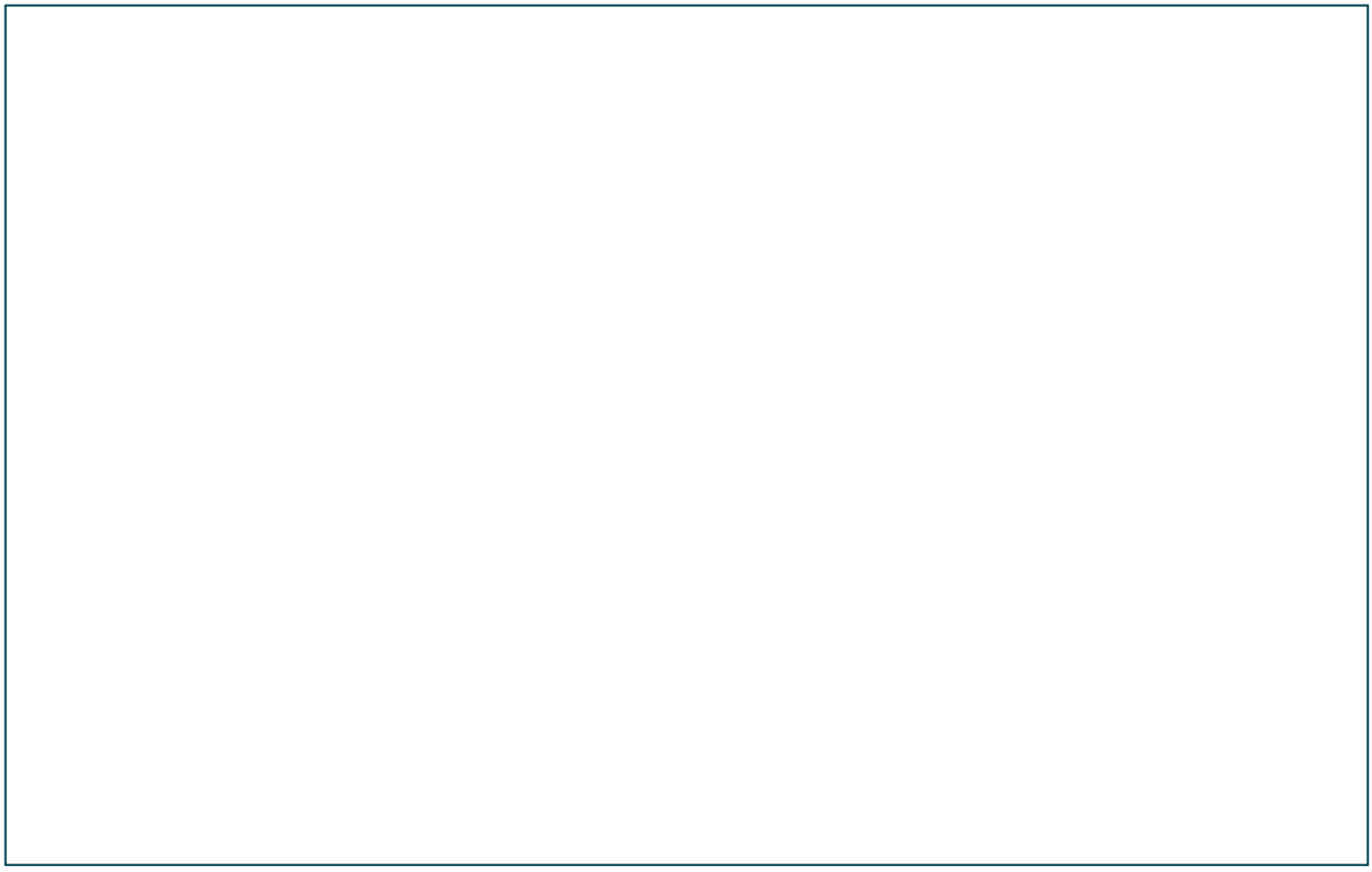
Jest allows custom matchers to be written.

- As long as they follow the correct pattern

Jest's documentation explains how to create these:

https://jestjs.io/docs/en/expect#custom-matchers-api

53



Jest Spy

Allows spying on the test code's function calls.

Intercepts call to actual function and allows logging and monitoring of calls to it.

For existing functions, the `spyOn()` method needs to receive the object and the name of the function.

Jest provides a number of matchers to verify that the function has been used appropriately:

Matcher	Description
`toHaveBeenCalled()`	Expect the function spied on to have been called at least once
`toHaveBeenCalledTimes(exp)`	Expect the function spied on to have been called an expected number of times
`toHaveBeenCalledWith(obj)`	Expect the function spied on to have been called with these particular arguments at least once
`toHaveBeenLastCalledWith (arg1, arg2, …)`	Expect the last call of the function to have had the arguments supplied
`toHaveReturned toHaveReturnedTimes(number) toHaveReturnedWith(value)`	Expect the function to have returned, returned a number of times or with a specific value

54

These are just a selection of matchers provided by Jest for use with functions.

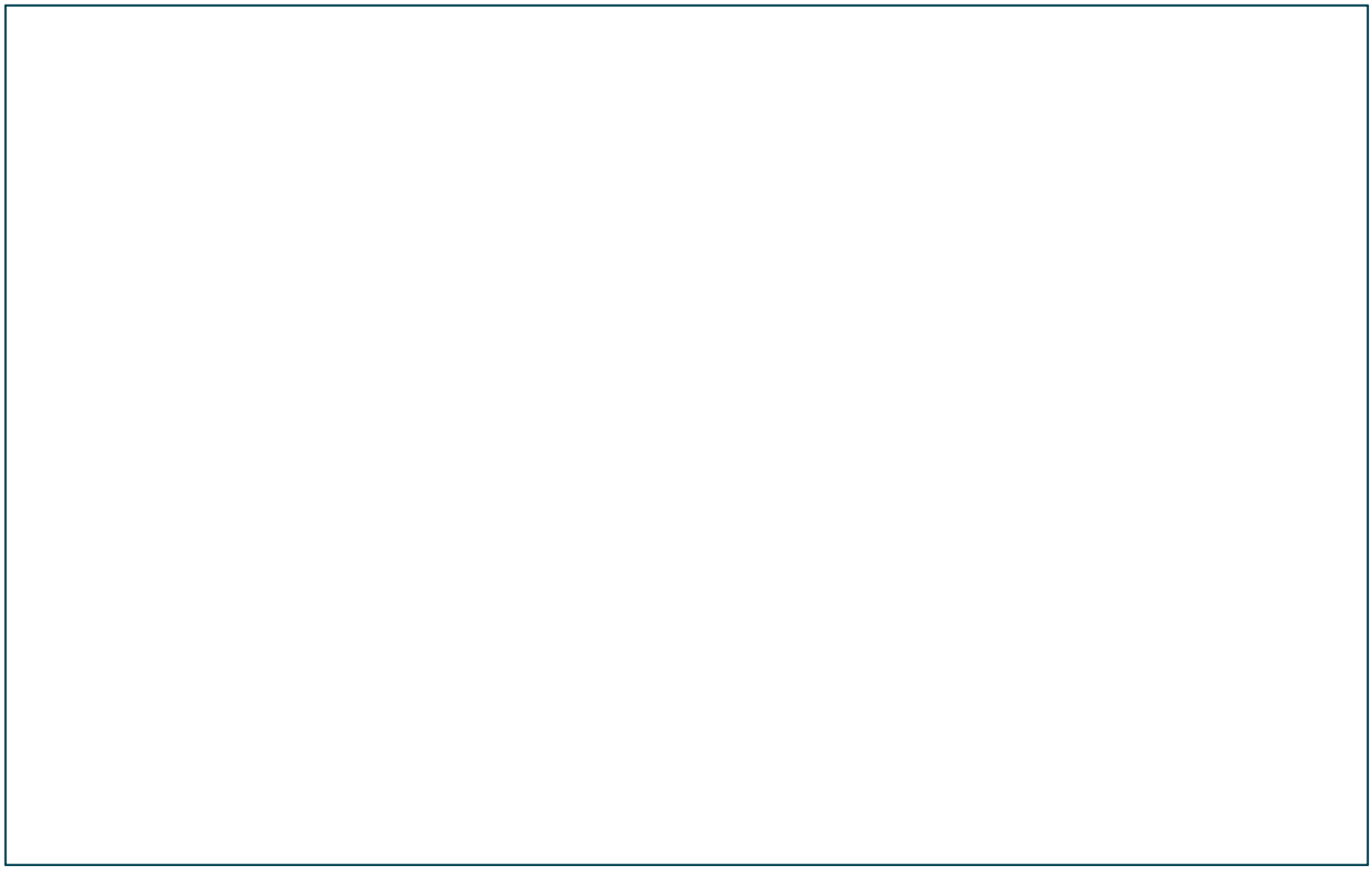
Jest fn() and spyOn

`jest.fn(implementation)` returns a new, unused mock function with an optional implementation.

`jest.spyOn(obj, methodName, accessType?)` similar to **`jest.fn()`** but also tracks calls to **`obj[methodName]`**. Returns a Jest mock function.

- By default it calls the spied method – this is different to other testing libraries

55



Set up and tear down

Code can be executed before and after each spec or suite is run.

Jest provides the following functions to facilitate this.

- Each receives can receive a function to execute (and a timeout for asynchronous use):

BEFORE		AFTER	
`beforeEach()`	`beforeAll()`	`afterEach()`	`afterAll()`
Runs before each `it()` or `test()` call executes when present at the start of a `describe()` block	Runs once, before any of the `it()` or `test()` calls are executed *Seen as dangerous as can result in leaking state between specs producing erroneous passes/fails*	Runs straight after each of `it()` or `test()` call executes when present	Runs after all of the `it()` or `test()` calls have executed *Seen as dangerous as can result in leaking state between specs producing erroneous passes/fails*

56

Code Coverage

Unit tests ran from the CLI can create code coverage reports.

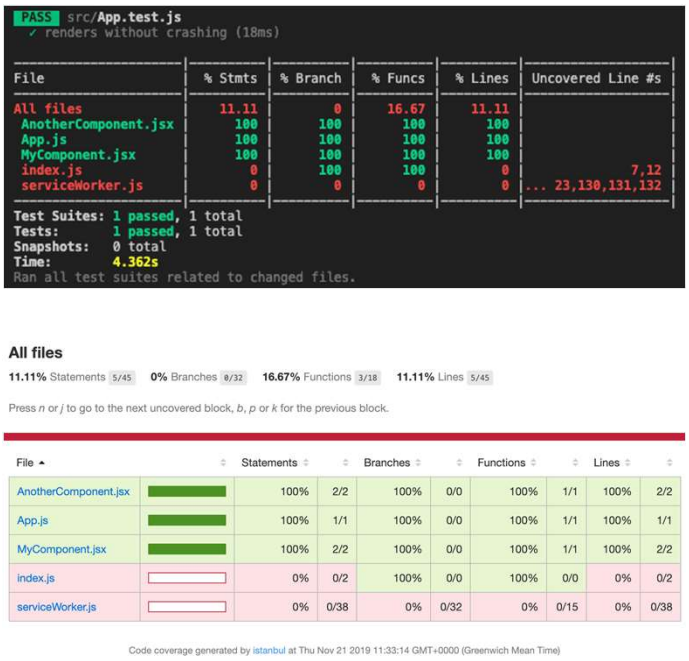
- Reports show parts of code base not properly tested by unit tests

Use command:

```
npm test -- --coverage
```

- Results stored in /coverage folder

→ HTML versions can be access in the lcov-report folder – index.html gives access to the overview



CONTINUED 57

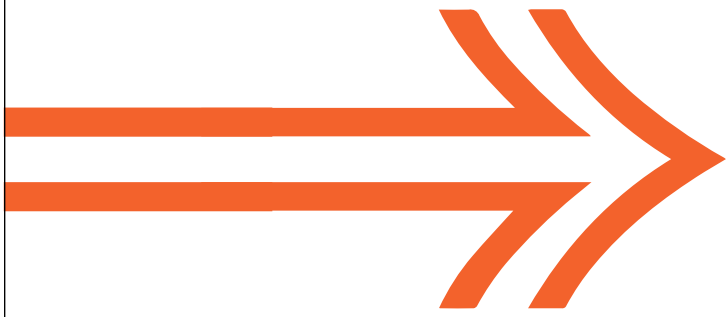
TESTING REACT APPLICATIONS

Methods for testing ReactJS components can be divided into 2 categories:

- By rendering a component tree in a test environment and asserting on their output
- Running end-to-end (e2e) tests in a realistic browser environment
  - e2e not really concerned with React components

ReactJS.org recommend using **Jest**, **react-test-renderer** and **react-testing-library** for testing components.

QuickLab 5 – Jest



There is no QuickLab for this section.

We will return to testing as we progress through the course.

Objectives

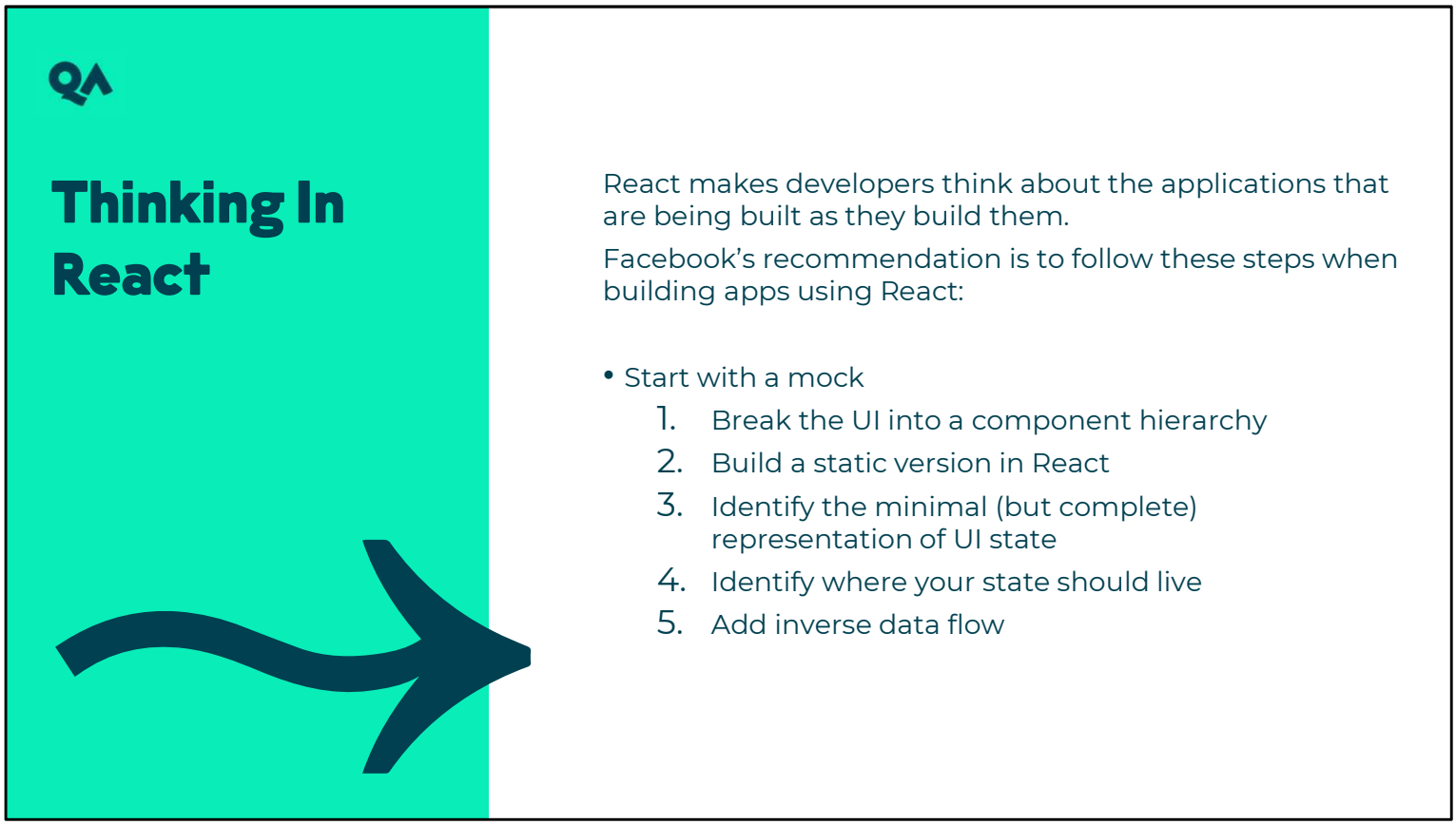
- To understand how tests are set up and run in a create-react-app React application
- To have an overview of the Jest testing library
- To be able to run and access code-coverage reports for tests

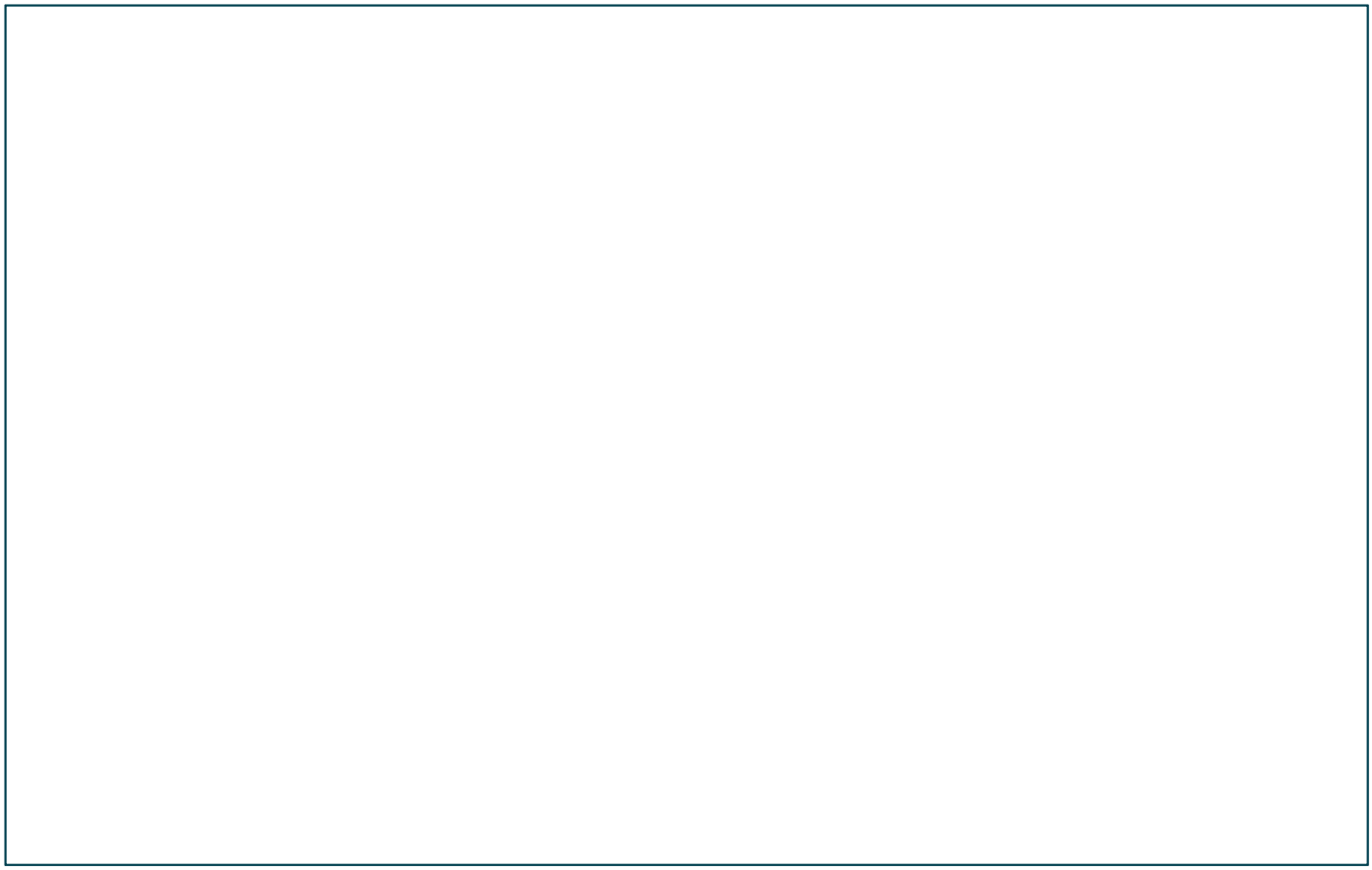
Thinking in React – Parts 1 and 2

Building Web Applications Using React

Objectives

- To understand the development methodology suggested for React applications
- To be able to identify the component hierarchy for a given application
- To be able to build a static version of an application
- Understand how to use props in components
- To be able to test component snapshots and components with props
- To be able to mock components when testing


Thinking In React
React makes developers think about the applications that are being built as they build them.
Facebook's recommendation is to follow these steps when building apps using React:
• Start with a mock
1. Break the UI into a component hierarchy
2. Build a static version in React
3. Identify the minimal (but complete) representation of UI state
4. Identify where your state should live
5. Add inverse data flow



Start with a mock

All UIs to be built should have a mock up or a wireframe.
Data should be available in a suitable format too.
Facebook's guide shows a Product Search page and some JSON for products.

```
[
  {category: "Sporting Goods", price: "$49.99", stocked: true, name: "Football"},
  {category: "Sporting Goods", price: "$9.99", stocked: true, name: "Baseball"},
  {category: "Sporting Goods", price: "$29.99", stocked: false, name: "Basketball"},
  {category: "Electronics", price: "$99.99", stocked: true, name: "iPod Touch"},
  {category: "Electronics", price: "$399.99", stocked: false, name: "iPhone 5"},
  {category: "Electronics", price: "$199.99", stocked: true, name: "Nexus 7"}
];
```

Thinking In React – Part 1
The Component Hierarchy

1. Break the UI into a component hierarchy

Draw boxes around every component and subcomponent in the mock and give them names.

- Image layers may end up being names of components

To help identify, components should only do one thing.

Data model and data should map nicely

`FilterableProductTable`

`SearchBar`

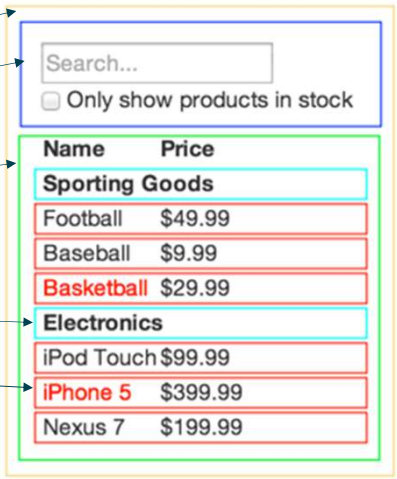
`ProductTable`

`ProductCategoryRow`

`ProductRow`

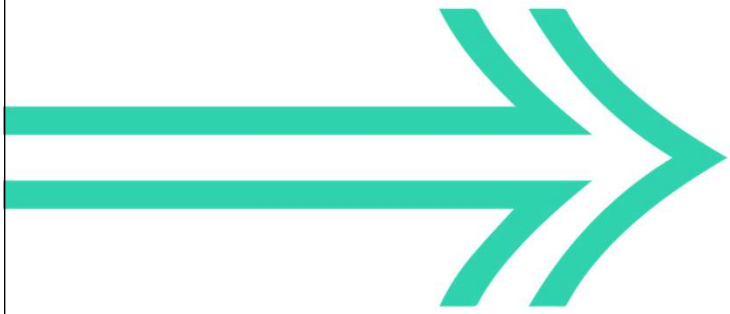
Component hierarchy already defined

Not the only way to implement...



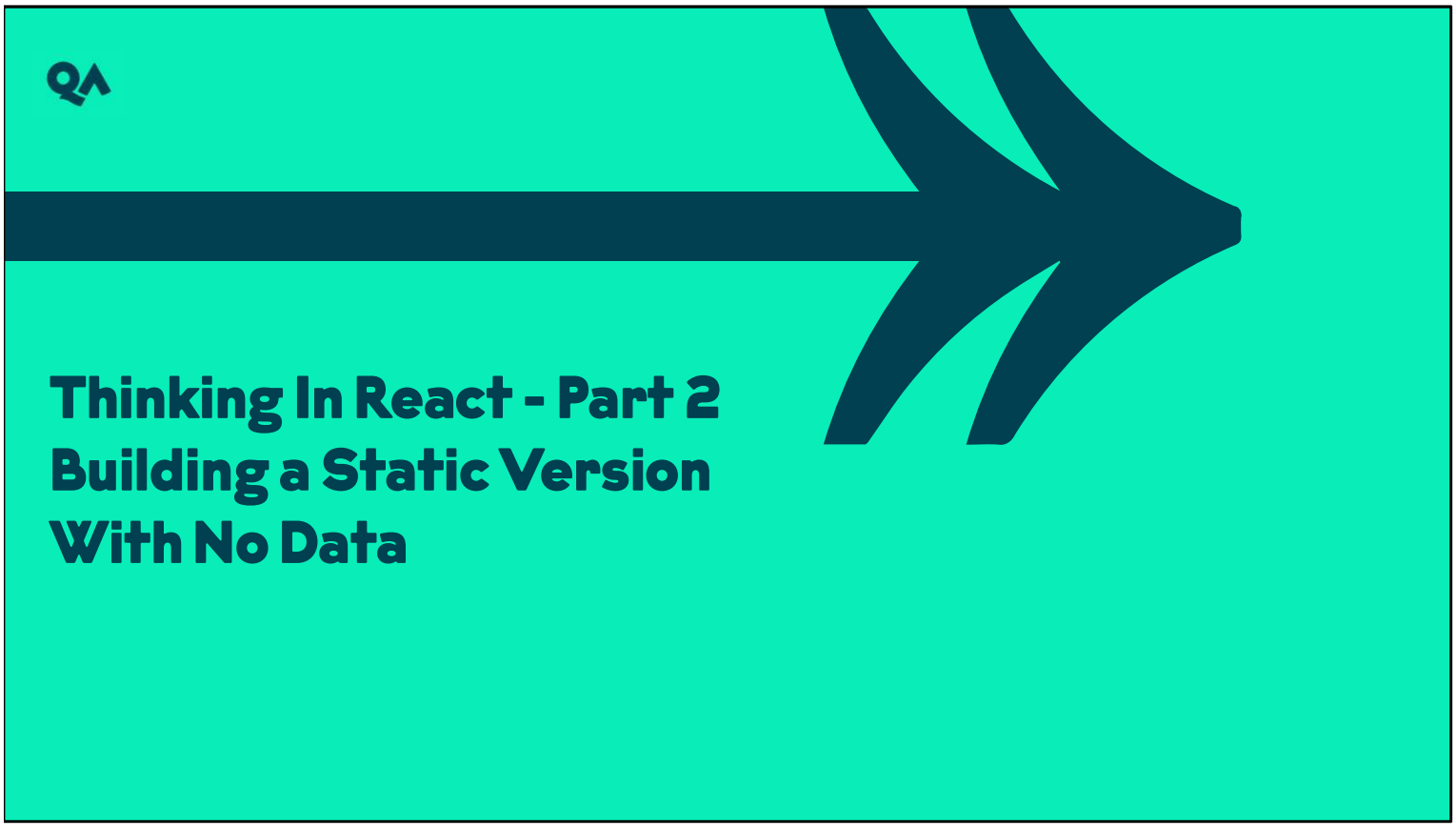
66

QuickLab 6 – Thinking In React Part 1 – Component Hierarchy



In this QuickLab, you will:

- Take provided sample data mock-ups to produce a potential component hierarchies for each required UI


Thinking In React - Part 2
Building a Static Version
With No Data

2. Build a static version in React

Build version that takes data model and renders UI without interactivity.

- Decoupling view and interactivity good as static versions are lots of typing and little thinking and vice versa for interactive versions
- Best practice is to build components that reuse other components and pass data using *props*

→ *state* not used as it is for interactivity

Top-down building as acceptable as bottom-up.

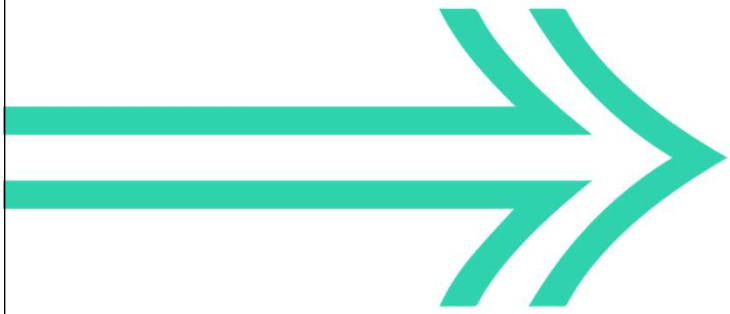
- Bottom-up more common in TDD environments

Helps establish library of reusable components to render each data model.

- Each only has render method

69

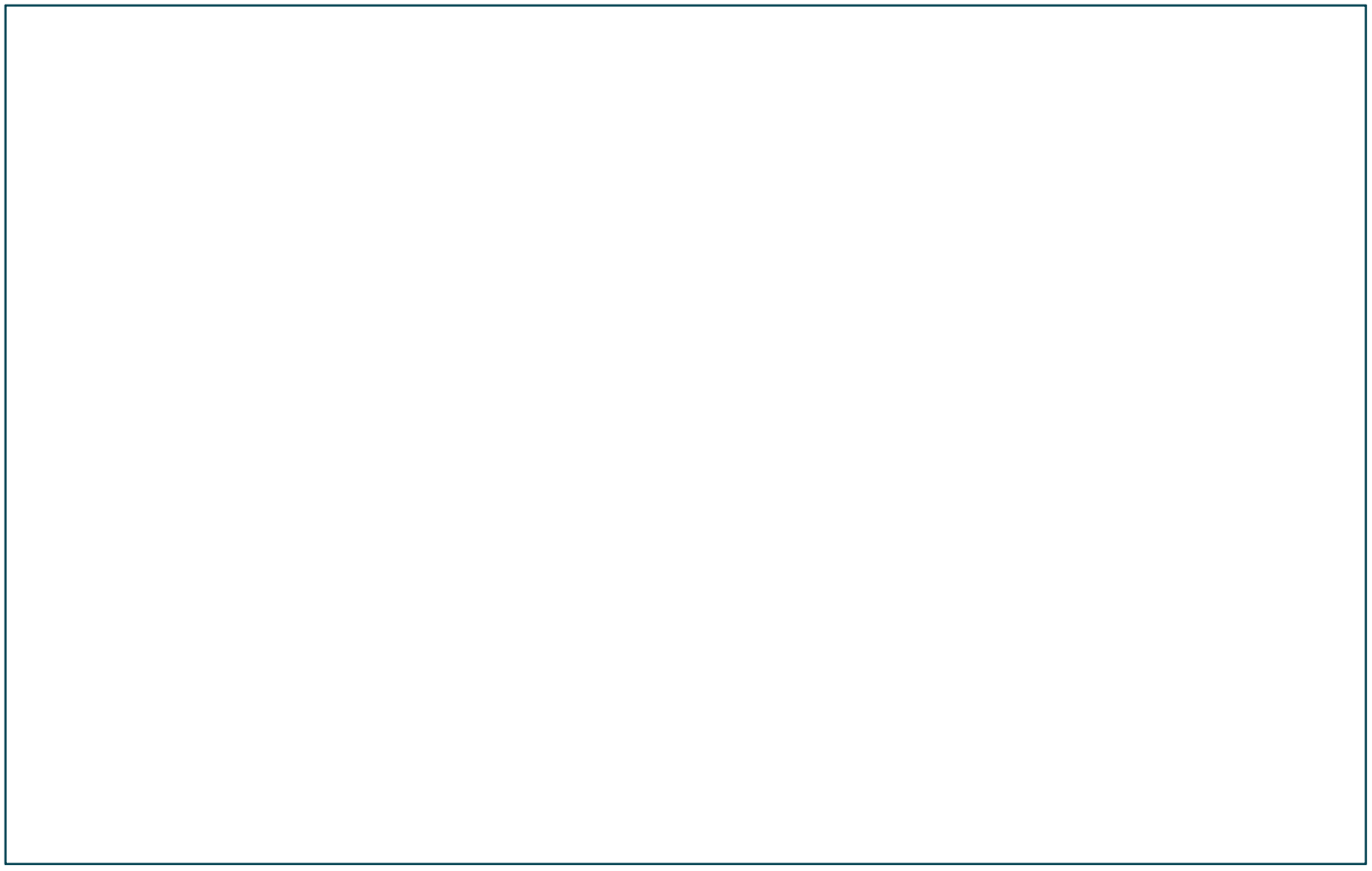
QuickLab 7a – Thinking In React Part 2 – Static Components



In this QuickLab, you will:

- Create header and footer components for the ToDo Application using the provided files and acceptance criteria

Testing Components – Snapshot Testing



TESTING COMPONENTS – SNAPSHOT TESTING

Useful for making sure that the UI does not change unexpectedly.

Process is to:

- Render a UI component
- Take a snapshot
- Compare snapshot to reference snapshot file stored

Test fails if 2 snapshots do not match.

- Because the change was unexpected
- Because the snapshot needs to be updated to new version of the UI component

`react-test-renderer` needs to be installed into the project.

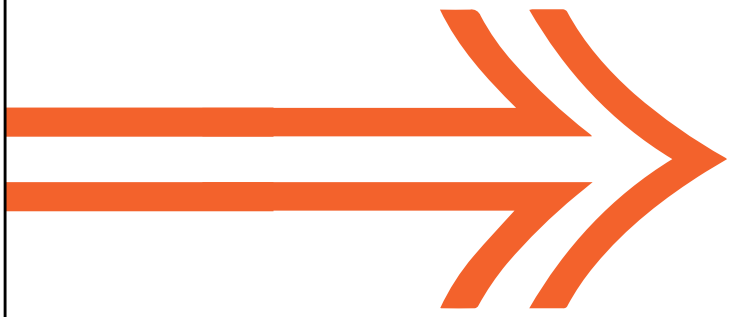
- **`create()`** is passed any component
  - Makes pure JS object as a representation of the React component
  - Use **`toJSON()`** and **`toMatchSnapshot()`** in an **`expect`** statement to compare
  - **`toMatchSnapshot`** compares the created component with the snapshot (and creates a snapshot if one doesn't exist)

TESTING COMPONENTS – SNAPSHOT TESTING EXAMPLE

```
import React from `react`;
const Button = () => (
  <button>Do nothing</button>
);
export default Button;
```

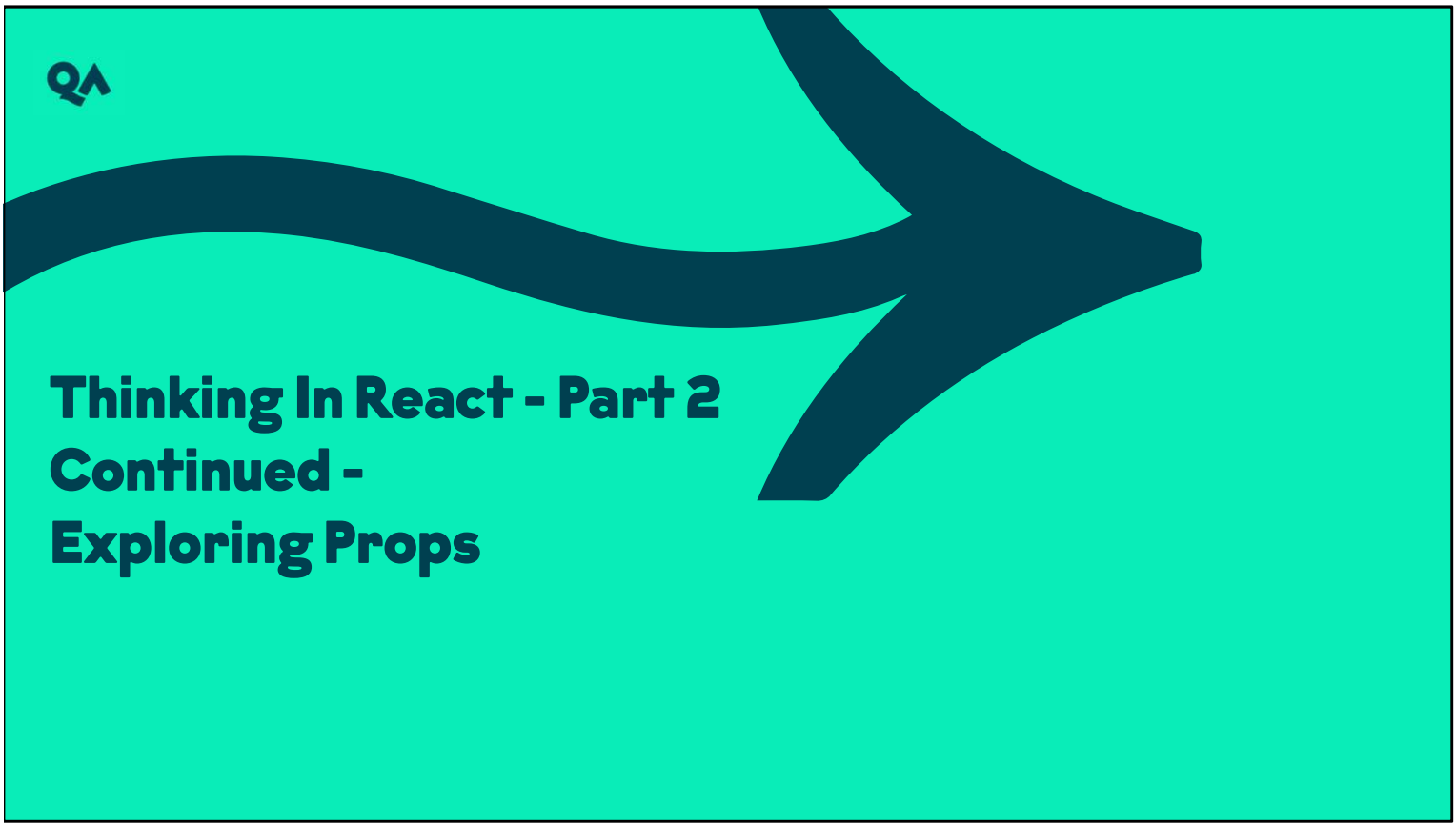
```
import React from `react`;
import { create } from `react-test-renderer`;
import Button from `./Button`;

test(`Test matches snapshot`, () => {
  const button = create(<Button />);
  expect(button.toJSON()).toMatchSnapshot();
});
```

QuickLab 7b – Thinking in React – Part 2 – Test Snapshots



In this QuickLab, you will:

- Write snapshot tests to ensure that the header and footer components do not change unexpectedly


Thinking In React - Part 2
Continued -
Exploring Props

React and data

React only supports UNIDIRECTIONAL data flow.

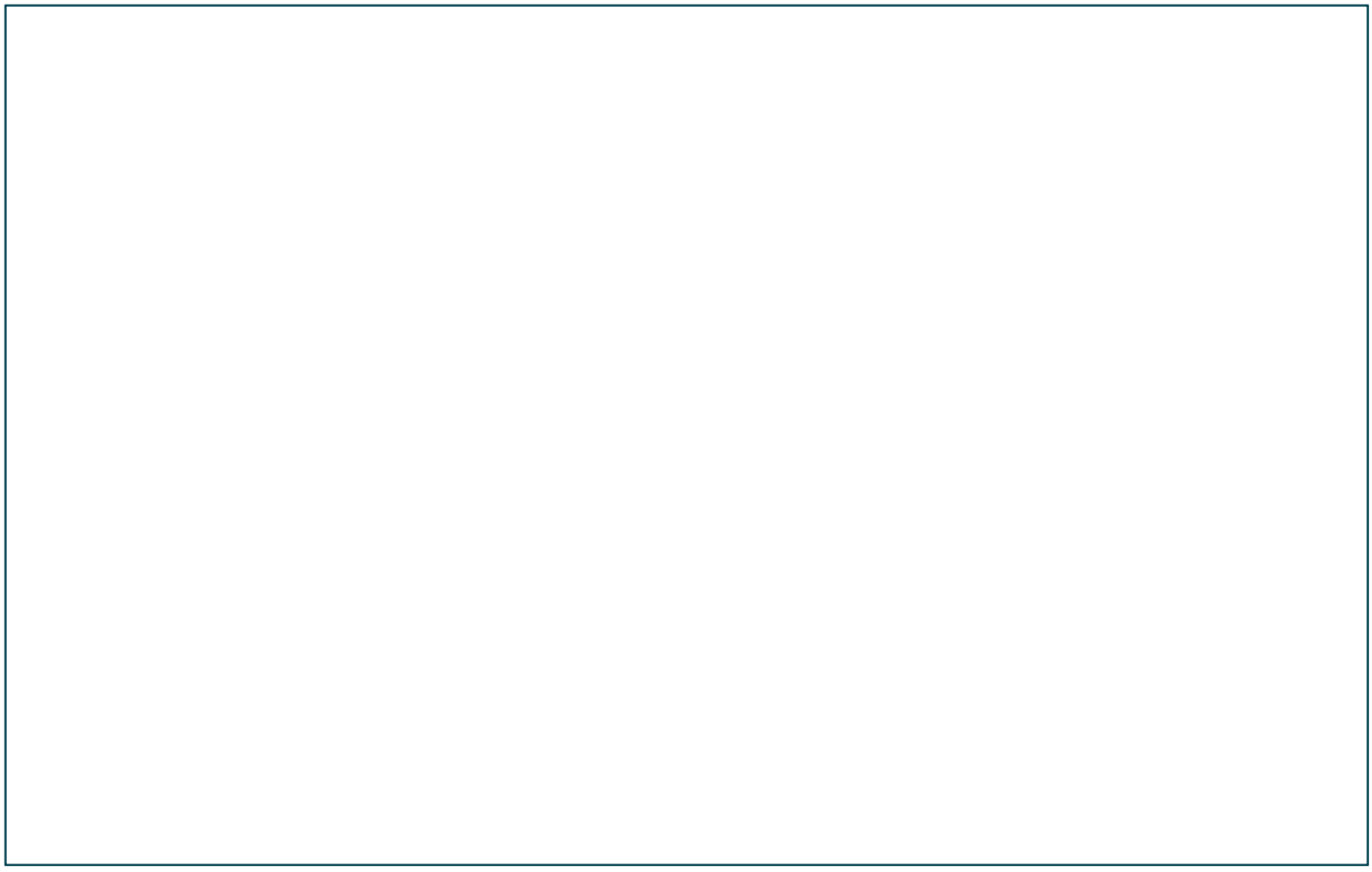
- Data flows from the top of a component tree to the bottom
- Data cannot flow back up the component tree

Data that does not change over the lifetime of the component should be considered as **props**.

Data that can change should be considered as **state**.

- State should be the single source of truth for changing data
- All components that rely on this should receive the data as props
- State should be in the highest common component of those that require the data

76



What are props?

"props are a way of passing data from parent to child".

http://facebook.github.io/react/docs/thinking-in-react.html

- i.e. a communication channel between components that always moves from the top (parent) to the bottom (child)

`props` are immutable - once set, they cannot change

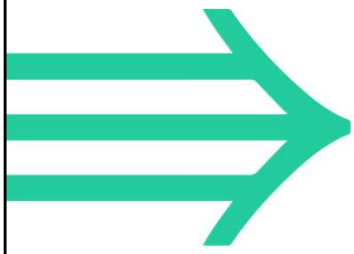
```
…<App headerProp = "Header from attr" />, doc…
```

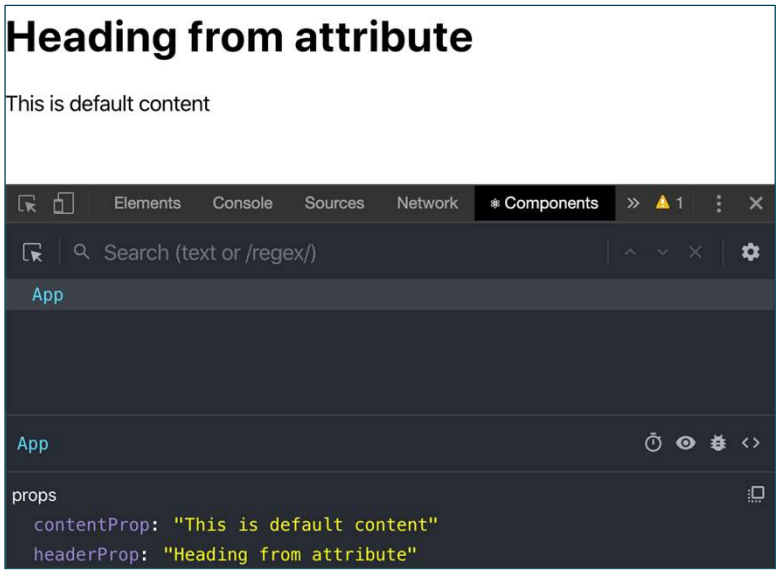
`props` can be added as attributes in the component used when rendering the component from **`ReactDOM.render`**

And/or **`default props`** can be defined under the component declaration in the **.jsx** file:

```
App.defaultProps = {
      headerProp : `Header from default`,
      contentProp : `Content from default`
}
```

Using Props in Components





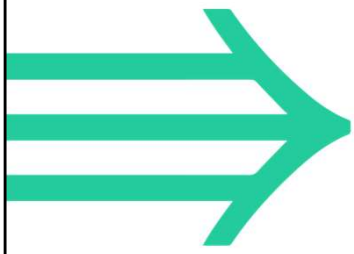

```
const App = props => (
  <div>
    <h1>{props.headerProp}</h1>
    <p>{props.contentProp}</p>
  </div>
);
```

`props` must be passed to a function component as an argument for it to be aware of them.

`props` rendered to the browser through the component return – either the default, or overriding value (if supplied)

78

Props Example



```
import React from `react`;
import ReactDOM from `react-dom`;
const App = props => {
  return (
    <div>
      <h1>{props.headerProp}</h1>
      <p>{props.contentProp}</p>
    </div>
  );
};
App.defaultProps = {
  headerProp: `This is the default heading`,
  contentProp: `This is default content`
}
ReactDOM.render(<App headerProp = "Header from attribute" />,
  document.querySelector('#app'));
```

props rendered in the component here

defaultProps set here

This component will be rendered as expected with the header being displayed from the overriding attribute setting and the content being rendered from the default.

79

Props can have type and validation...

Sub-object **`propTypes`** can be used for both typing and validation.

- Uses **`PropTypes`** class from **prop-types** npm package (from React 15.5)
- Useful for ensuring correct usage of components
- Makes code more readable – can see how component should be used

Typing:

- Any valid JavaScript type can be used
- Will produce console warning if correct type is not used for prop

Validation:

- To ensure that a prop has a value supplied
- **`.isRequired`** is chained to **`propTypes`** declaration
- Will produce console warning if prop is not available

Undeclared props are ignored by the browser.

When using **create-react-app**, the **prop-types** package is included by default.

- Only included in bundle if imported in app

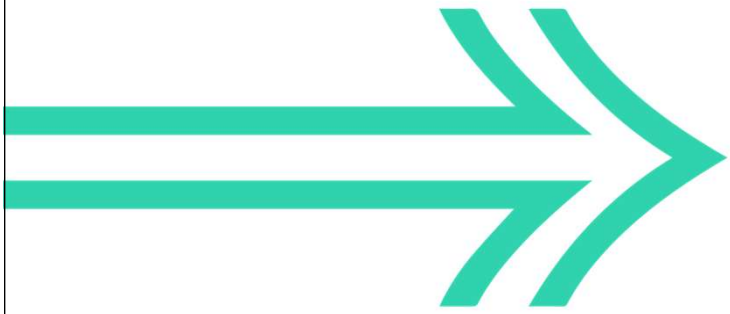
Prop-typing and validation example

```
import React from `react`;
import ReactDOM from `react-dom`;
import PropTypes from `prop-types`;

const App = props => {
  return (
    <>
      <h1>{props.headerProp}</h1>
      <p>{props.contentProp}</p>
      <p>Value of numberProp is: {props.numberProp}</p>
    </>
  );
};

App.defaultProps = {
  headerProp: `This is the default heading`,
  contentProp: `This is default content`
}

App.propTypes = {
        headerProp: PropTypes.string.isRequired,
        contentProp: PropTypes.string.isRequired,
        numberProp: PropTypes.number
}

ReactDOM.render(<App numberProp={10} />,
  document.querySelector('#app'));
```

In this example, the number prop has to be a number if it is supplied, else there will be a console warning. Therefore, this component will be rendered as expected with the header and content being displayed from the default and **`numberProp`** being evaluated to **`10`** through the attributes.

81

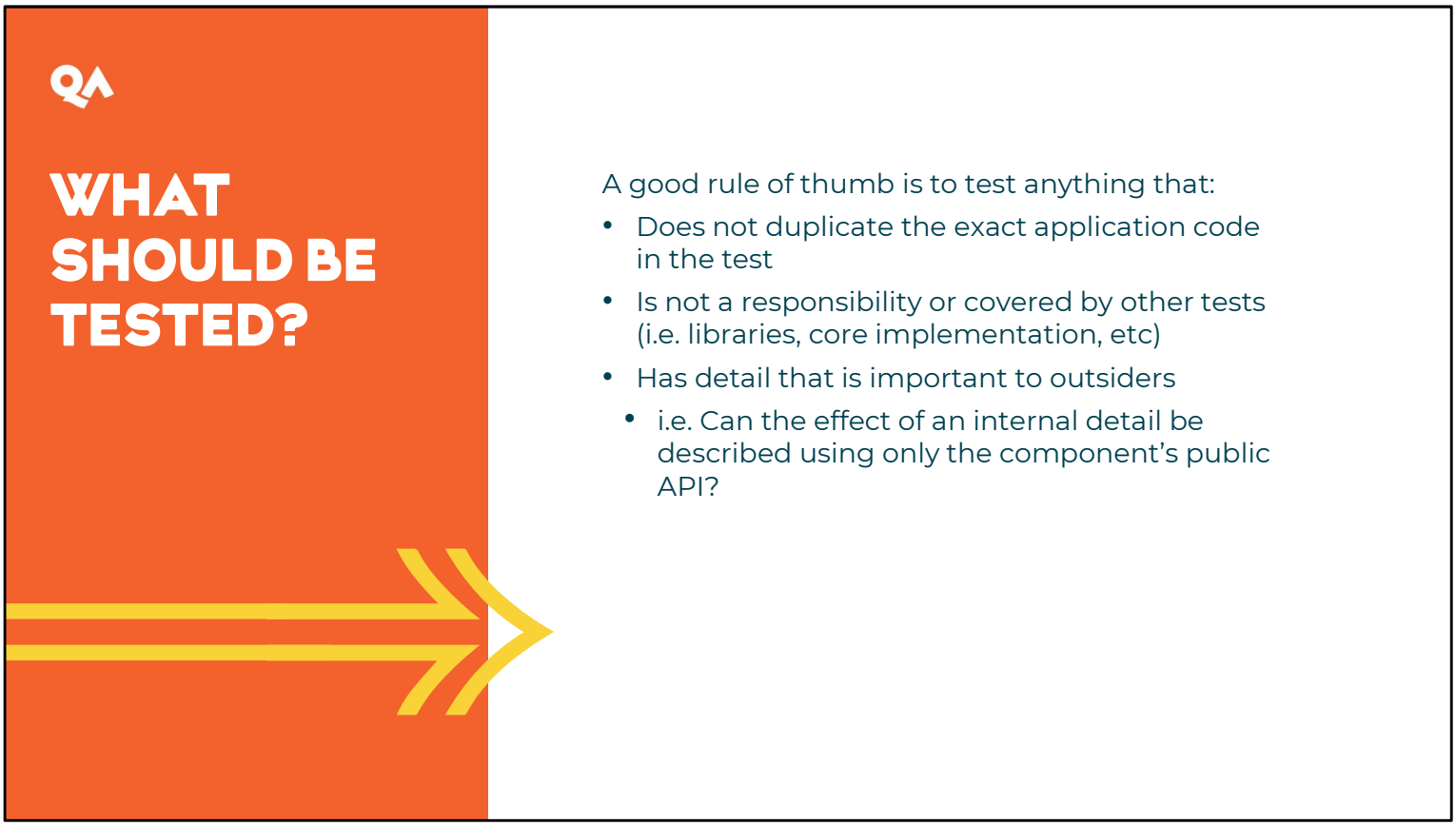
QuickLab 8 – Exploring Props

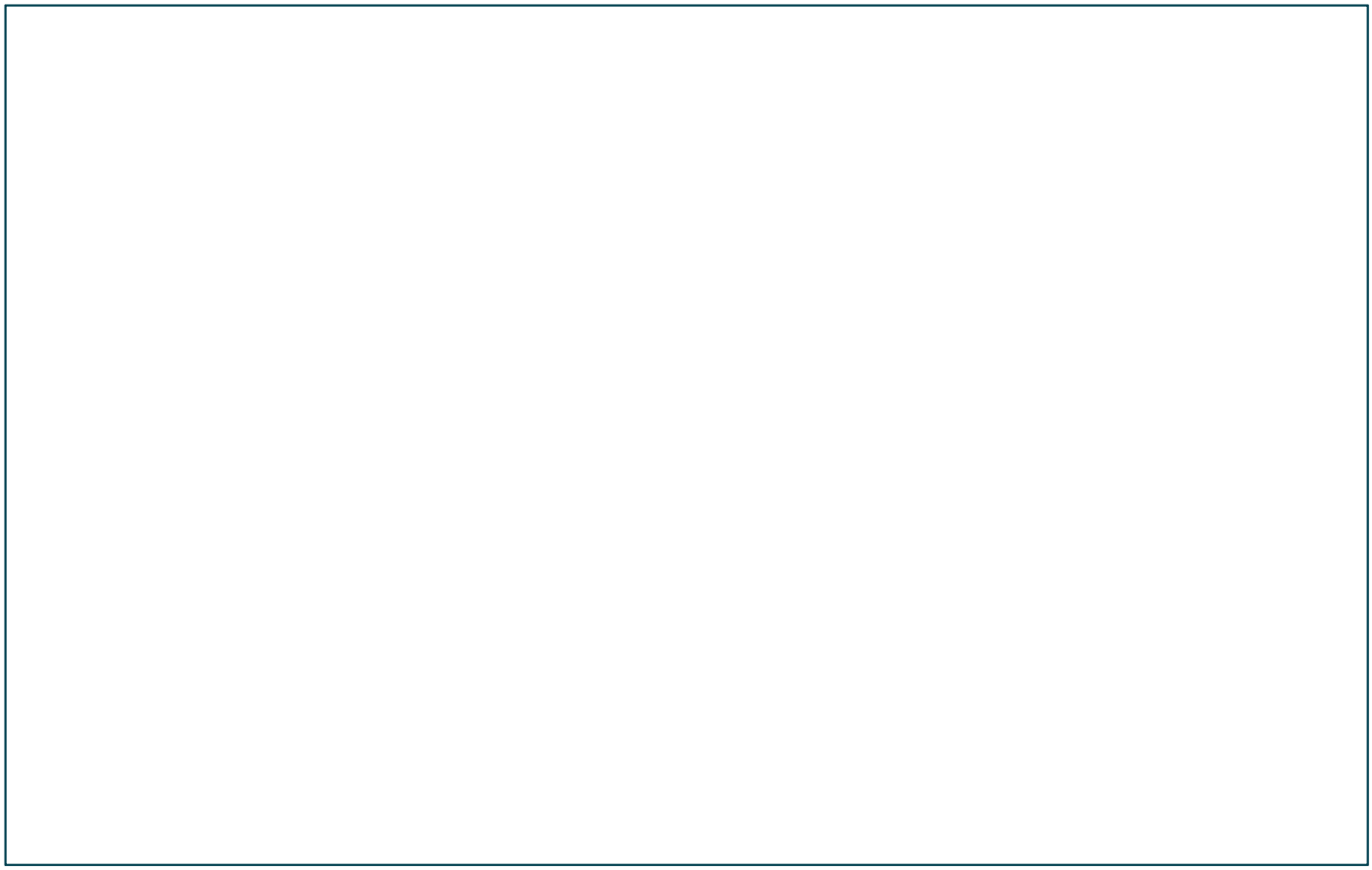


In this QuickLab, you will:

- You will experiment with adding props to a component
- Use default props
- Use prop-types

Testing Components With Props


WHAT SHOULD BE TESTED?
A good rule of thumb is to test anything that:
• Does not duplicate the exact application code in the test
• Is not a responsibility or covered by other tests (i.e. libraries, core implementation, etc)
• Has detail that is important to outsiders
• i.e. Can the effect of an internal detail be described using only the component's public API?



TESTING PROPS ARE RENDERED

There is no need to test if **`defaultProps`** are rendered.

- This is part of the implementation of React and will be tested already

There is no need to test if **`PropTypes`** work.

- This is the concern of the **`prop-types`** package and will be tested already

We should test to see if the value of props received is rendered, especially if this can change at runtime.

CREATE, RENDERER, ROOT AND INSTANCES

create can be used to make a pure JavaScript object representation of a component - **testRenderer**

```
const testRenderer = create(<MyComponent />);
```

A **testInstance** can be created using root – returns an object that is useful for making assertions about specific nodes in the tree

```
const testInstance = testRenderer.root;
```

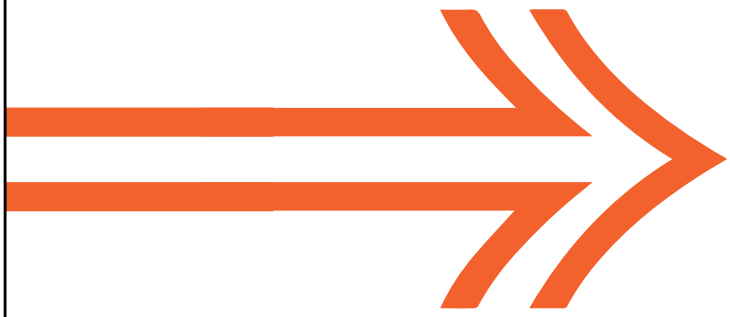
Several helper methods exist, such as **find()**, **findByType()**, **findByProps()**, **findAllByType()** and **findAllByProps()** to get **testInstances** within the tree

```
const testInstanceP =
    testInstance.findByType(`p`);
```

Using **children** on the **testInstance** allows access to the child nodes of it – including its **textContent**

```
expect(testInstanceP.children).toEqual(`Test`);
```

QuickLab 9 – Testing Components With Props



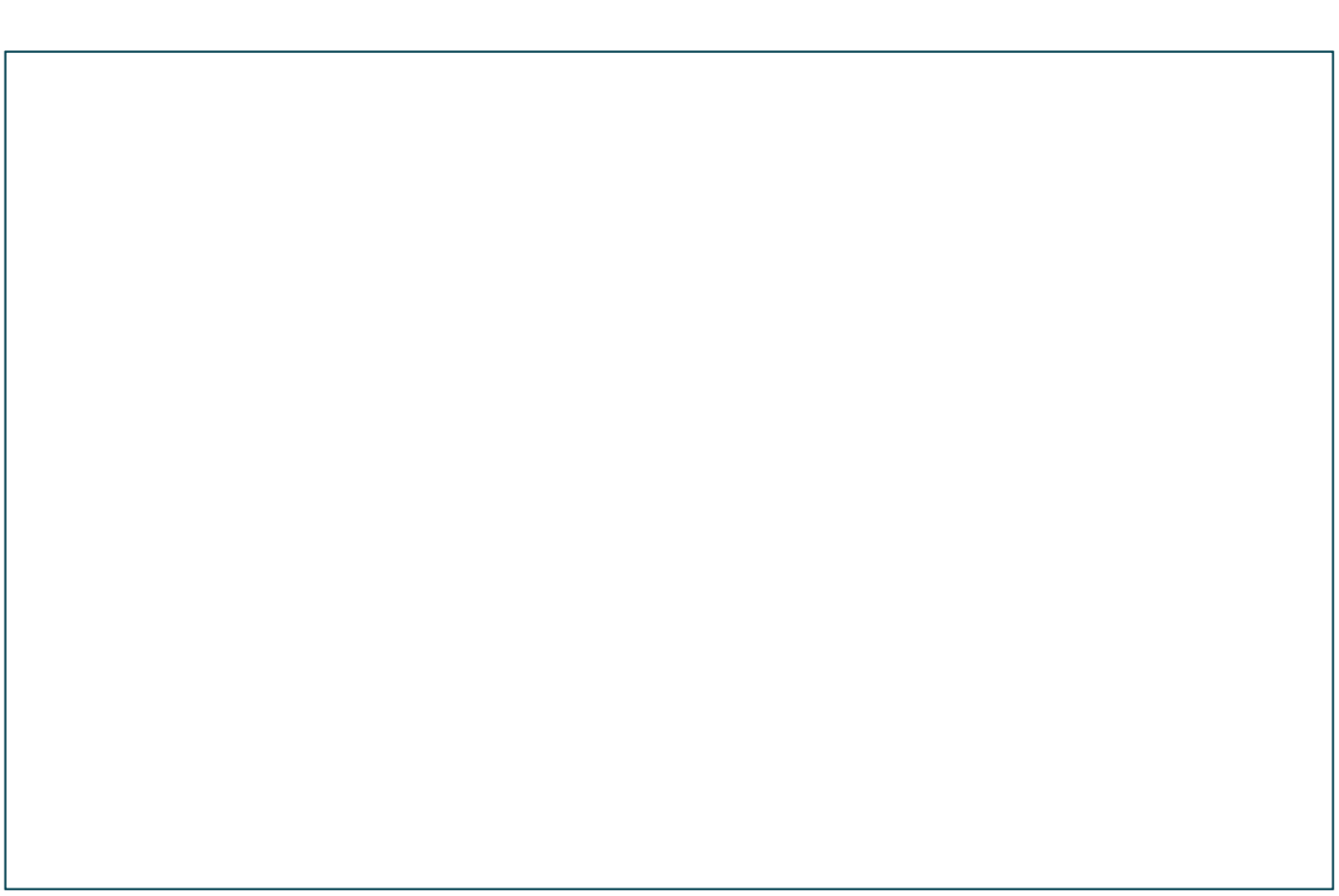
In this QuickLab, you will:

- Test the component with props to ensure that passed props are rendered

Objectives

- ~~To understand the development methodology suggested for React applications~~
- ~~To be able to identify the component hierarchy for a given application~~
- To be able to build a static version of an application
- Understand how to use props in components
- ~~To be able to test component snapshots and components with props~~
- To be able to mock components when testing

Thinking In React - Part 2
Using Static Data in Applications



Data sources

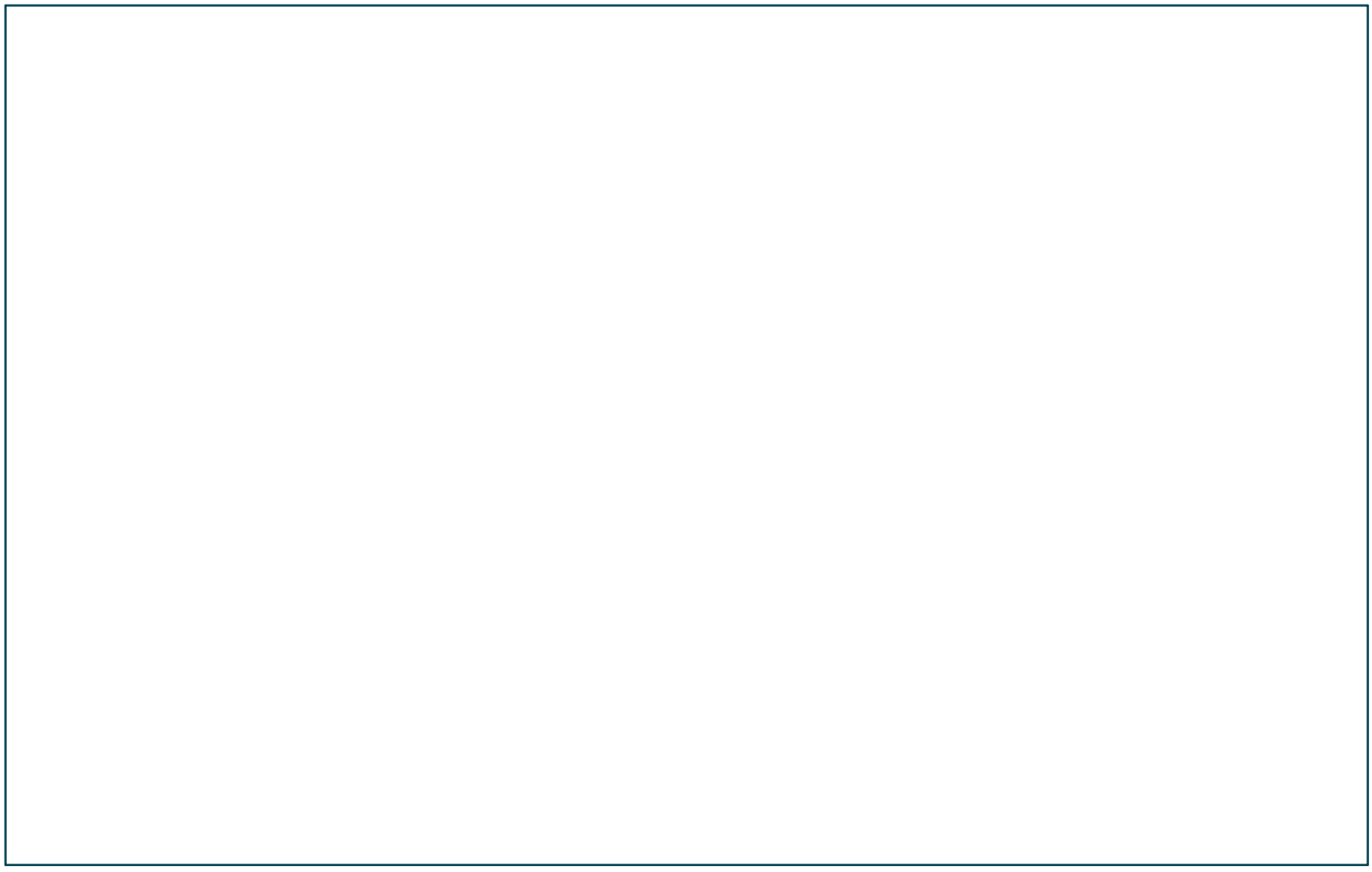
React is for producing front end components.

Need to be able to convert data from data sources into state and/or props.

- E.g. Displaying a list of items in a table from a HTTP search response

Can convert data source into an array or create one from it.

- Array can then be stored in state
- Component can then use JavaScript map function to return state array as modified array of components which are then displayed on the screen



Importing static data

If a JSON file can be made available, this can be imported with a name into the application.

An alias is given so that the array can be used.

Often in React an array of data is mapped to return a component.

- Each element in the array is passed to the component as a prop and used to generate a unique component
- The mapped array is used to display the contents of the array

```
import React from `react`;
import Data from `./Data`;
import someData from `./someData.json`;
const AllData = () => {
  const allData = someData.map(data =>
    <Data item={data} key={data.id} />);
  return (
    <>
      <h2>The data supplied is:</h2>
      <div>
        {allData}
      </div>
    </>
  );
};
export default AllData;
```

91

Whilst this may incur a small amount of technical debt, it is often worth it to ensure that components function correctly from the start.

Using a dataset - map() example

Imagine we have an array called **`people`** that stores a person object holding an ID, their first name and their country of origin, imported from an external source such as a JSON file.

Array can then be used in the render function to produce JSX code for each person in the array through the **`map()`** function.

```
…
   {people.map(person => (
     <p>This is {person.name}, they're from{person.country}</p>)};
…
```

`map()` works in the following way

1. For every item in the array, takes item itself and its index.
2. Creates a new anonymous function that receives item and index and returns a value of the modified data.

Remember: => (called fat-arrow notation) is ES2015+'s way of declaring a return from an anonymous function.

Multiple HTML elements could be used here as long as they have a wrapping element, eg.

```
…
            {this.state.people.map((person, index) => (
                        <div>
                                    <p>Hello, {person.name}!</p>
                                    <p>You're from {person.country}.</p>
                                    <hr />
                        <div>
            ))};
…
```

Array/iterators and keys

Running the code in this way would produce a warning on the console.

React likes each every top-level item printed by a loop to have a KEY attribute that identifies it uniquely.

- Called RECONCILIATION
- Helps React identify which items have been modified
- Best practice to select a string that uniquely identifies a list item amongst its siblings (no need to be globally unique)

  Most often ID from the dataset.

  Can be index of array as a last resort.
- Keys should be kept on the array element produced rather than the root element

For more information on RECONCILIATION see:
https://facebook.github.io/react/docs/reconciliation.html

Arrays/iterators and keys

Dealing with the warning is important for more advanced apps.

The key attribute should be added to the wrapping element of the return of the anonymous function.

```
…
  {people.map(person =>
    <p key={person.id.toString()}>
      This is {person.name}, they're from{person.country}
    </p>
   )};
…
```


If multiple HTML elements used, the key should be included in the wrapping element, eg.

```
…
            {this.state.people.map((person, index) => (
                        <div key={person.id.toString()}>
                                    <p>Hello, {person.name}!</p>
                                    <p>You're from {person.country}.</p>
                                    <hr />
                        <div>
            ))};
…
```

Sub-components

Data can be passed from state of a parent component to a child using props.

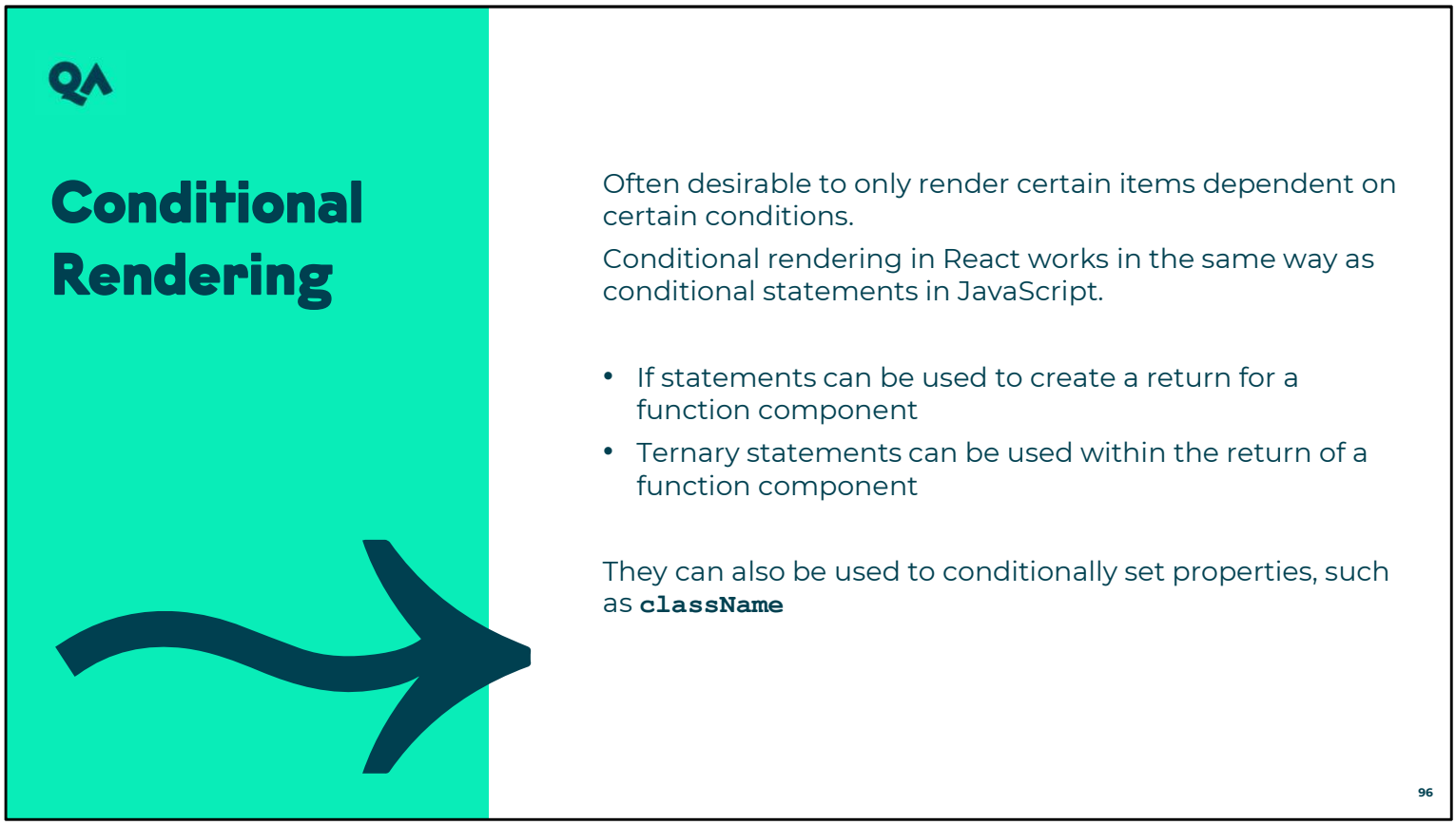
- Using a sub-component - change code in parent **`render()`** function

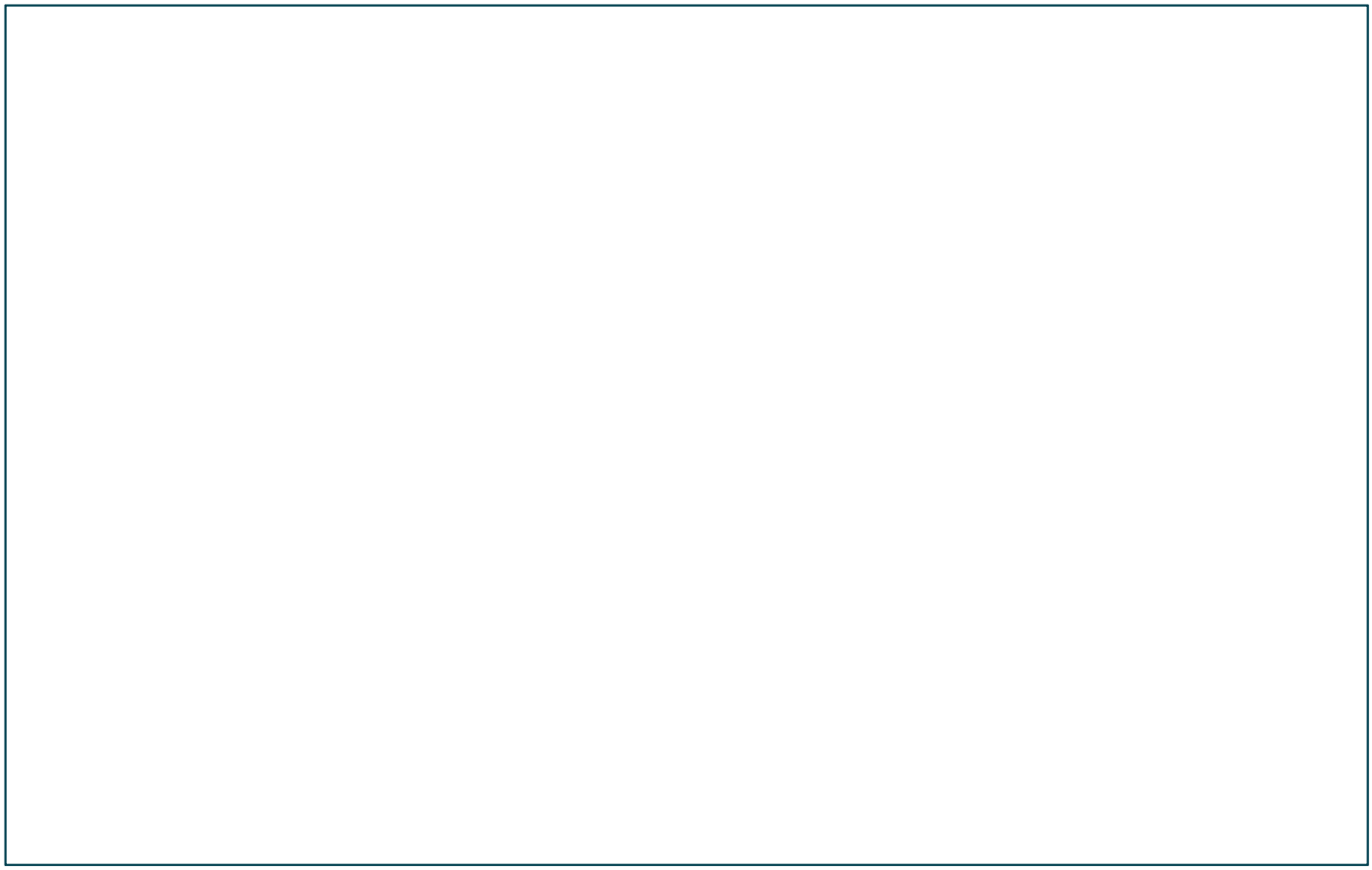
```
…
   {people.map(person => (
     <Person key={person.id} name={person.name}
       country= {person.country} />
   ))};
…
```

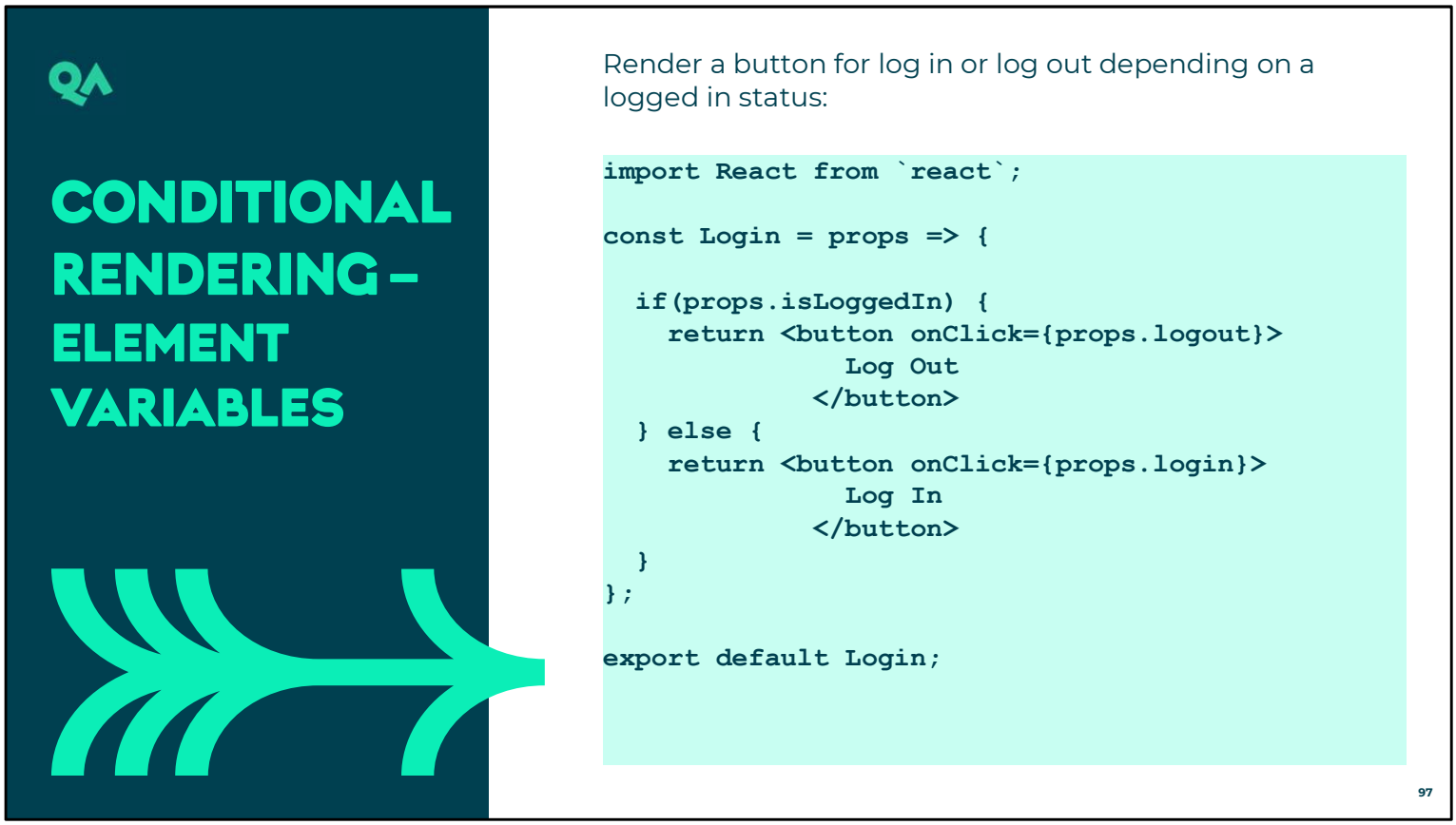
- Create a Person function component

```
const Person = props => (
  <p>This is {props.name}, they're from {props.country}</p>
);

export default Person;
```


Note that the key attribute stays within the parent as this is the wrapping component.


Conditional Rendering
Often desirable to only render certain items dependent on certain conditions.
Conditional rendering in React works in the same way as conditional statements in JavaScript.
• If statements can be used to create a return for a function component
• Ternary statements can be used within the return of a function component
They can also be used to conditionally set properties, such as className
96




CONDITIONAL RENDERING – ELEMENT VARIABLES
Render a button for log in or log out depending on a logged in status:
import React from `react`;
const Login = props => {
if(props.isLoggedIn) {
return <button onClick={props.logout}>
Log Out
</button>
} else {
return <button onClick={props.login}>
Log In
</button>
}
};
export default Login;
97

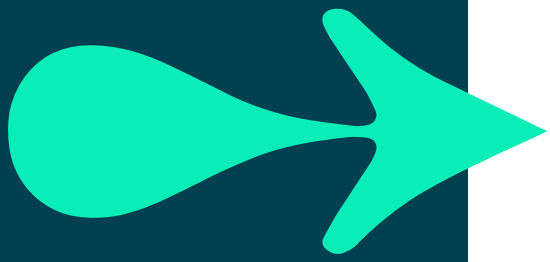
CONDITIONAL RENDERING – INLINE IF-ELSE WITH CONDITIONAL OPERATOR

The previous example could have been re-written as below, using a ternary statement (with an additional example):

```
import React from `react`;

const Login = props => (
  <>
    {props.isLoggedIn ? (
      <button onClick={props.logout}>
        Log Out
      </button>
    ) : (
      <button onClick={props.login}>
        Log In
      </button>
    )}
  </>
);
export default Login;
```

98

CONDITIONAL RENDERING – INLINE IF WITH LOGICAL && OPERATOR

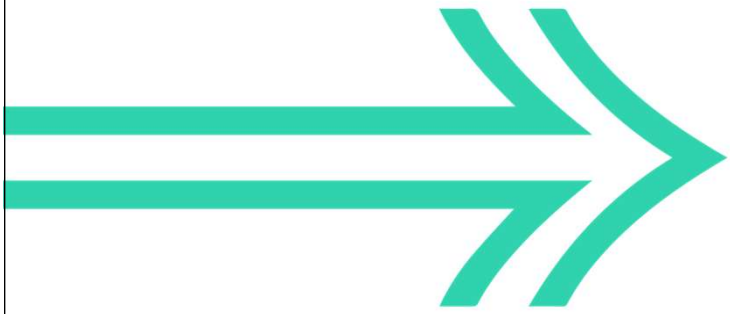


Render an element dependent on a condition. JSX expressions can be embedded by using curly braces, including the use of the && logical operator:

```
import React from `react`;

const UnreadMessages = props => (
  <>
    <h2>Welcome back, {props.name}!</h2>
    {props.unreadMessages.length > 0 &&
      <p>
         You have {props.unreadMessages.length}
         unread messages.
      </p>
    }
  </>
);

export default UnreadMessages;
```

99

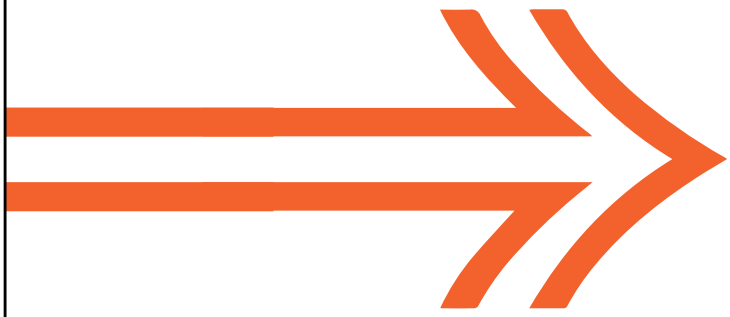
Quick Lab 10a – Thinking in React Part 2 – A Static Version – Components with Static Data



In this QuickLab, you will:

- Create components to display all of the ToDos
- Use props to pass some static data
- Create multiple components from a data-set
- Conditionally use values and/or markup

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



In this QuickLab, you will:

- Write tests to ensure that the components render correctly dependent on the props and data they use

React and Forms – Static Forms

Getting React to render a form is straightforward.

Returning the HTML required to create the form is sufficient.

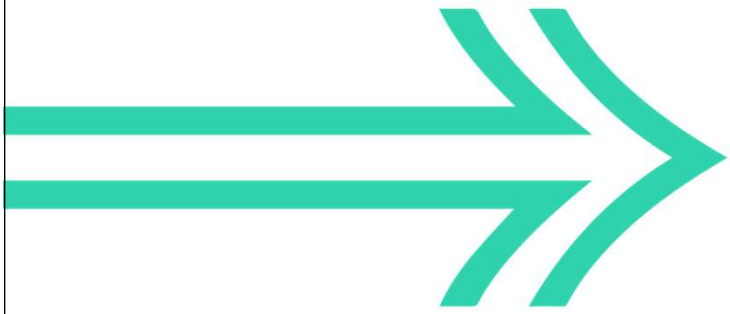
For the static version of the application, this is all that is needed.

Static forms need not be tested (unless there is interdependencies and/or conditional rendering).

Form interactivity will require state.

102

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



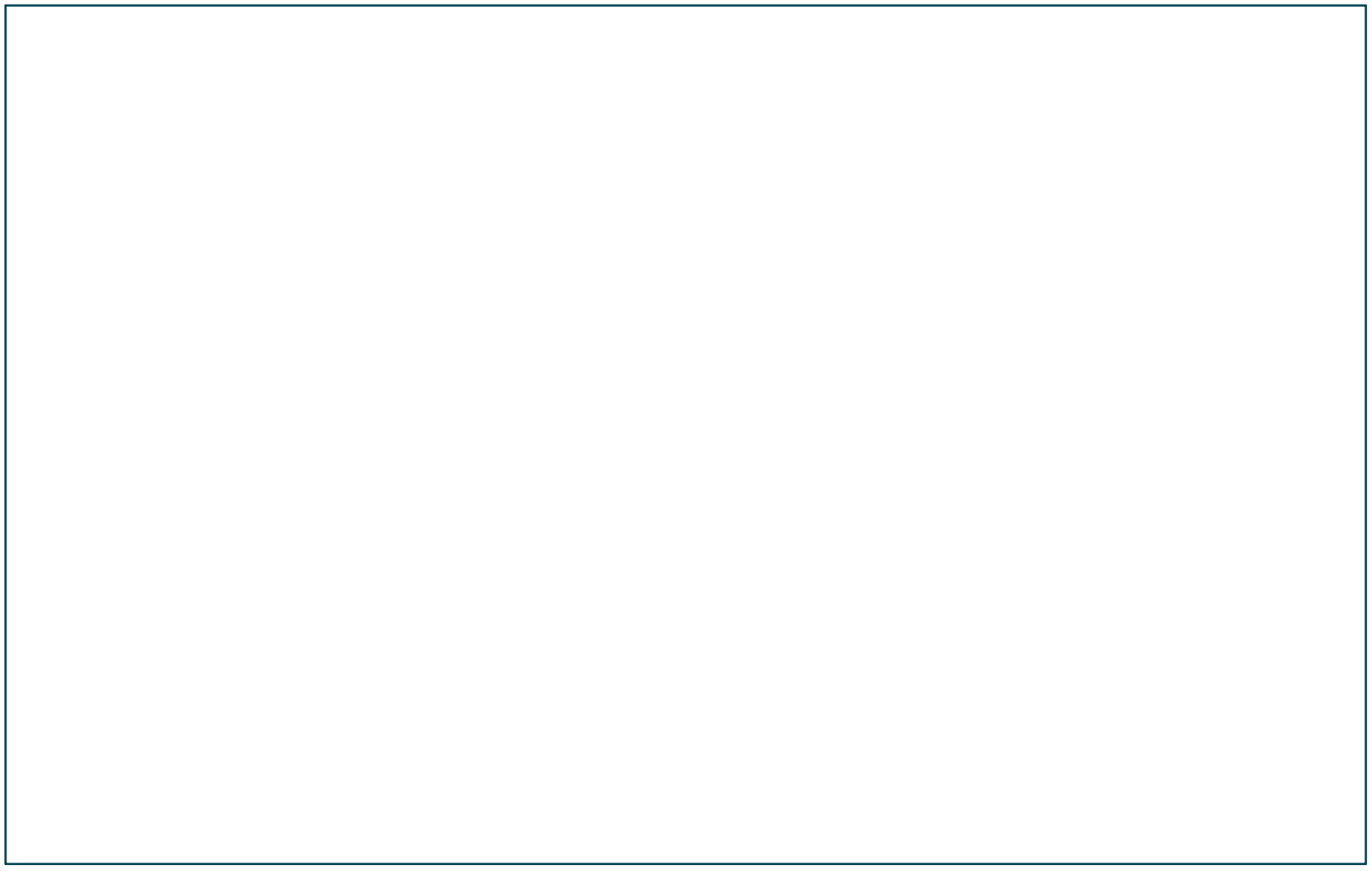
In this QuickLab, you will:

- Create components to add or edit a ToDo

Objectives

- ~~To understand the development methodology suggested for React applications~~
- ~~To be able to identify the component hierarchy for a given application~~
- ~~To be able to build a static version of an application~~
- ~~Understand how to use props in components~~
- ~~To be able to test component snapshots and components with props~~
- To be able to mock components when testing

Mocking Components for Tests



Using jest.mock to return a component

Why mock components?

- To unit test a component that renders other components

  Not interested in the sub-component's implementation

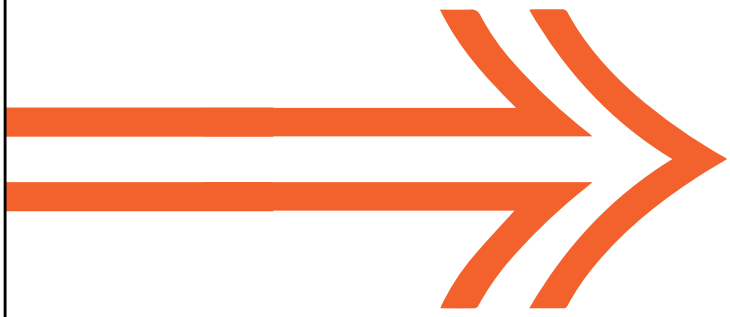
How to mock a component:

- Call `jest.mock(`./path/to/component/to/mock`, [, options]);`
- Supply a callback in options that returns a function to mock the component

```
jest.mock(`./components/myComponentToMock`, () => {
  return function MyMockedComponent () {
    return <span>Mocked Component</span>
  }
});
```

106

QuickLab 10d – Testing Components With Mocks



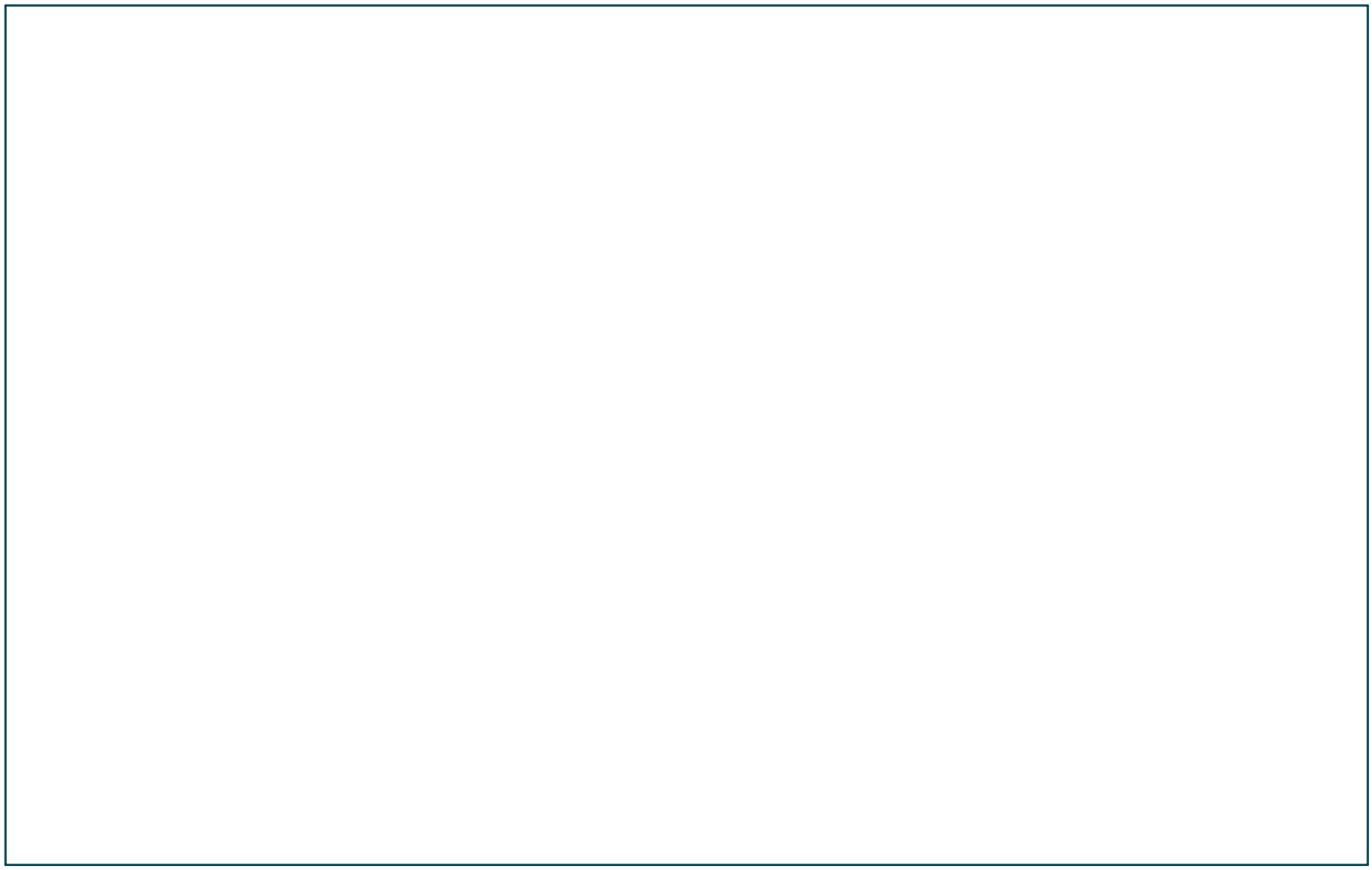
In this QuickLab, you will:

- Test that the form renders correctly, mocking the DateCreated component

Objectives

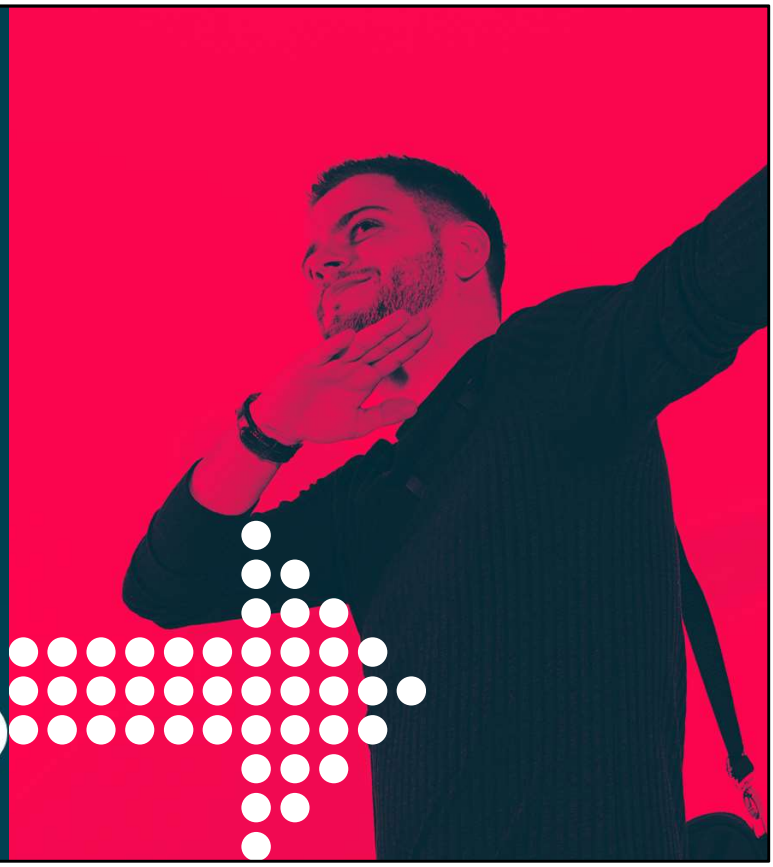
- ~~To understand the development methodology suggested for React applications~~
- ~~To be able to identify the component hierarchy for a given application~~
- ~~To be able to build a static version of an application~~
- ~~Understand how to use props in components~~
- ~~To be able to test component snapshots and components with props~~
- ~~To be able to mock components when testing~~


Thinking In React –
Parts 3 and 4
Identifying State

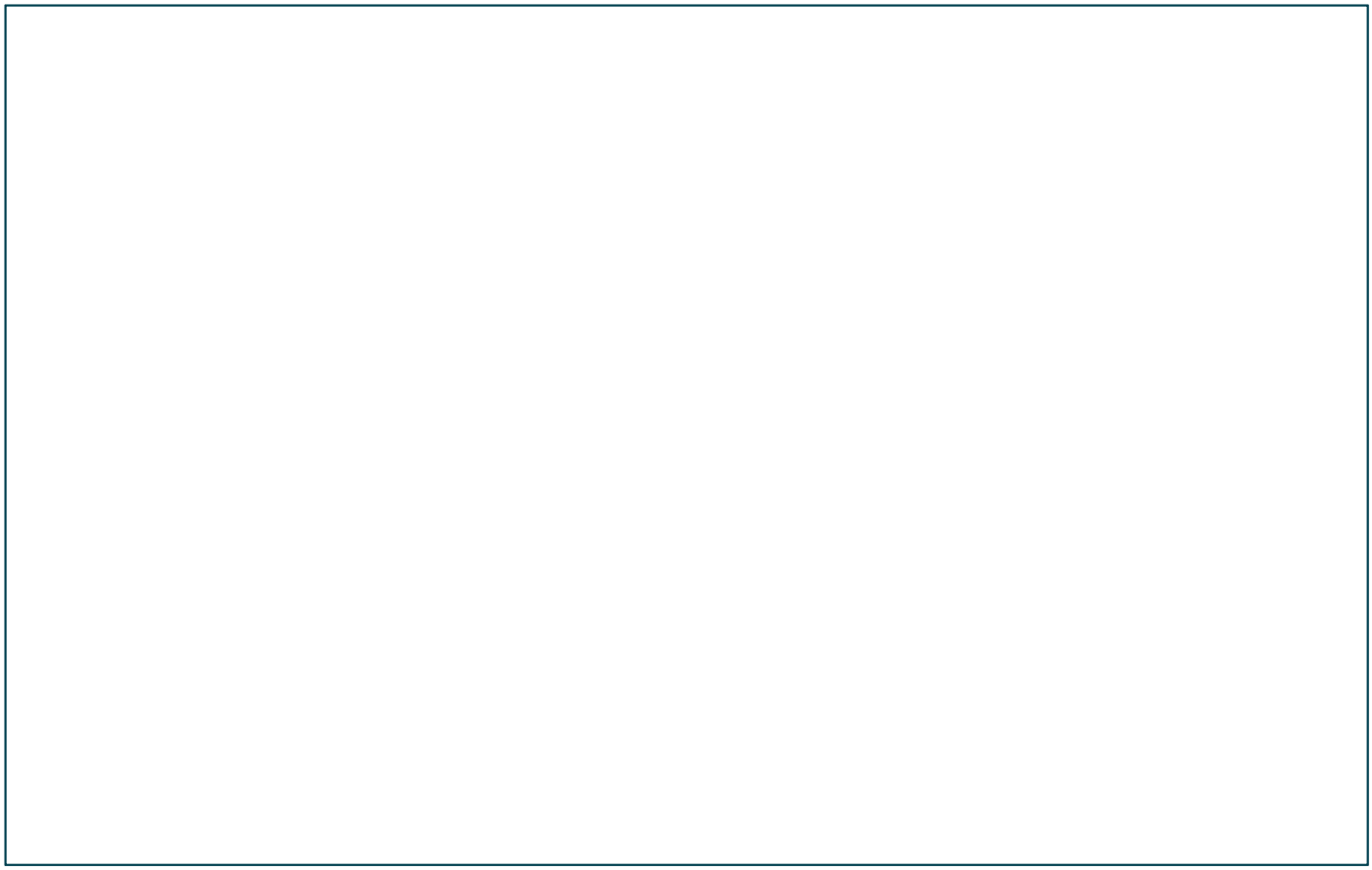


Objectives

- To be able to identify state in an application and add it to the appropriate component
- To be able to work with form data within the application




Thinking In React – Part 3
Identify state



React and data

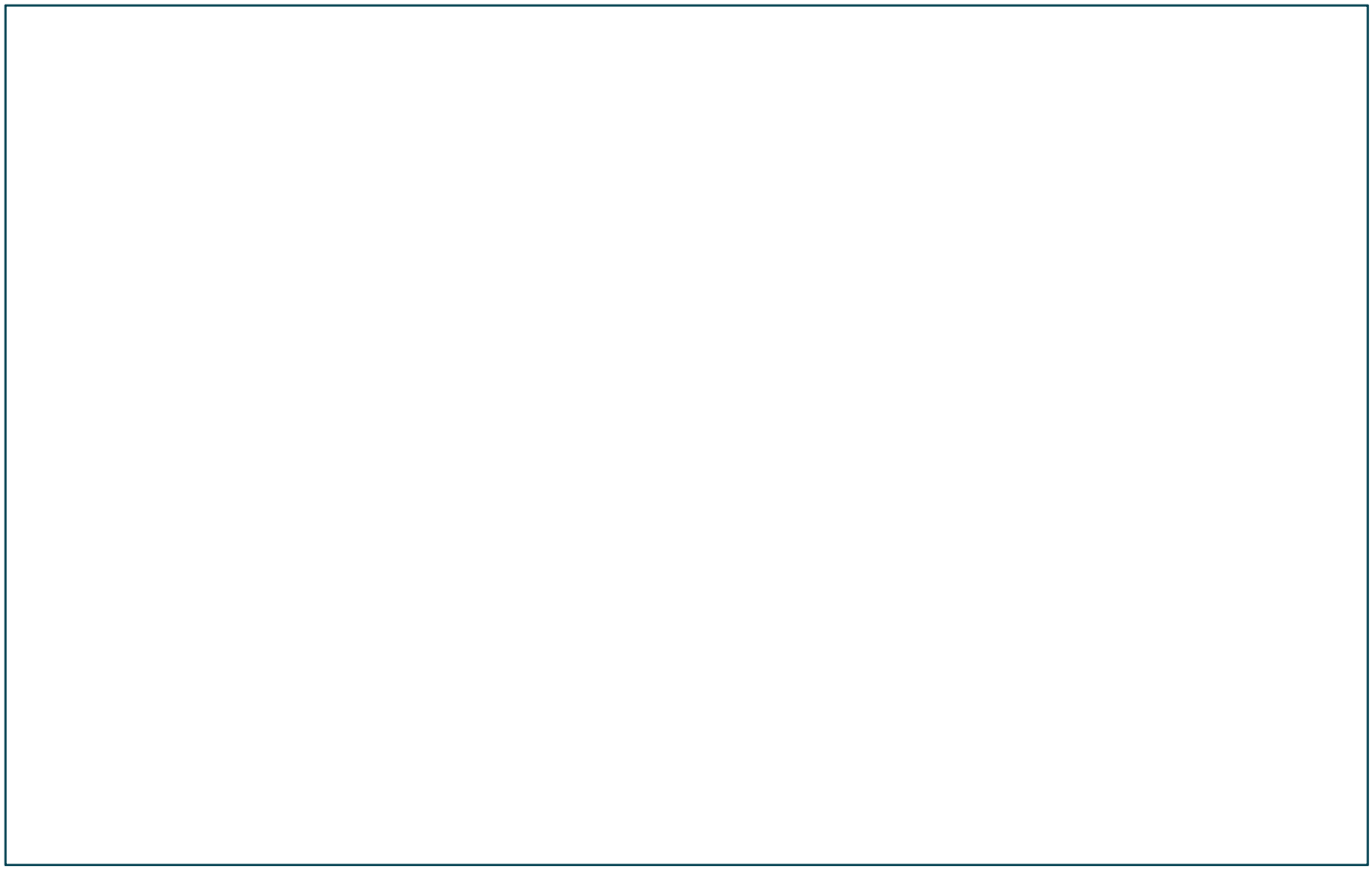
React only supports UNIDIRECTIONAL data flow.

- Data flows from the top of a component tree to the bottom
- Data cannot flow back up the component tree

Data that does not change over the lifetime of the component should be considered as **props**.

Data that can change should be considered as **state**.

- State should be the single source of truth for changing data
- All components that rely on this should receive the data as props
- State should be in the highest common component of those that require the data



3. Identify minimal (complete) representation of UI state

Require ability to trigger changes.

- Made easy through use of *state*

Best practice to first think of minimal set of mutable state app needs.

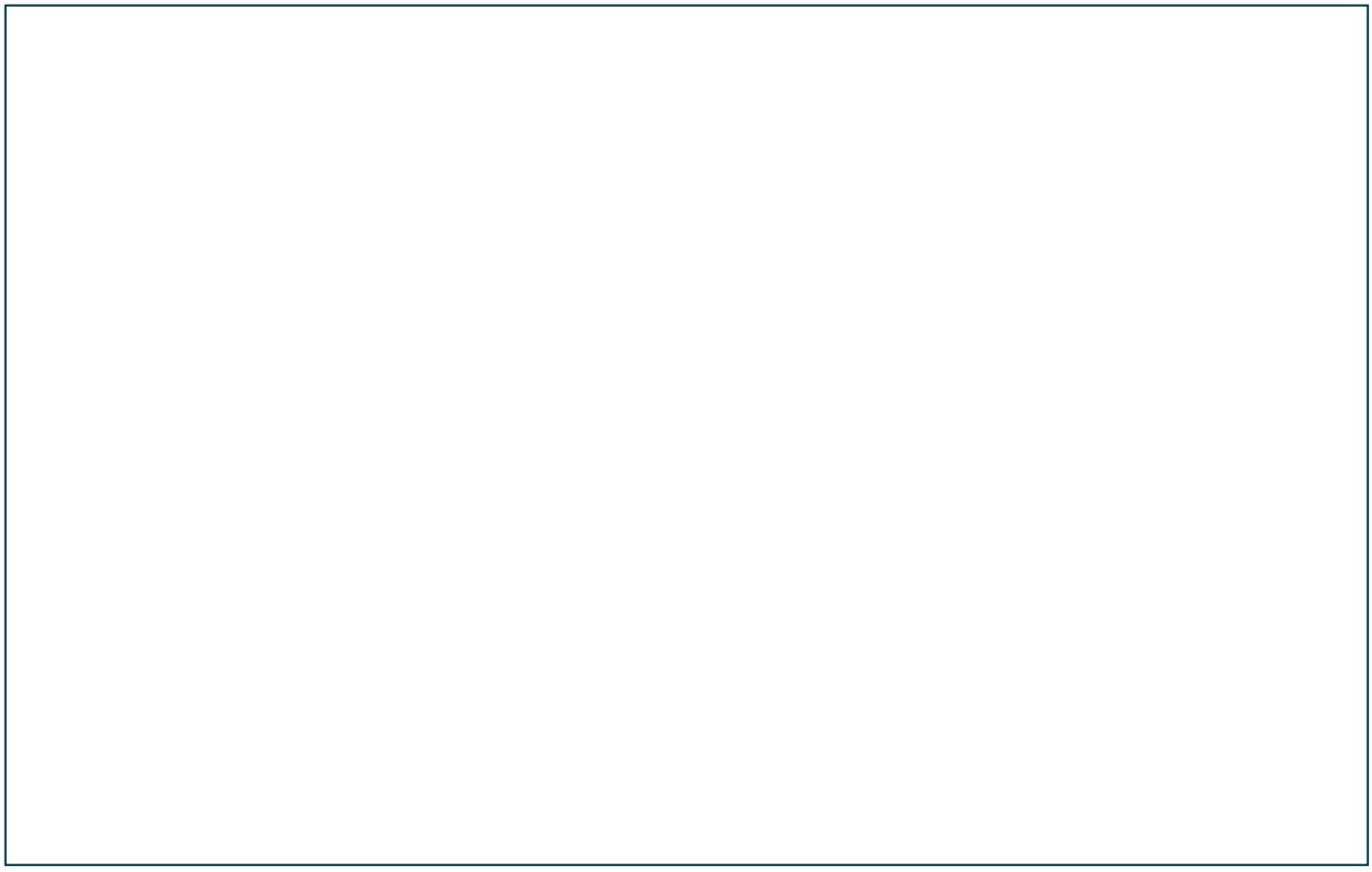
- Everything else computed on demand
- In example, these are:
  - → Original list of products
  - → Search text user enters
  - → Value of checkbox
  - → Filtered list of products

Props is data passed in as an HTML attribute when the component is created.

To figure out state, ask:

1. Is it passed in from a parent via props? Probably isn't state
2. Does it remain unchanged over time? Probably isn't state
3. Can you compute it based on any other state or props in the component? Isn't state

113

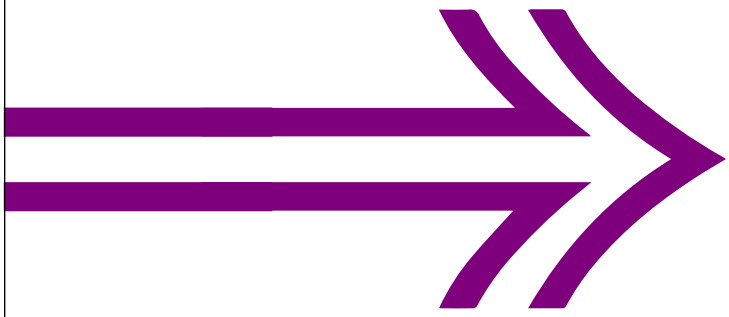


3. Identify minimal (complete) representation of UI state

In the Filterable Product Table example:

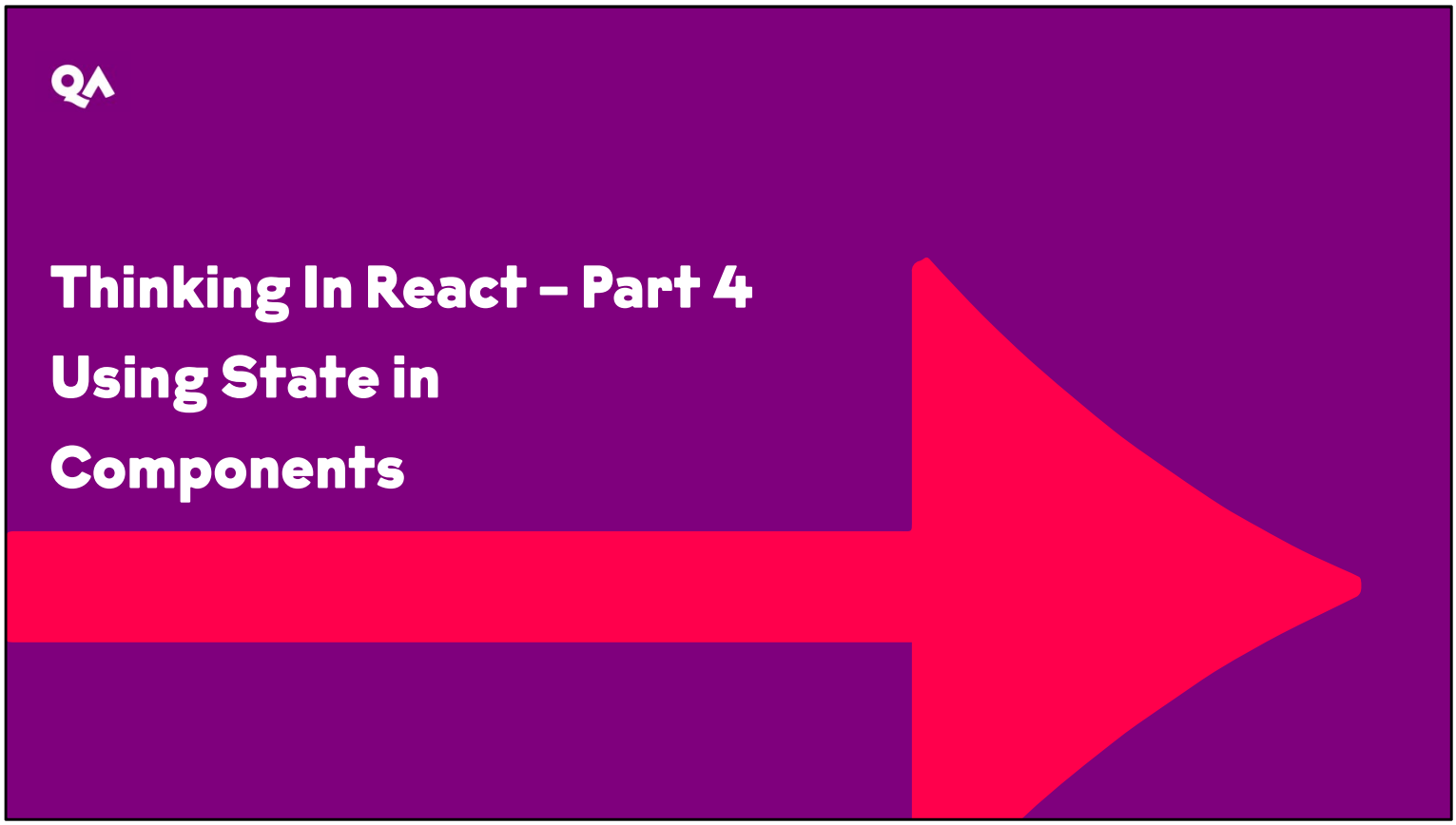
- *Original list of products* – passed in as props and does not changes over time – **not state**
- *Search text user enters* – **state**
- *Value of checkbox* – **state**
- *Filtered list of products* – can be computed by combining original list of object, search text and checkbox value – **not state**

QuickLab 11 – Thinking In React Part 3 – Identifying State



In this QuickLab, you will:

- Identify what state is needed for the ToDo application


Thinking In React – Part 4
Using State in
Components

4. Identify where state should live

Involves identifying which component mutates or owns the state.

- React all about one-way data flow down component hierarchy
- May not be immediately clear which component should own state

To work out where state should live:

1. Identify every component that renders something based on state
2. Find common owner component
3. Either common component, or component even higher up, should own state
4. If no component makes sense, create new component to hold state and add it into the hierarchy above the common owner component

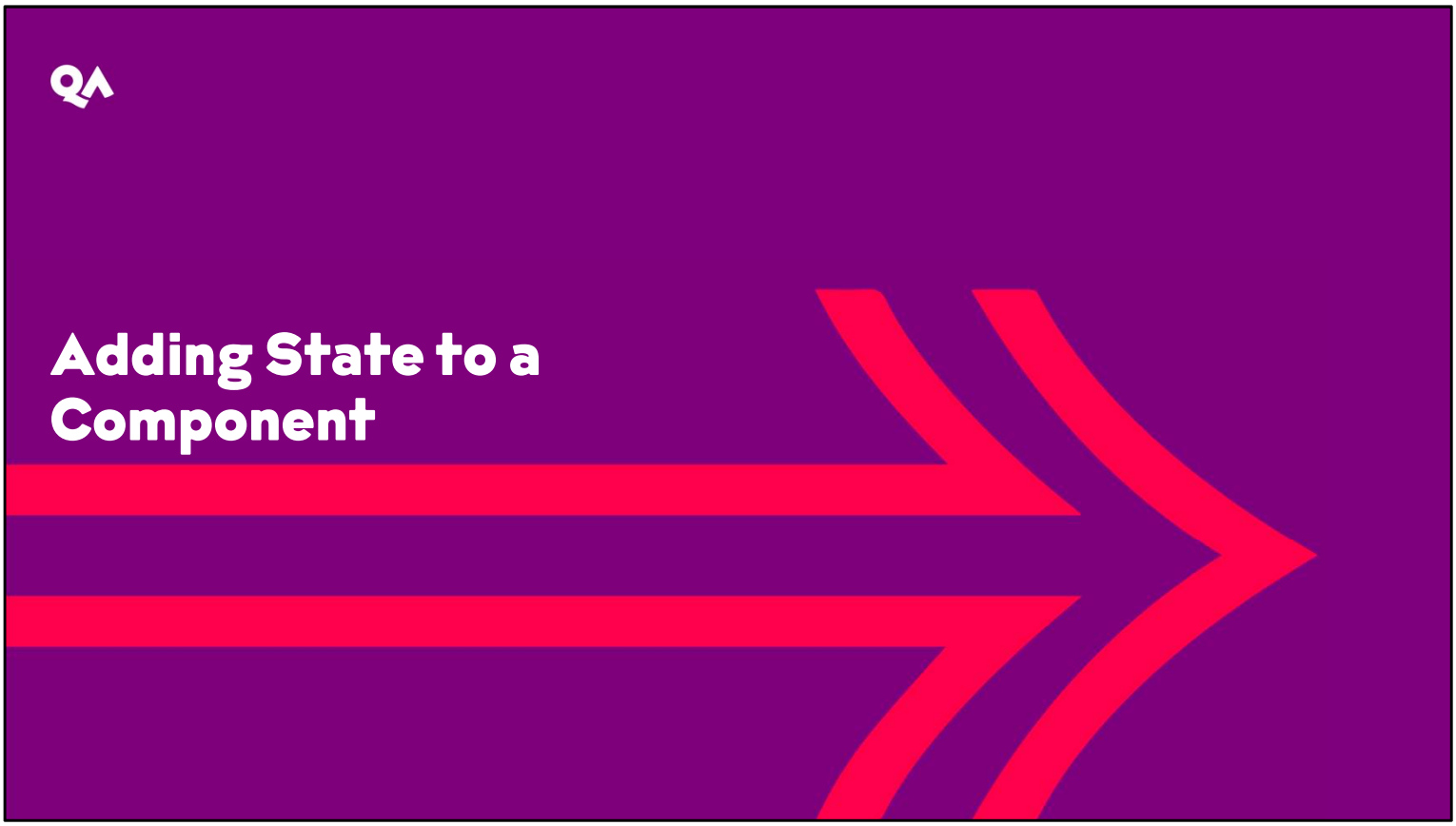
For the example:

- **`ProductTable`** needs to filter product list based on state and **`SearchBar`** needs to display search text and checked state
- Common owner component is **`FilterableProductTable`**
- Conceptually makes sense for filter text and checked value to live in **`FilterableProductTable`**

117

QuickLab 12 – Thinking In React Part 4 – Identifying Where State Should Live

- In this QuickLab, you will:
- Identify which components state should be placed in for the ToDo application


Adding State to a Component

State in function components

Previous *to React v16.8* **state** was not allowed in **Function** components.

Hooks were introduced that allows **Function** components to have **state**.

`{ useState }` needs to be imported from React to allow this functionality

Declaring **state** needs the *destructured* setting of an *array* containing the *state name* and a *function to update it* assigned to a call to `useState` with an *initial value*

- On the initial render, the initial value passed will be used

```
// Abstract
const [myState, setMyState] =
  useState(initialMyState);


// Component example
import React, { useState } from 'react';
const App = () => {
  const [count, setCount] = useState(0);
  // count initially set to 0
  return(
    <p>Count is: {count}</p>
  );
};
export default App;
// Would display: Count is: 0
```

120

Destructuring allows the unpacking of values from arrays or properties in objects into distinct variables

```
let a, b, rest;
[a, b] = [10, 20];

console.log(a);
// expected output: 10

console.log(b);
// expected output: 20

[a, b, ...rest] = [10, 20, 30, 40, 50];

console.log(rest);
// expected output: [30,40,50]
```

Updating of state in function components

State often updated due to occurrence of an event.

- E.g. a click or a change

The ***setState*** function can be called with the new value of **state**.

- Can be called as part of an *arrow function* used for the *event handler*
- Alternatively can be called as part of a larger handling function if required

Calling of ***setState*** function **triggers re-render** of *any Component* dependent on the **state**.

- Including those that use it as a **prop**

```
import React, { useState } from 'react';
const App = () => {
  const [count, setCount] = useState(0);
  // count initially set to 0

  return(
    <>
      <p>Count is: {count}</p>

      <button onClick={() =>
         setCount(count + 1)}>Add</button>
      // Adds 1 to count every button click
      // and triggers re-render of component
    </>
  );
};
export default App;

// Would display:
//     'Count is: 0' initially
//     'Count is: 3' after 3 clicks
```

121

State in class components

Before *React v16.8*, if state was required a **Class component had** to be used.

State is initialised in the **constructor**.

setState is called to update state.

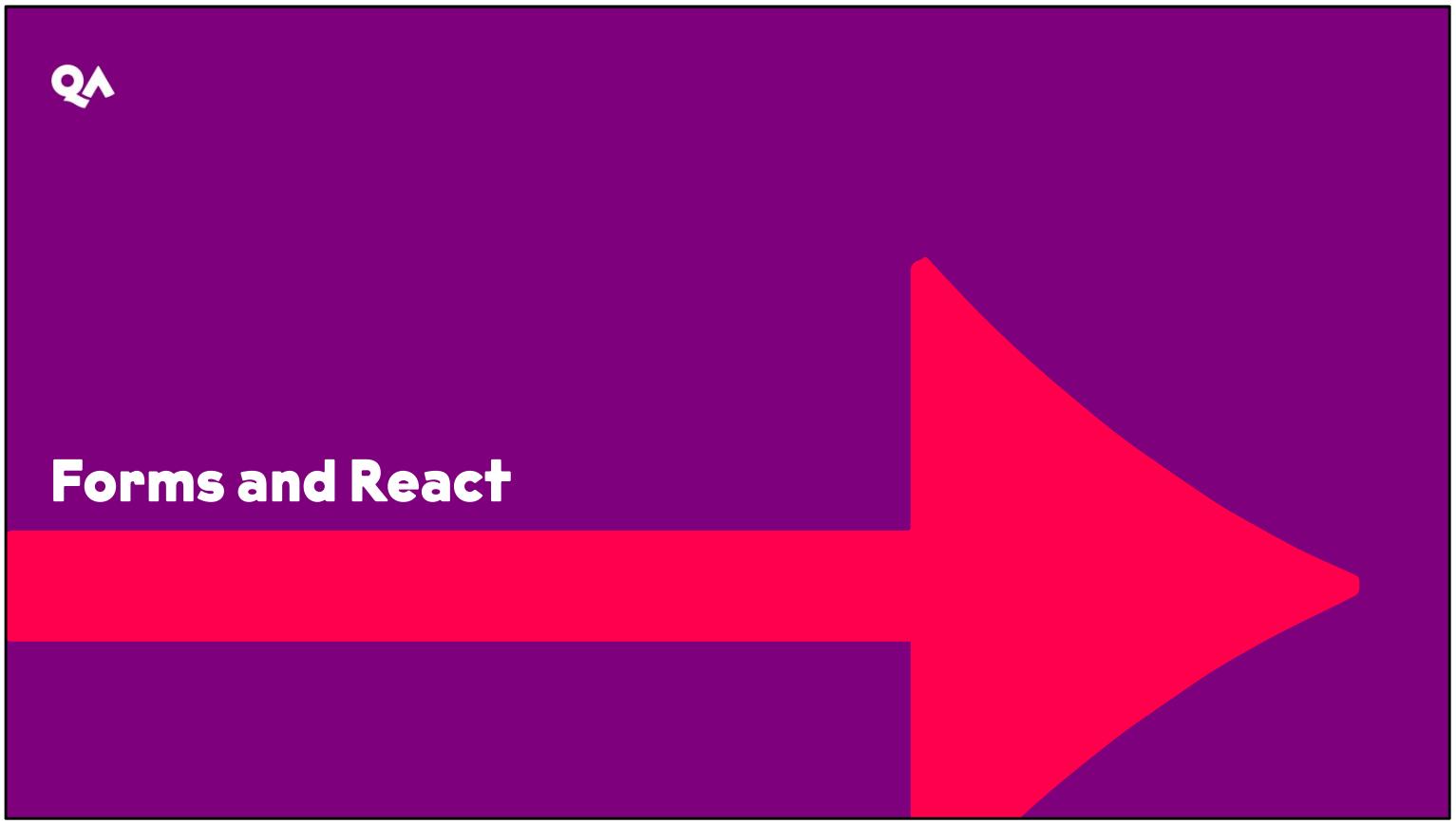
- Only properties and values to update need to be included
- Calling of **setState** triggers a re-render of any component dependent on **state**

→ Including those that use the state as a **prop**

```
import React, { Component } from `react`;
class App extends Component {
  constructor(props) {
    super(props);
    this.state = {count : 0};
  }
  render() {
    return (
      <p>Count is: {this.state.count}</p>
      <button onClick={() =>
         this.setState(
           {
             count: this.state.count + 1
           }
         )
      }>
         Add
      </button>
    );
  }
}
export default App;
```

122

Class Components have been used in React since its inception. Facebook have no plans to deprecate Class Components so you may see this syntax in any applications that you work on that have been around for a while. Class Components need not be converted to Function components – but all new components should be built using Functions and Hooks.


Forms and React

Forms and React

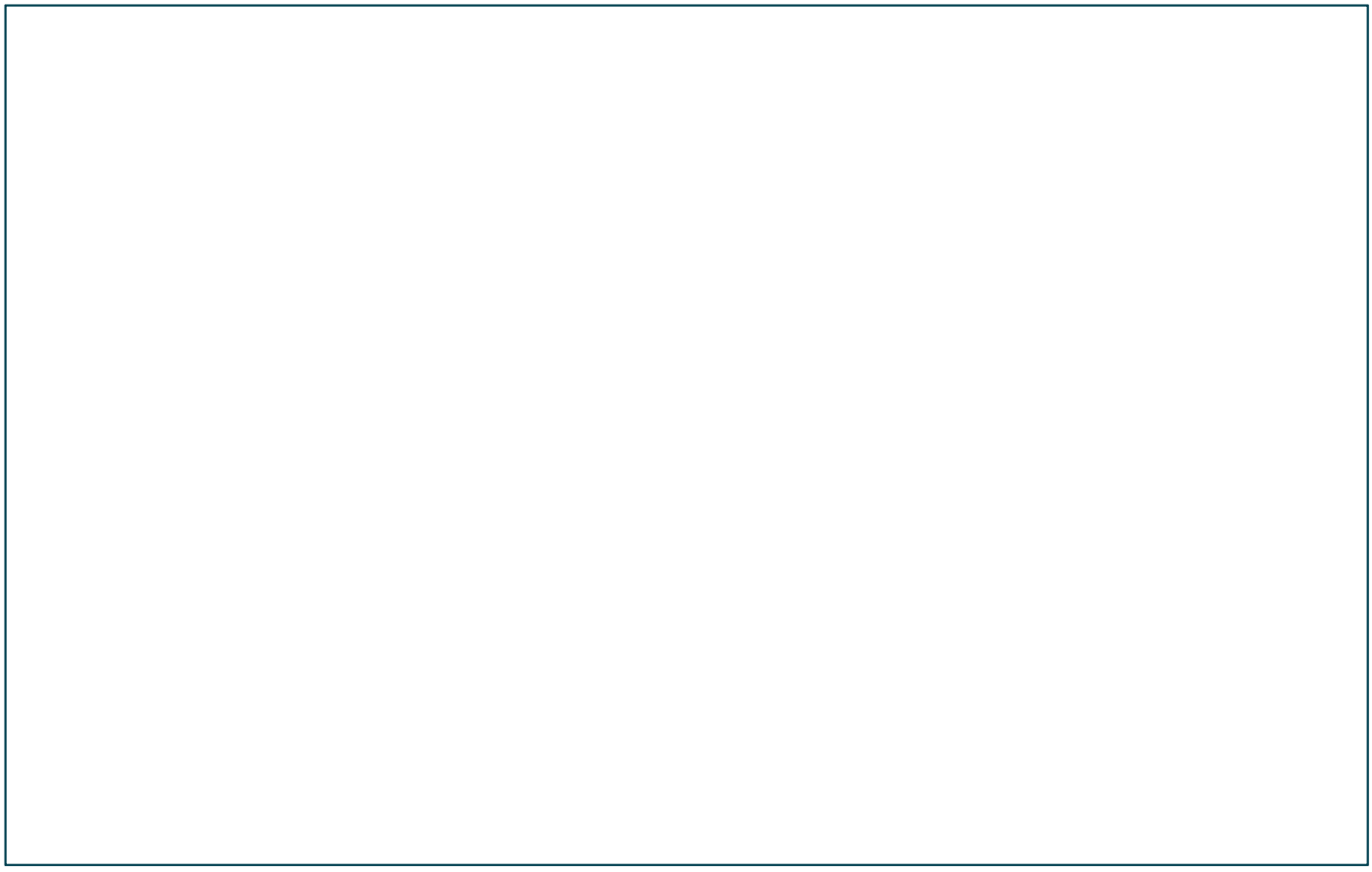
Forms inherently keep some internal state and therefore React has to work differently with them.

- A standard form with a submit button would work in React out of the box (as shown below)
- Usually want to have access to the form values that have been submitted
- Controlled components are the recommended way to achieve this

Events trigger the updating of state in React components to keep everything in sync.

```
<form>
  <label>
          Name:
    <input type="text" name="name" />
  </label>
  <input type="submit" value="Submit" />
</form>
```

124



Since we've mentioned events...

Method to handle events with React elements is very similar to handling events on DOM elements.

Syntactic differences:

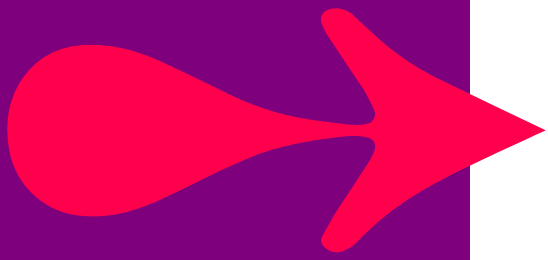
- React events are **camelCased** rather than lowercase
- With JSX you pass a function as the event handler rather than a string
- Cannot return false to prevent default behaviour – must use `preventDefault()`
- Don't need to call `addEventListener()` in React, just provide listener when the element is initially rendered
- Common pattern for ES2015 classes is for the event handler to be a method of the class or an arrow function set in the constructor – can also be arrow functions in a Function component
- *If using a method in a class it is necessary to **bind** the method, either in the constructor or in the callback*

Function components do not need worry about binding.

The React Documentation explain the meaning of this as follows:

You have to be careful about the meaning of this in JSX callbacks. In JavaScript, class methods are not bound by default. If you forget to bind this.handleClick and pass it to onClick, this will be undefined when the function is actually called.
This is not React-specific behavior; it is a part of how functions work in JavaScript. Generally, if you refer to a method without () after it, such as onClick={this.handleClick}, you should bind that method.
If calling bind annoys you, there are two ways you can get around this. If you are using the experimental property initialiser syntax, you can use property initializers to correctly bind callbacks.
If you aren't using property initializer syntax, you can use an arrow function in the callback.
The problem with this syntax is that a different callback is created each time the LoggingButton renders. In most cases, this is fine. However, if this callback is passed as a prop to lower components, those components might do an extra re-rendering. We generally recommend binding in the constructor to avoid this sort of performance problem.

EXAMPLES OF EVENTS IN A FUNCTION COMPONENT



```
import React, { useState } from 'react';
const App = () => {
  const [count, setCount] = useState(0);
  const [currClass, setCurrClass] = useState(`green`);
  const changeClass = e => {
    if(e.type === `mouseenter`) setCurrClass(`red`);
    if(e.type === `mouseleave`) setCurrClass(`blue`);
  }
  return(
    <>
      <p>Count is: {count}</p>
      <button onClick={() => setCount(count + 1)}>+</button>
      // onClick event triggers callback and a re-render
      <p onMouseEnter={e => changeClass(e)}
         onMouseLeave={e => changeClass(e)}
         className={currClass}>
         Move the mouse over this text to change colour
      <p>
      // onMouseEnter and onMouseLeave calls the changeClass
      // function, passing the event as an argument and
      // setting the value of currClass causing a re-render
    </>
  );
};
export default App;
```

Note how the callback can either be defined as the action – in a simple situation where a change in a state value is required, or as a call to a more complex method.

Collecting form data

Form elements usually maintain their own state and it is changed through user input.

React component's mutable state is kept in the state property and can only be changed by calling its ***setState*** method.

The form shown here would work but it could be seen as an **'uncontrolled'** component.

- React does not explicitly control the value displayed in the input box
- To make it a controlled component, set the **`value`** attribute to the *associated value in state*

```
value={name}
```

```
import React, { useState } from 'react';
const Form = () => {
  const [name, setName] = useState(``);
  return(
    <form>
      <input
        type="text"
        name="name"
        onChange={event =>
          setName(event.target.value)} />
    </form>
  );
};
export default Form;
// Typing in the Name input box updates
// the name state
```

Collecting form data

Form elements usually maintain their own state and it is changed through user input.

React component's mutable state is kept in the state property and can only be changed by calling its ***setState*** method.

The form shown here would work now with a **'controlled'** component.

- React ***now explicitly controls*** the value displayed in the input box
- The **`value`** attribute to the *associated value in state*

`value={name}`

```
import React, { useState } from 'react';
const Form = () => {
  const [name, setName] = useState(``);
  return(
    <form>
      <input
        type="text"
        name="name"
        value={name}
        onChange={event =>
          setName(event.target.value)} />
    </form>
  );
};
export default Form;
// The output of the input is now
// controlled by React
```

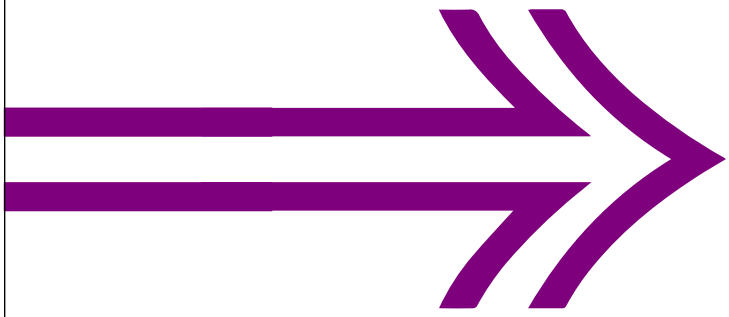
128

More on controlled and uncontrolled components can be found in the documentation here:

Controlled Components: https://reactjs.org/docs/forms.html#controlled-components

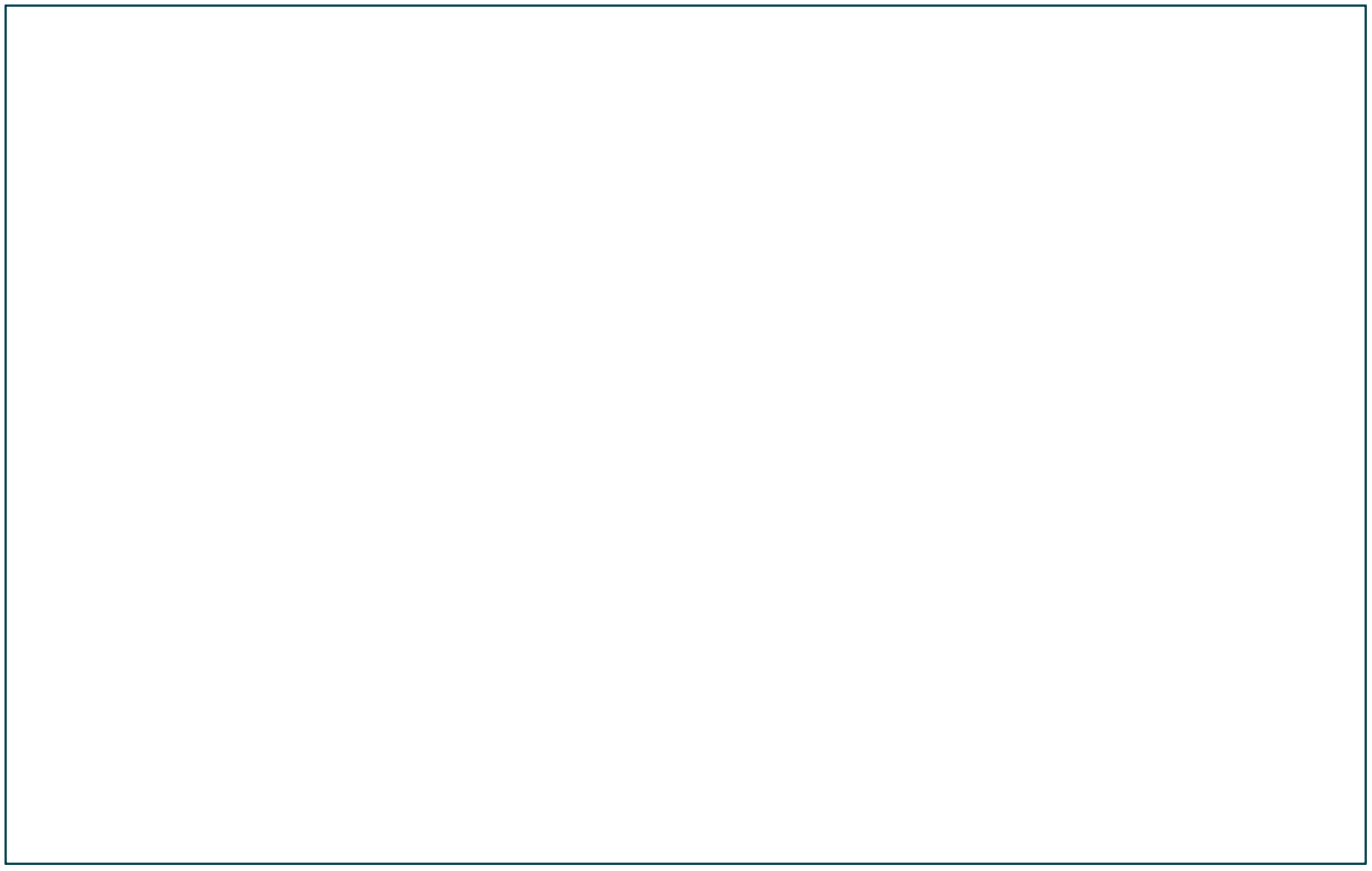
Uncontrolled Components: https://reactjs.org/docs/uncontrolled-components.html

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



- In this QuickLab, you will:
- Add state to appropriate components in the ToDo application
- Manage the state of the ToDo form
- Pay back some of Technical Debt incurred using static data
- Check the tests still pass

Performing Act(ions) in Tests



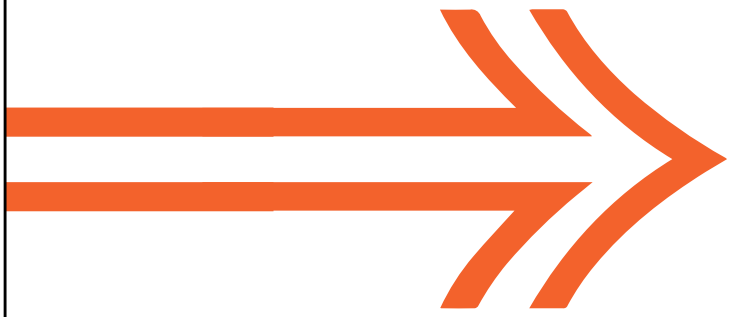
Using act within tests

The act function prepares a component for assertions.
It can wrap function calls within the component – such as triggering event handlers.
If an event is to change a value a object with a target object that sets a value has to be supplied.

```
...
  someTestInput = testInstance.findByProps({name: "testInput"});
  expect(someTestInstance.props.value).toBe(``);
  // Trigger the onChange event handler in act to update the value and check
  act(() => someTestInput.props.onChange({target: {value: `some test value`}}));
  expect(someTestInstance.props.value).toBe(`some test value`);
```

131

QuickLab 13b – Testing Events on the Form

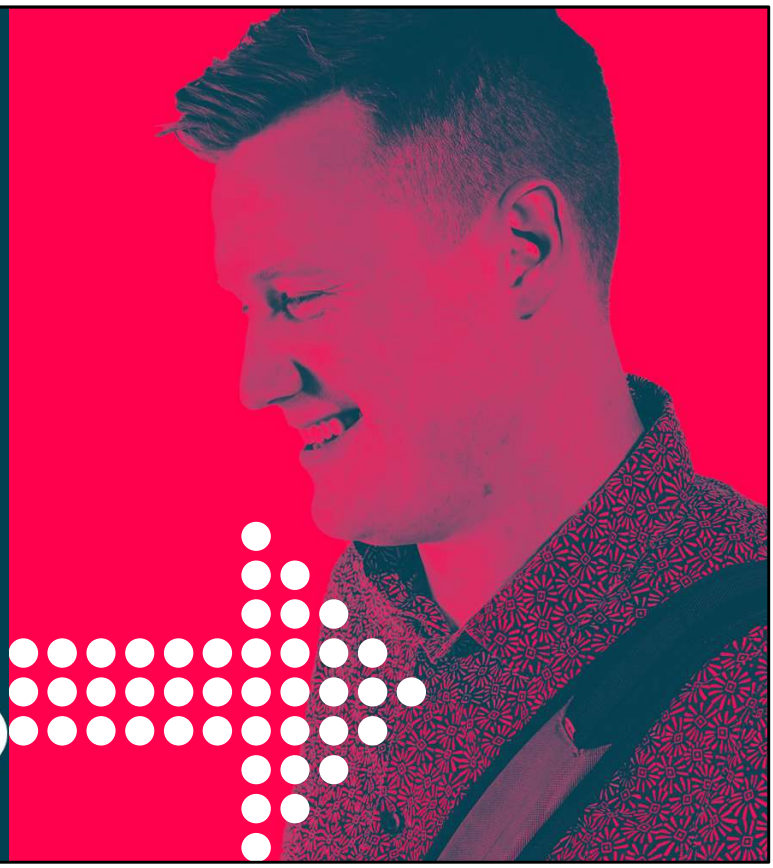


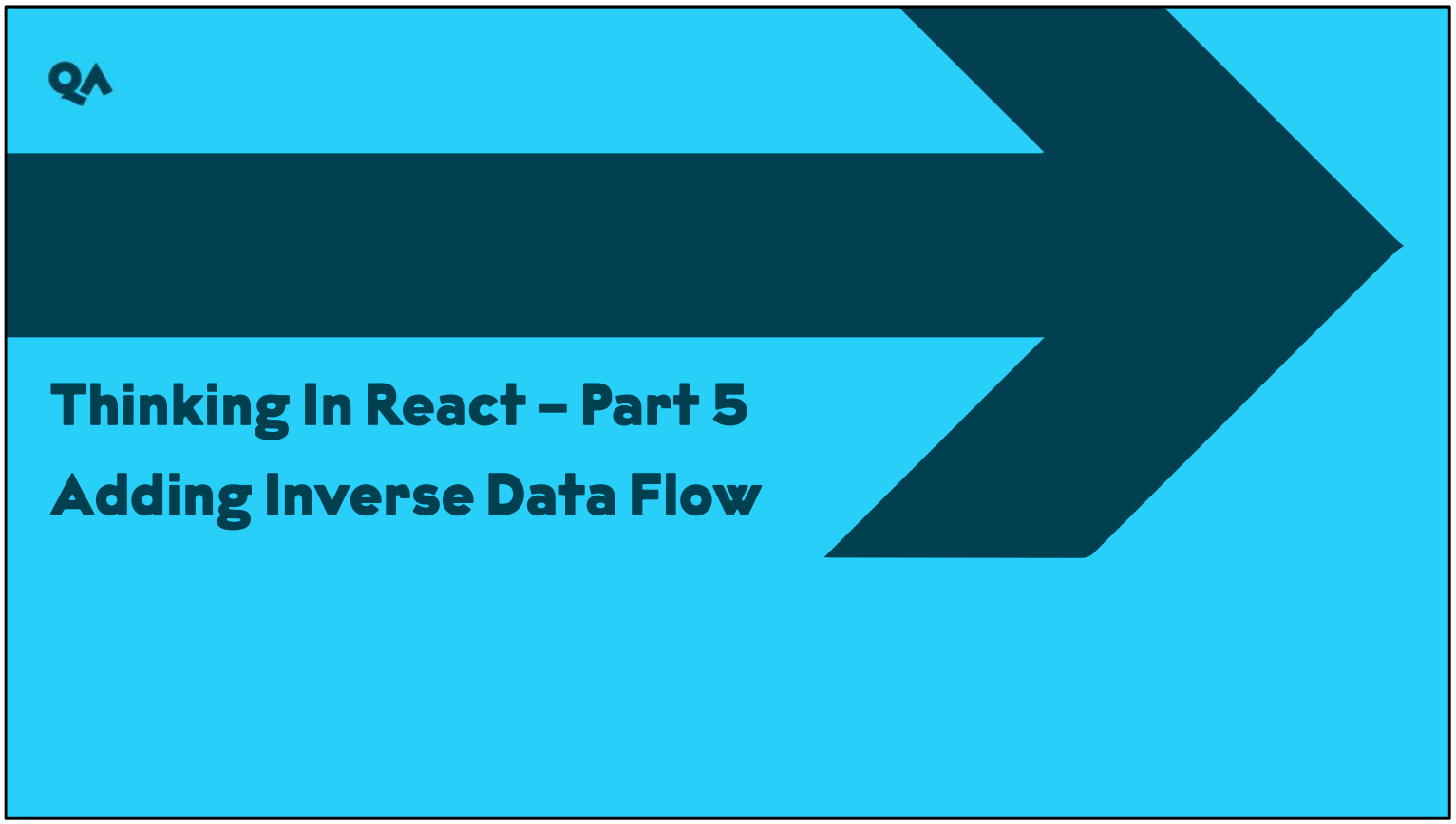
In this QuickLab, you will:

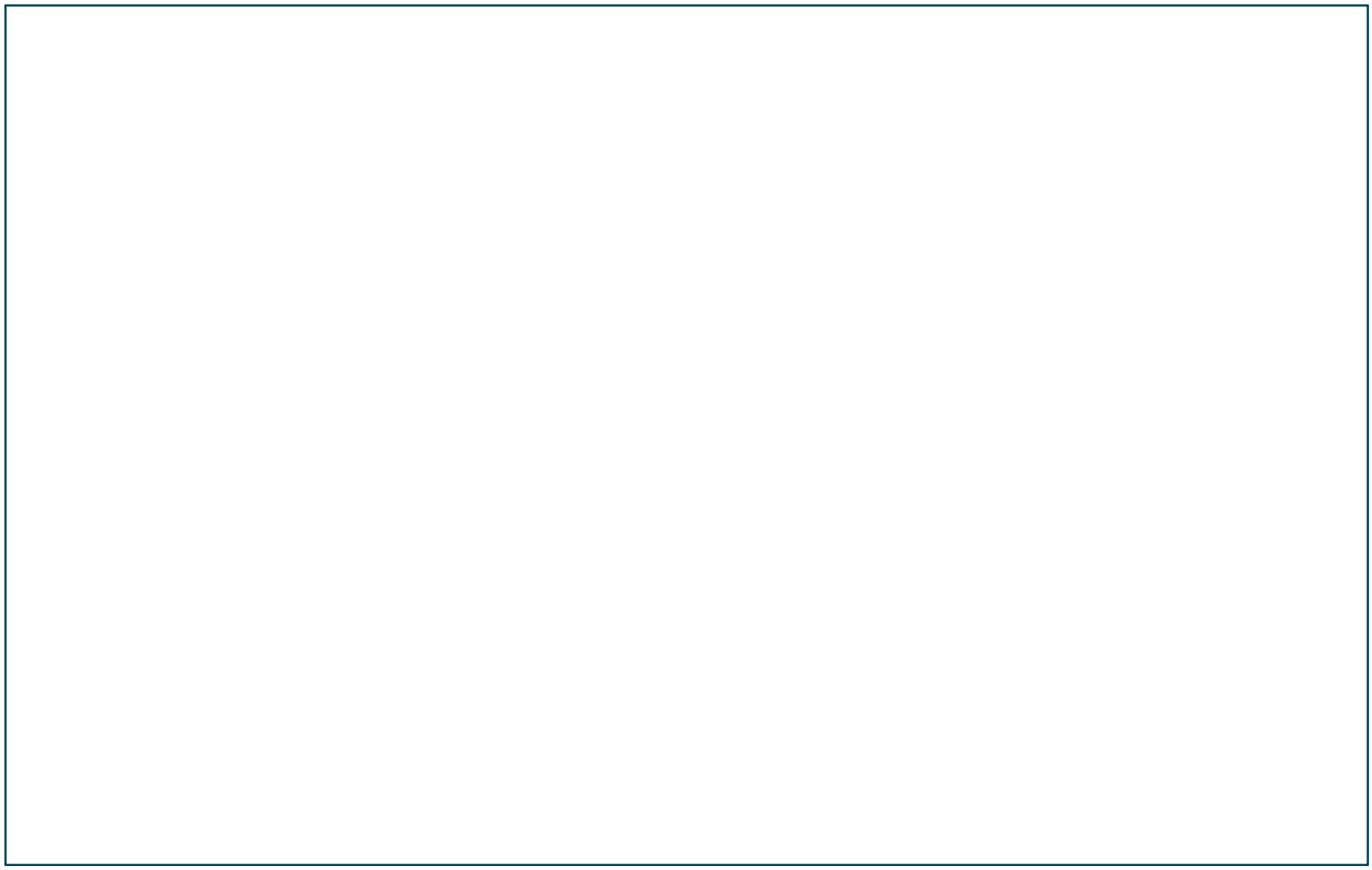
- Test that events on the form update the render correctly

Objectives

- To be able to identify state in an application and add it to the appropriate component
- To be able to work with forms




Thinking In React – Part 5
Adding Inverse Data Flow



Objectives

- To be able to use inverse data flow in applications

5. Add inverse data flow

App renders correctly as function of *props* and *state* flowing down hierarchy.

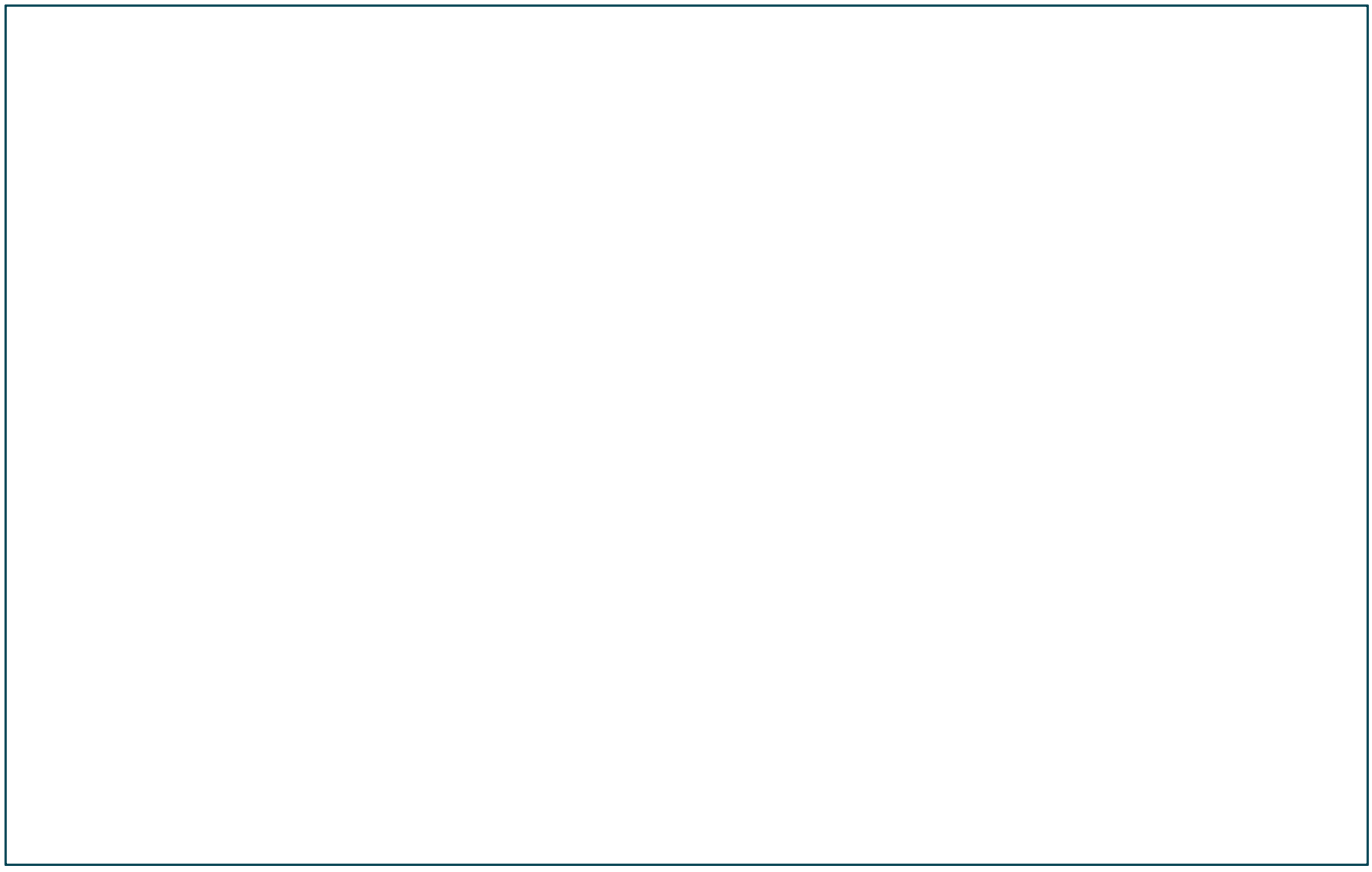
To support data flowing other way, form components deep in hierarchy need to update state in **`FilterableProductTable`**.

Want to make sure whenever user changes form, state is updated.

- **`FilterableProductTable`** needs to pass callback to **`SearchBar`**

  Fires whenever state should be updated.
- Use **`onChange`** event on inputs

  Callback will change the state and update the application UI.



Calling functions as props

```
import React, { useState } from 'react';
const Form = props => {
  const [name, setName] = useState(``);
  const handleSubmit = event => {
    event.preventDefault();
    props.handleSubmitInParent(name); // Callback supplied as
    setName(``);                      // prop in parent component
  }
  return(
    <form onSubmit={handleSubmit}>
      <input
        type="text" name="name" value={name}
        onChange={event =>
          setName(event.target.value)}
      />
      <input type=submit value="Submit" />
    </form>
  );
};
export default Form;
// Clicking the submit button calls handleSubmitInParent, passing
// it the state value of name
```

To facilitate passing data from a child component to a parent, the parent can supply a **callback** to the child as props.

Child component can use the **callback**, supplying any data required.

- Particularly useful when submitting forms and updating application state in ancestor components

137

Calling functions as props

```
import React, { useState } from `react`;
import Form from `./Form`;

const App = () => {
  const [names, setNames] = useState([]);

  const addNameToList = name => {
    const updatedNames = [...names, name];
    setNames(updatedNames);
  };

  const nameList = names.map((name, index) =>
    <li key={index}>{name}</li>);

  return(
    <>
      <h1>Hello people!</h1>
      <ul>{nameList}</ul>
      <h2>Add Name:</h2>
      <Form handleSubmitInParent={addNameToList} />
  );
};
export default Form;
```

When the **Form** component calls the **handleSubmitInParent** callback it actually calls **addNameToList** from the **App** component supplying the **name** argument.

This principle can be used to have a wrapping form component around input items, storing the state of the form in a single place.

138

QuickLab 14a – Thinking In React Part 5 – Inverse Data Flow and Form Submission

In this QuickLab, you will:

- Pass data from the ToDo form to its parent up the component hierarchy
- Use the data to update the application state
- Pay back some of Technical Debt incurred using static data

MOCKING FUNCTIONS IN JEST

Useful for when functions are passed in as props to a component.

Allows spying on function calls to ensure that things happen.

As simple as a declaration:

```
const functionToMock = jest.fn();
```

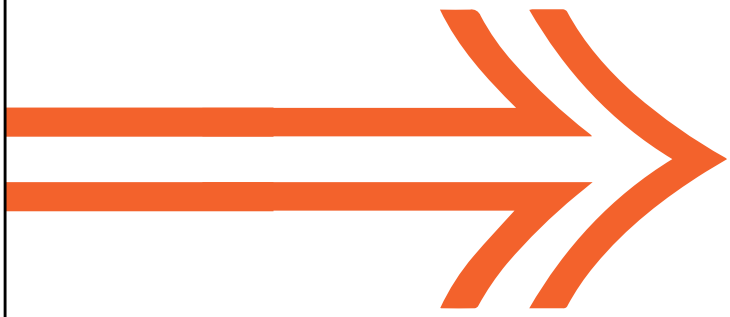
Can then be used as a prop when creating a component:

```
const testRenderer = create(
  <MyComponent functionAsProp={functionToMock} />
);
```

Asserting can then use the function matchers to check calls:

```
expect(functionToMock).toHaveBeenCalledTimes(1);

expect(functionToMock)
  .toHaveBeenCalledWith(TestObject);
```

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

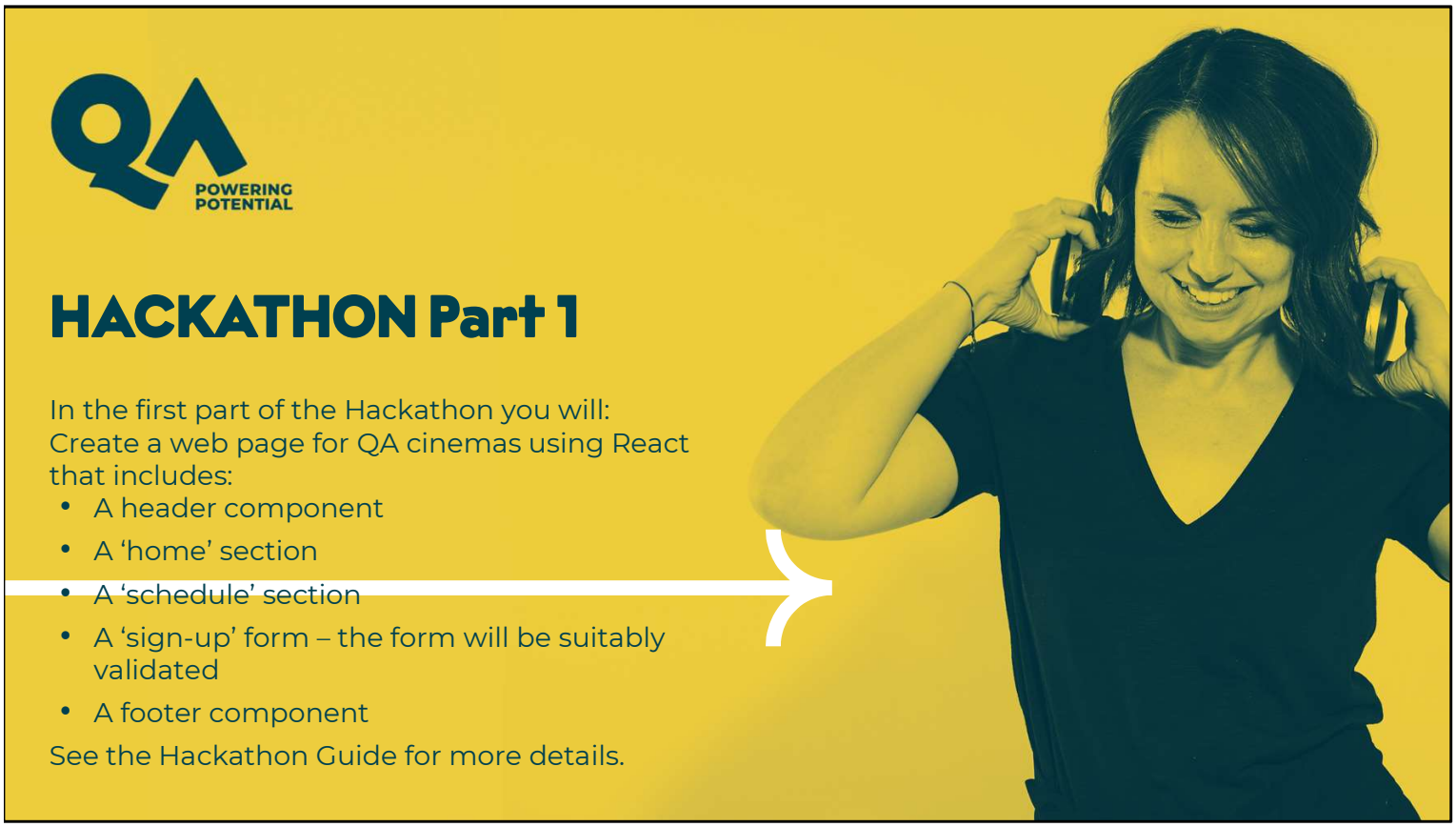


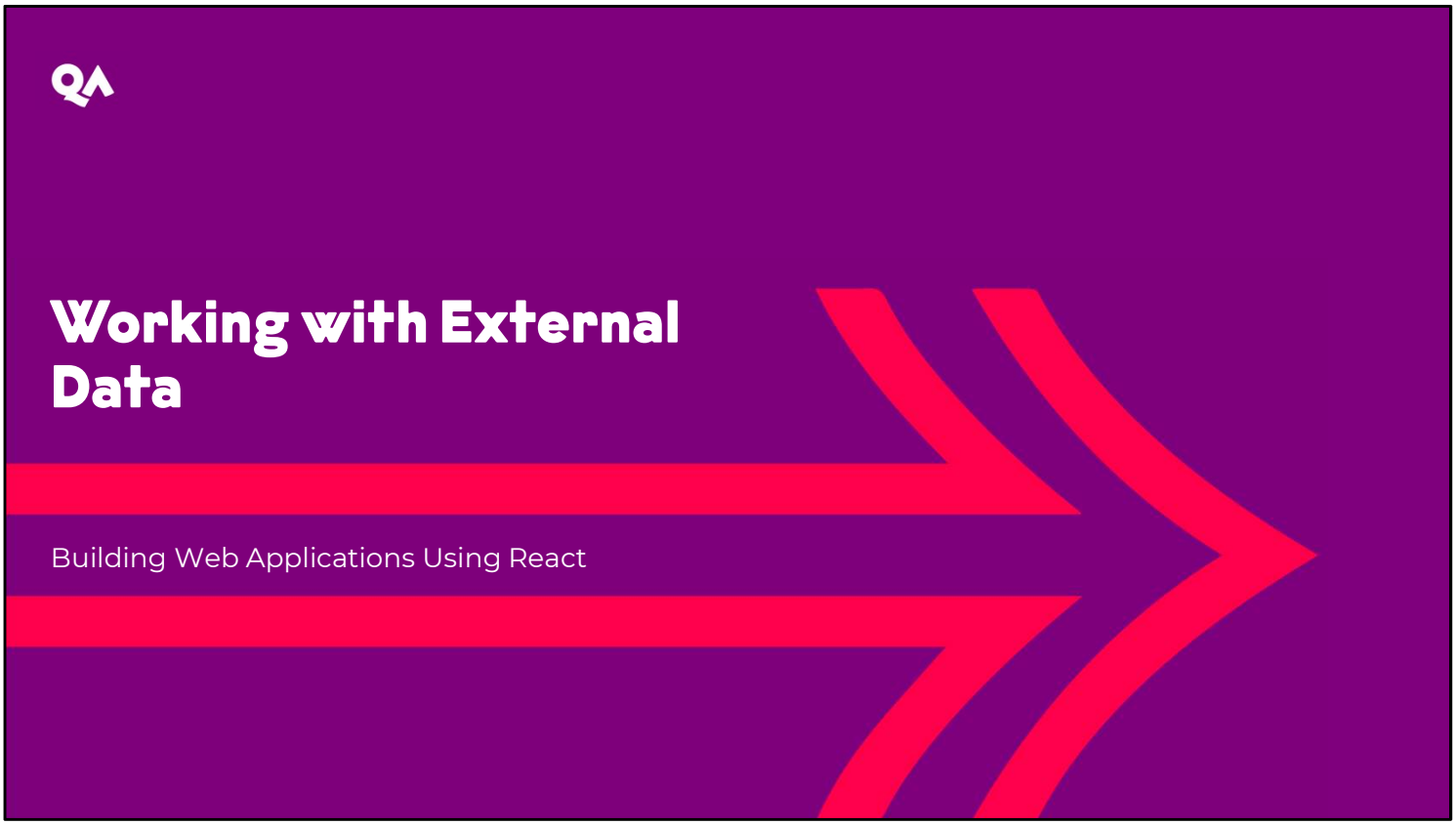
In this QuickLab, you will:

- Test that the form submission performs the expected actions

Objectives

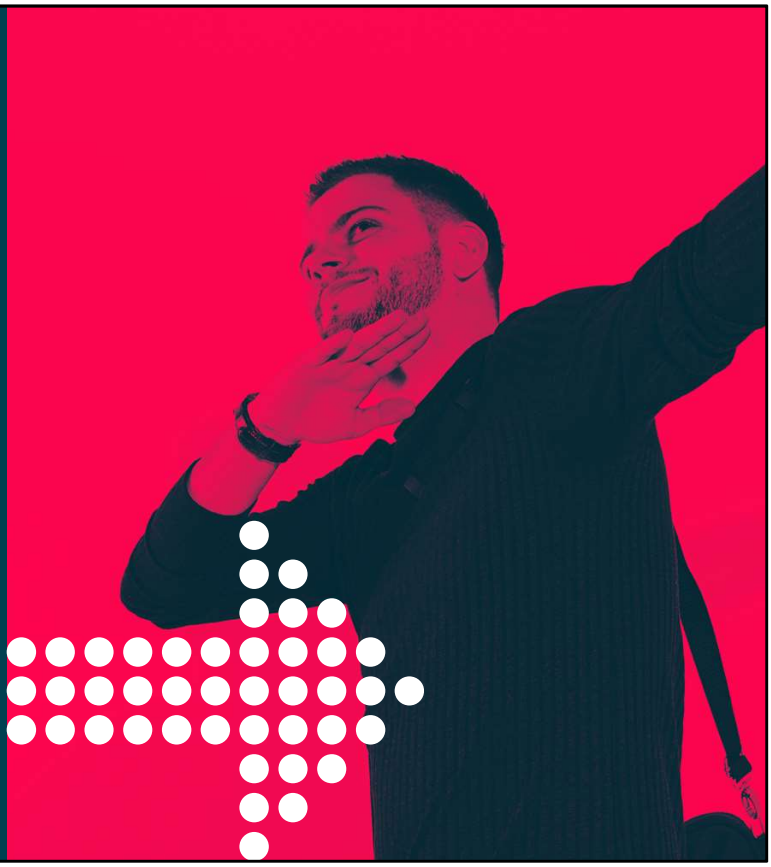
- To be able to use inverse data flow in applications
- To be able to test that functions have been called

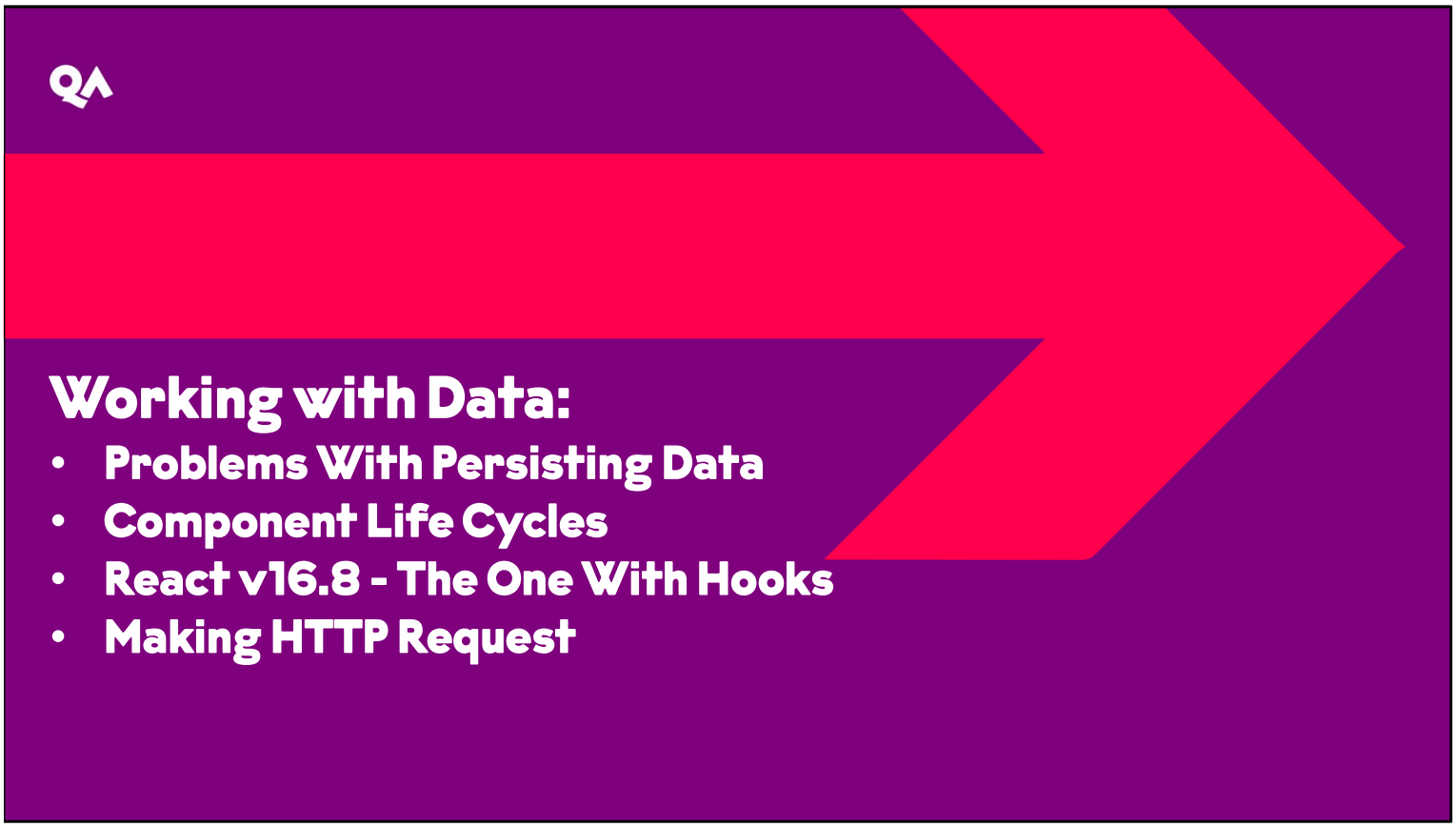

POWERING
POTENTIAL
HACKATHON Part 1
In the first part of the Hackathon you will:
Create a web page for QA cinemas using React that includes:
• A header component
• A 'home' section
• A 'schedule' section
• A 'sign-up' form – the form will be suitably validated
• A footer component
See the Hackathon Guide for more details.


Working with External Data
Building Web Applications Using React

Objectives

- To understand component life cycles
- To be able to understand different hooks available in React
- To be able to use the useEffect hook
- To understand what HTTP Verbs are used in RESTful services
- To be able to set up a mock RESTful service
- To be able to get data from an external service
- To be able to send data to an external service




Working with Data:
• Problems With Persisting Data
• Component Life Cycles
• React v16.8 - The One With Hooks
• Making HTTP Request

Problems with persisting data

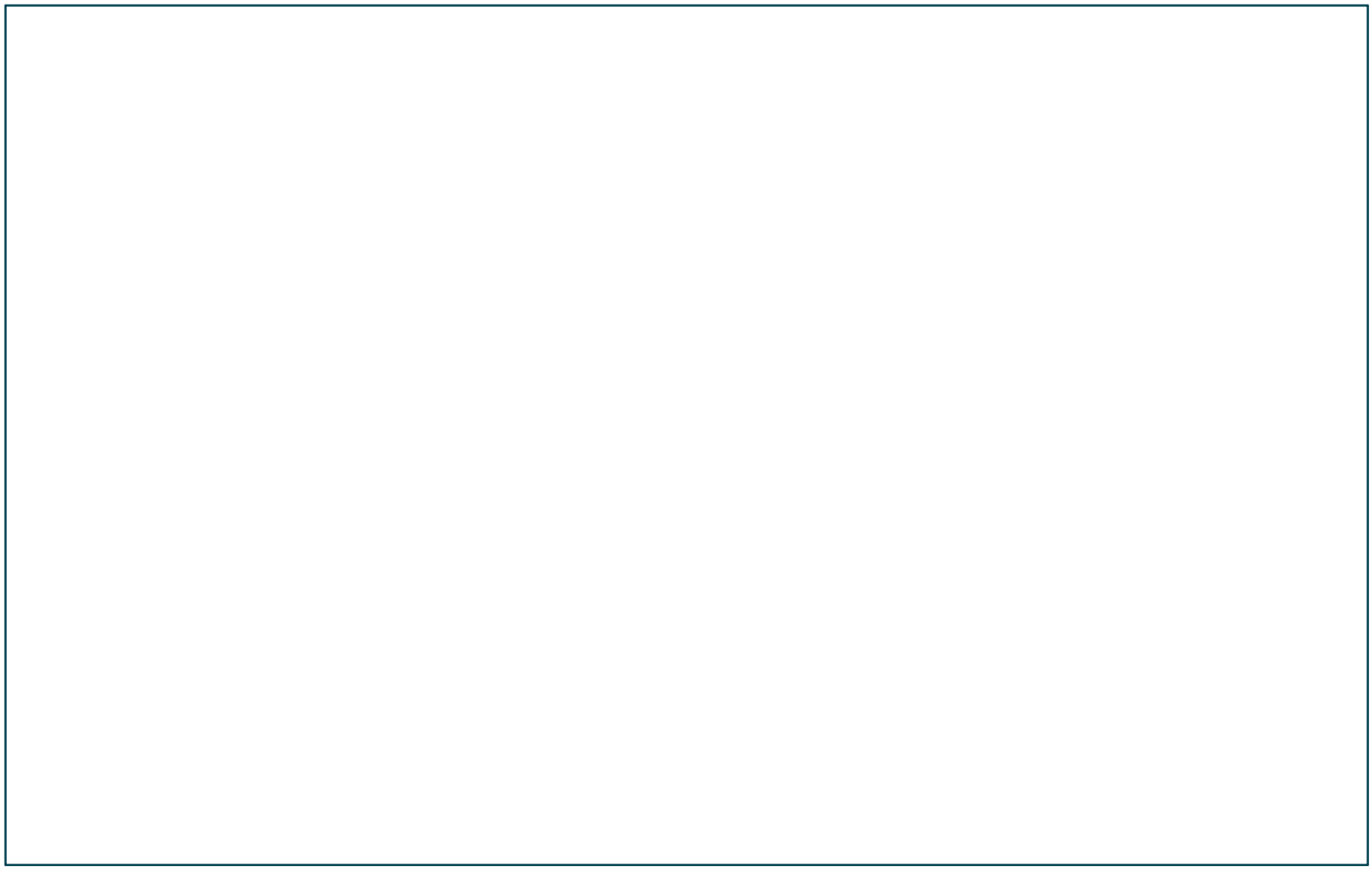
Refreshing the application in the browser will destroy any data created during its life.
Persisting data requires a temporary or permanent store.

- **LocalStorage** could be leveraged at a client level to store data
  Can be removed by the client.
  Does not allow for server level data to be persisted.

- Usual for applications to request from or send to some form of data service
  Make HTTP requests for data.
  Handle data when it is received.
  Causes chicken and egg situation for requesting and rendering.

  - Need to understand component life cycles and when hooks run

147



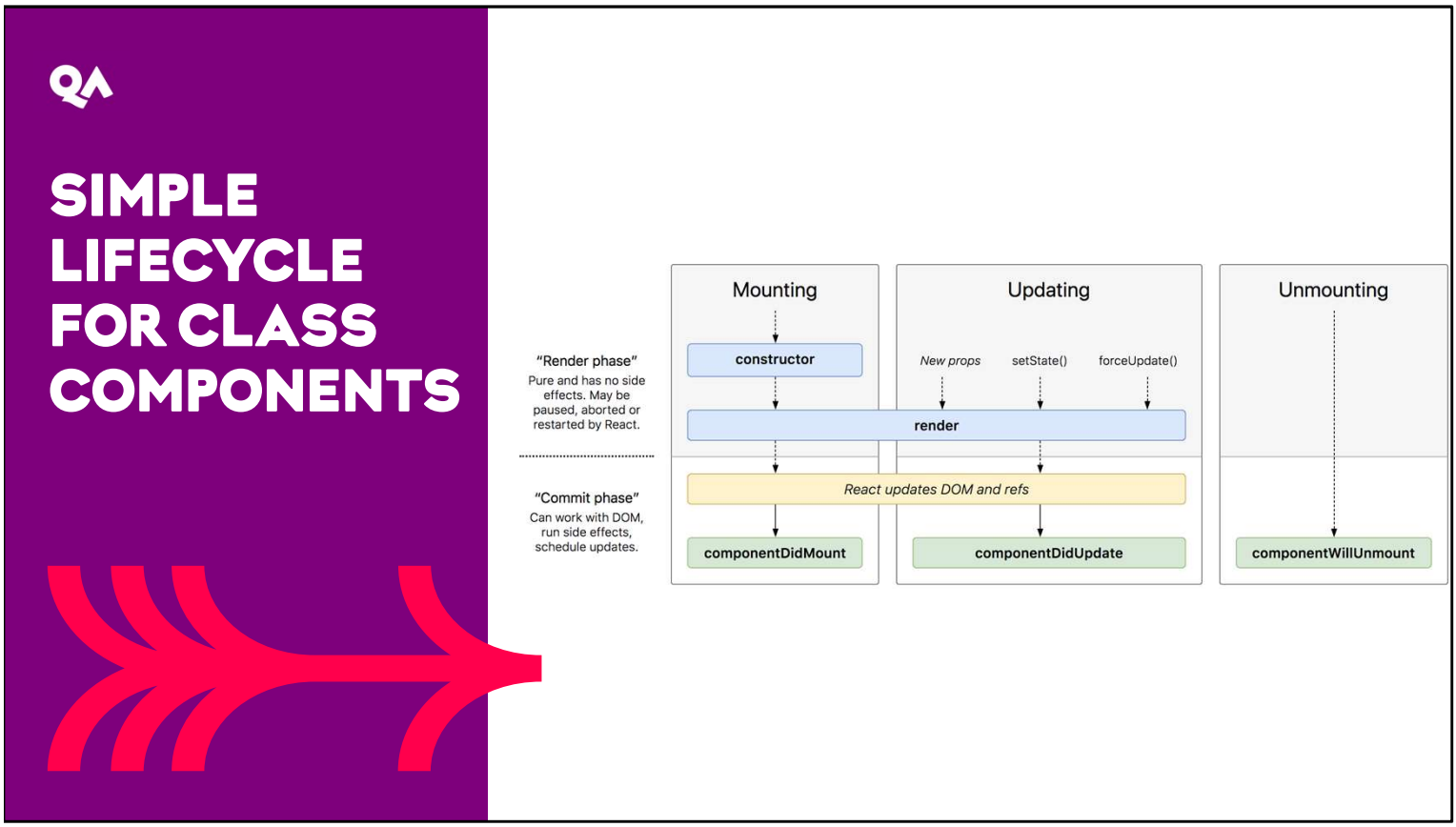


Image by Dan Abramov: https://twitter.com/dan_abramov/

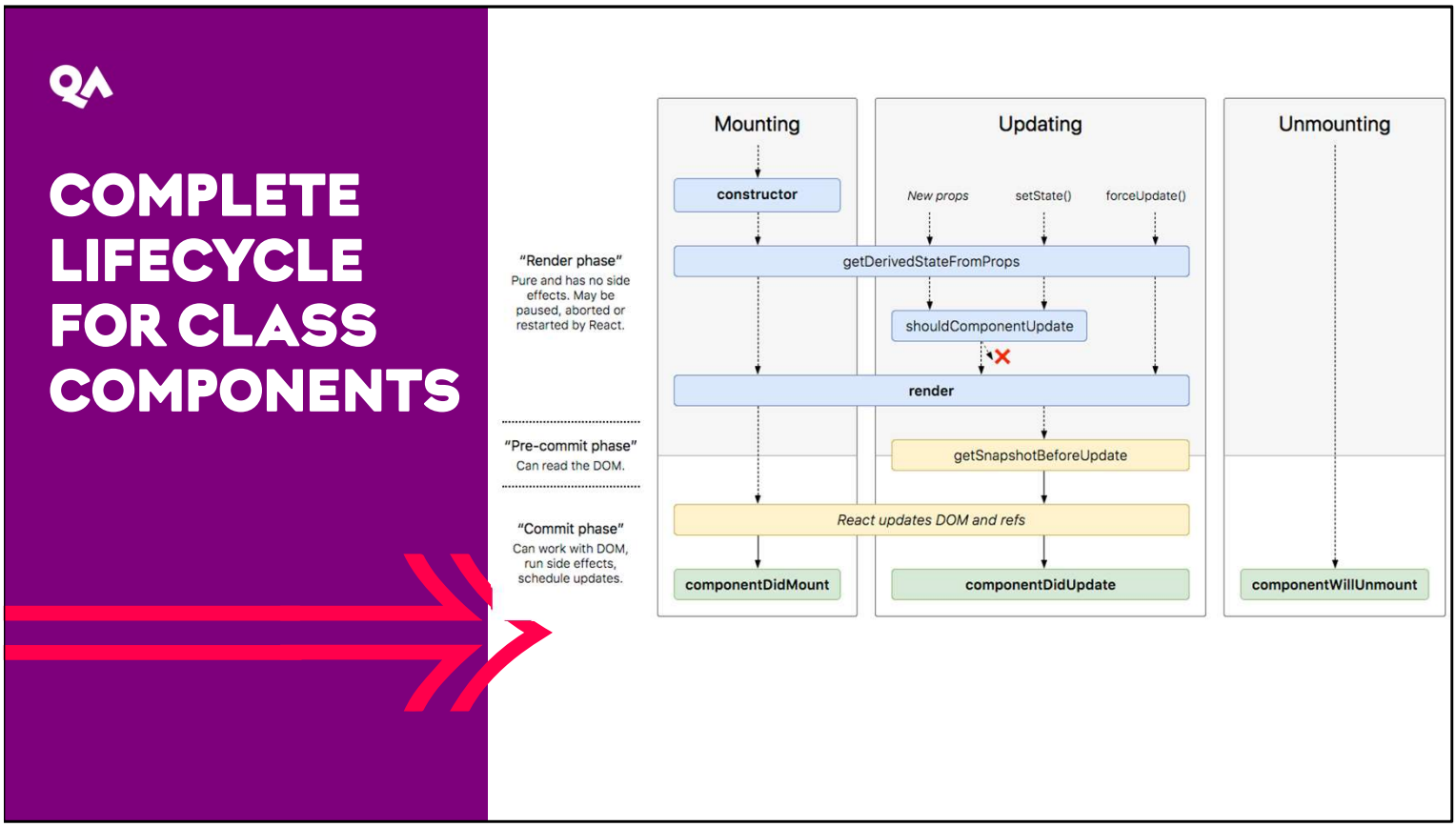
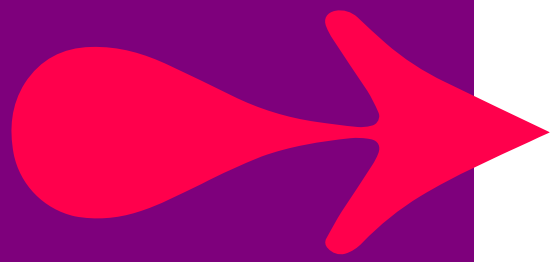


Image by Dan Abramov:
https://twitter.com/dan_abramov/status/981712092611989509/photo/1

LIFECYCLE OF A FUNCTION COMPONENT WITH HOOKS



React Hook Flow Diagram

v1.3.1 github.com/donavon/hook-flow

Mount	Update	Unmount
Run lazy initializers		
	Render	
	React updates DOM	
	Cleanup LayoutEffects	
	Run LayoutEffects	
	Browser paints screen	
	Cleanup Effects	
	Run Effects	

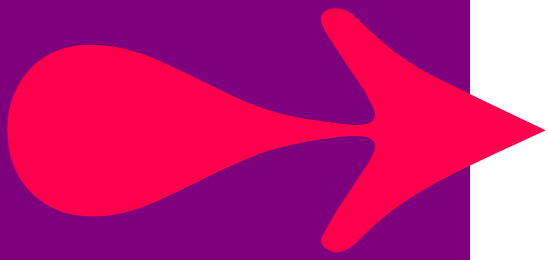
Notes:

1. Updates are caused by a parent re-render, state change, or context change.
2. Lazy initializers are functions passed to useState and useReducer.

150


React v16.8
– The One with Hooks

WHY HOOKS?



Move to promote functional components needed mechanism for state and other React features

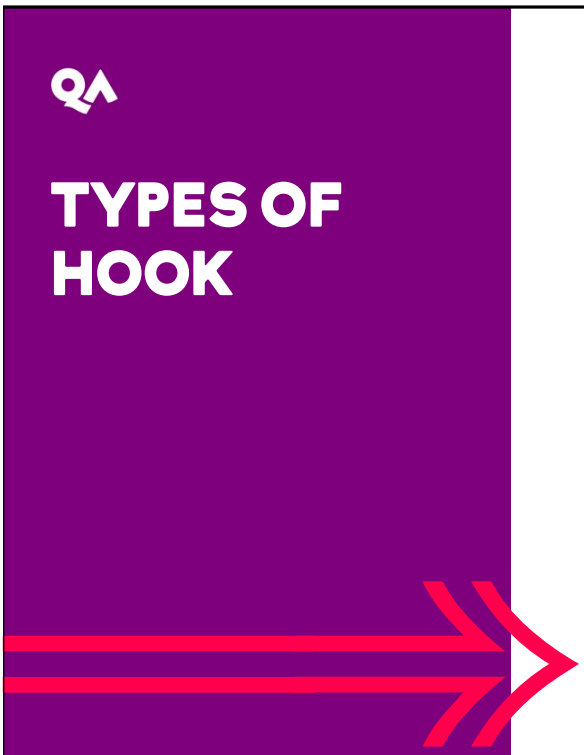
Backwards compatible.

Lets you 'hook into' React's state and life cycle features.

Solve a raft of problems encountered by the React team:

- Reuse of stateful logic between components
- Splitting of complex components into smaller functions based on relationship
- Simplifies syntax of class components to functions – which have to be transpiled anyway

No need to re-write class components to use Hooks immediately, just make new ones using them!



State Hook - revisited

Allows addition of local state to a function component.

Requires useState to be used (this is the HOOK)

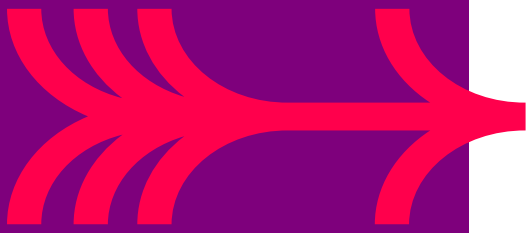
- Takes an argument of the initial state
- Returns the current state and a function to update it
  - Similar to **`this.setState`** in a class
  - Calling the setting function causes a re-render of the component

Can have more than one state hook in a component.

```
import React, { useState } from `react`;
const ExampleWithManyStates = () => {
  // Declare state variables as needed
  const [name, setName] = useState(``);
  const [count, setCount] = useState(10);
  const [myObj, setMyObj] = useState(
    { myKey1: `myVal1`, myKey2: true});
  //…
```

Detailed documentation: https://reactjs.org/docs/hooks-reference.html#usestate

TYPES OF HOOK



Effect Hook

Replacment for **`ComponentDidMount`**, **`ComponentDidUpdate`** and **`ComponentWillUnmount`** life cycle methods.

Used to perform side effects on components.

- Fetching data, subscriptions or manually changing DOM from React components

Runs after React flushes changes to DOM – after every render, including the first.

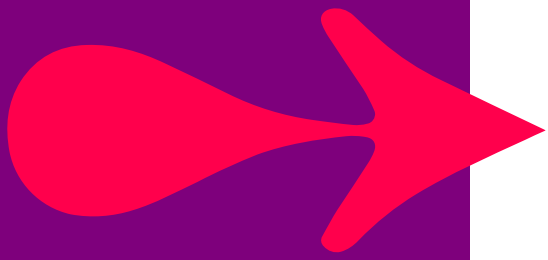
Declared inside component so have access to **`props`** and **`state.`**

Perform clean-up by adding a **return callback** function.

Can have more than one **effect** hook in a component.

Optionally takes a second argument of an array of dependencies to help decide if the effect should run.

TYPES OF HOOK – THE EFFECT HOOK EXAMPLE



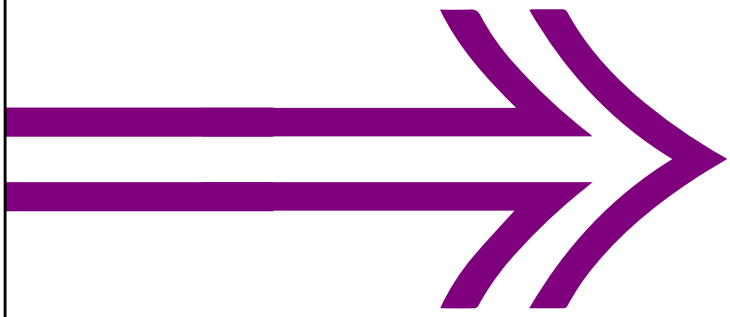
```
import React from `react`;
import { useState, useEffect } from `react`;

const Counter = () => {
  const [count, setCount] = useState(0);
  useEffect(() => {
    // Replaces to CDM and CDU
    document.title = `Clicked {count} times`;
    // Replaces CWU
    return(() => console.log(`Final: ${count}`));
  });
  return (
    <>
      <p>You have clicked the button {count}
         times</p>
      <button onClick={()=>setCount(++count)}>
        Click Me!
      </button>
    </>
  );
}
```

OTHER HOOKS

The documentation for Hooks provides several other flavours of Hooks (not discussed in detail here):

- `useContext`
- `useReducer`
- `useCallback`
- `useMemo`
- `useRef`
- `useImperativeHandle`
- `useLayoutEffect`
- `useDebugValue`

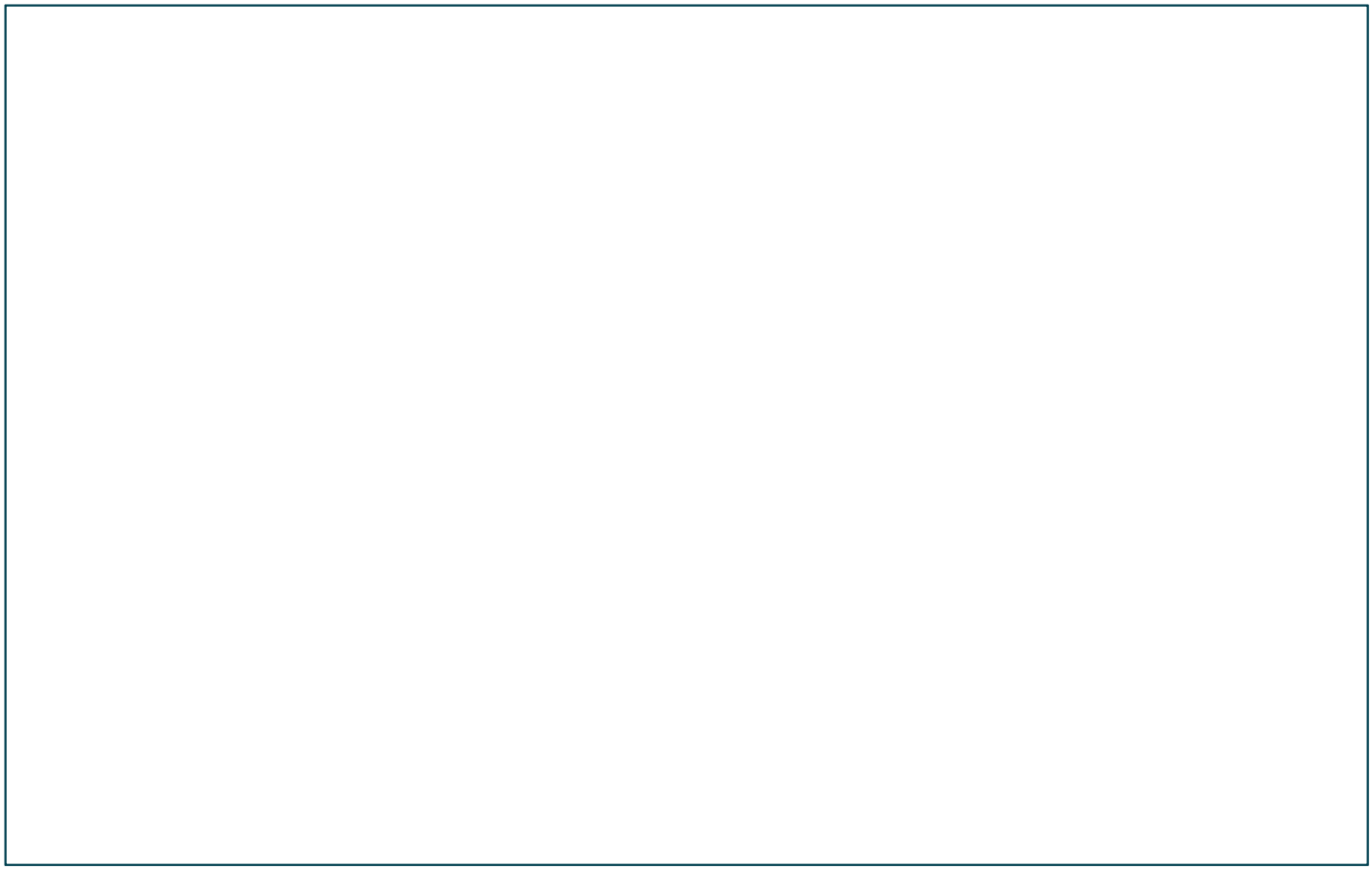
QuickLab 15 – The useEffect Hook



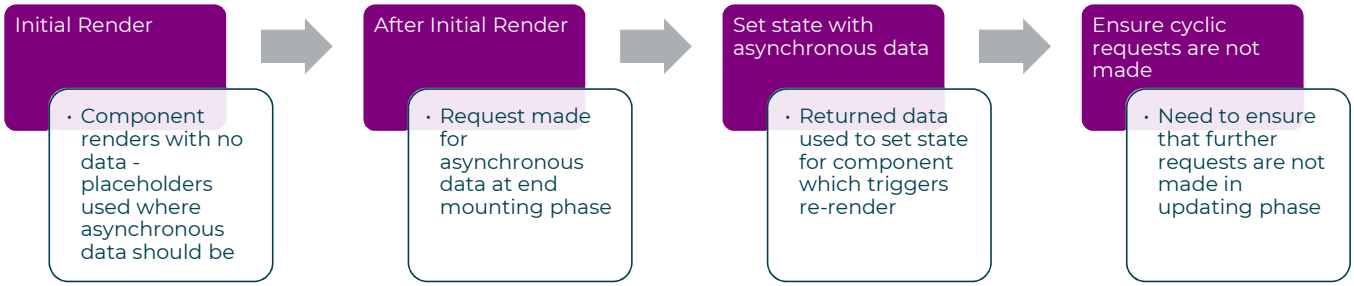
In this QuickLab, you will:

- Move the setting of state through the external data to the useEffect hook
- Further reduce the technical debt

Making External Requests for Data

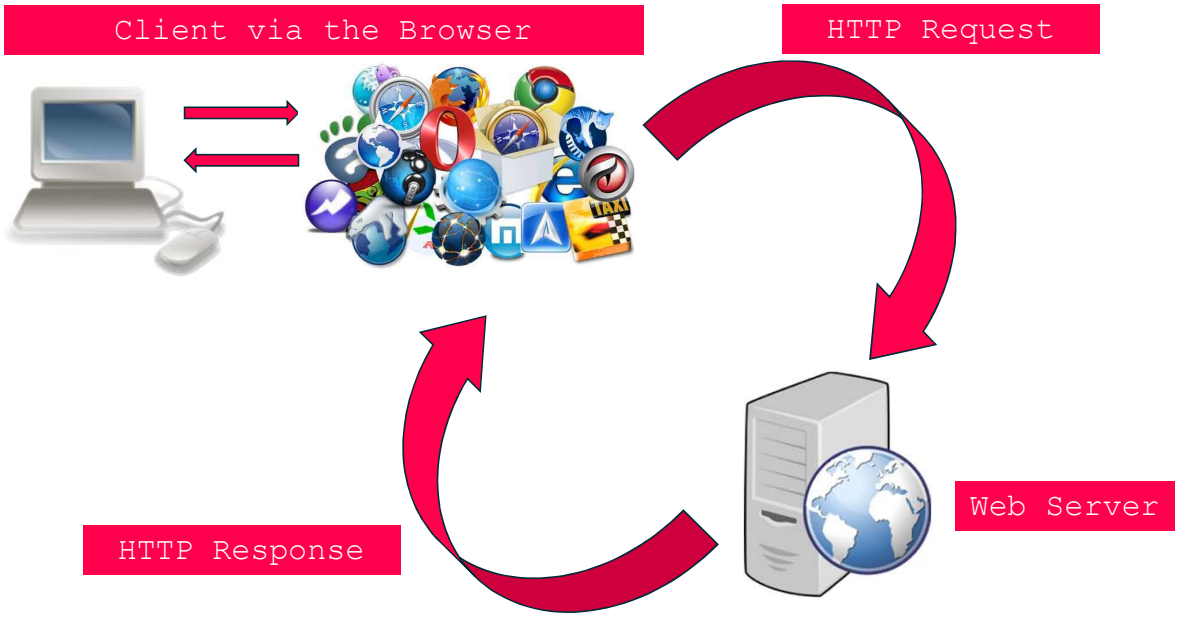


Making external requests for data



159

HTTP requests and responses



160

HTTP request structure

HTTP requests have:

1. Request Line
2. Zero or more header fields followed by CRLF (\r\n)
3. An empty line (CRLF) indicating the end of the header fields
4. An optional message body (ie., data / content)

```
POST /php/myapplication.php HTTP/1.1
User-Agent: Mozilla/4.0 (compatible; MSIE5; Windows NT)
Host: www.example.com
Content-Type: application/x-www-form-urlencoded
Content-Length: length
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Connection: Keep-Alive

key=value&username=example&email=me@example.com
```

161

HTTP responses

HTTP responses have:

1. A status line
2. Zero or more header fields followed by CRLF (\r\n)
3. An empty line (\r\n)
4. An optional message body (data, content)

```
HTTP/1.1 200 OK
Date: Mon, 21 Mar 2016 09:15:56 GMT
Server: Apache/2.2.14 (Win32)
Last-Modified: Mon, 21 Mar 2016 09:14:01 GMT
Content-Length: 88
Content-Type: text/html
Connection: Closed
<html>
<body>
<h1>Hello, World!</h1>
</body>
</html>
```

HTTP verbs

GET -- Retrieve information for a given URI
POST -- Send data to the given URI
PUT -- Replace data at the given URI
DELETE -- Remove data at the given URI

```
GET /index.php HTTP/1.1
User-Agent: Mozilla/4.0 (compatible; MSIE5; Windows NT)
Host: www.example.com
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Connection: Keep-Alive
```

```
DELETE /users/id/1012 HTTP/1.1
User-Agent: Mozilla/4.0 (compatible; MSIE5; Windows NT)
Host: www.example.com
```


Others:

Often not implemented or of limited use, there are other HTTP Verbs
HEAD
OPTIONS
TRACE

GET

GET requests have no request body.

- The URI specifies most of the data the server requires

The protocol, eg. HTTP.

The host, eg. example.com.

The resource location, eg. /index.php.

The query string, eg. ?username=eg&email=foo@example.com.

```
http://example.com/index.php?username=eg&email=foo@eg.com
```

The query string contains associative key/value pairs.

Technically limited to 255 characters however in practice they can be over 1000 long.

- Other information can be supplied with headers, either standard ones or custom headers typically preceded by "X-"

These are commonly seen when you are submitting a search form for products.

POST

POST requests *build on* GET requests by including a message body.

- They can use the query string system *and* the message body to send data
- Larger or more sensitive information is usually sent via POST

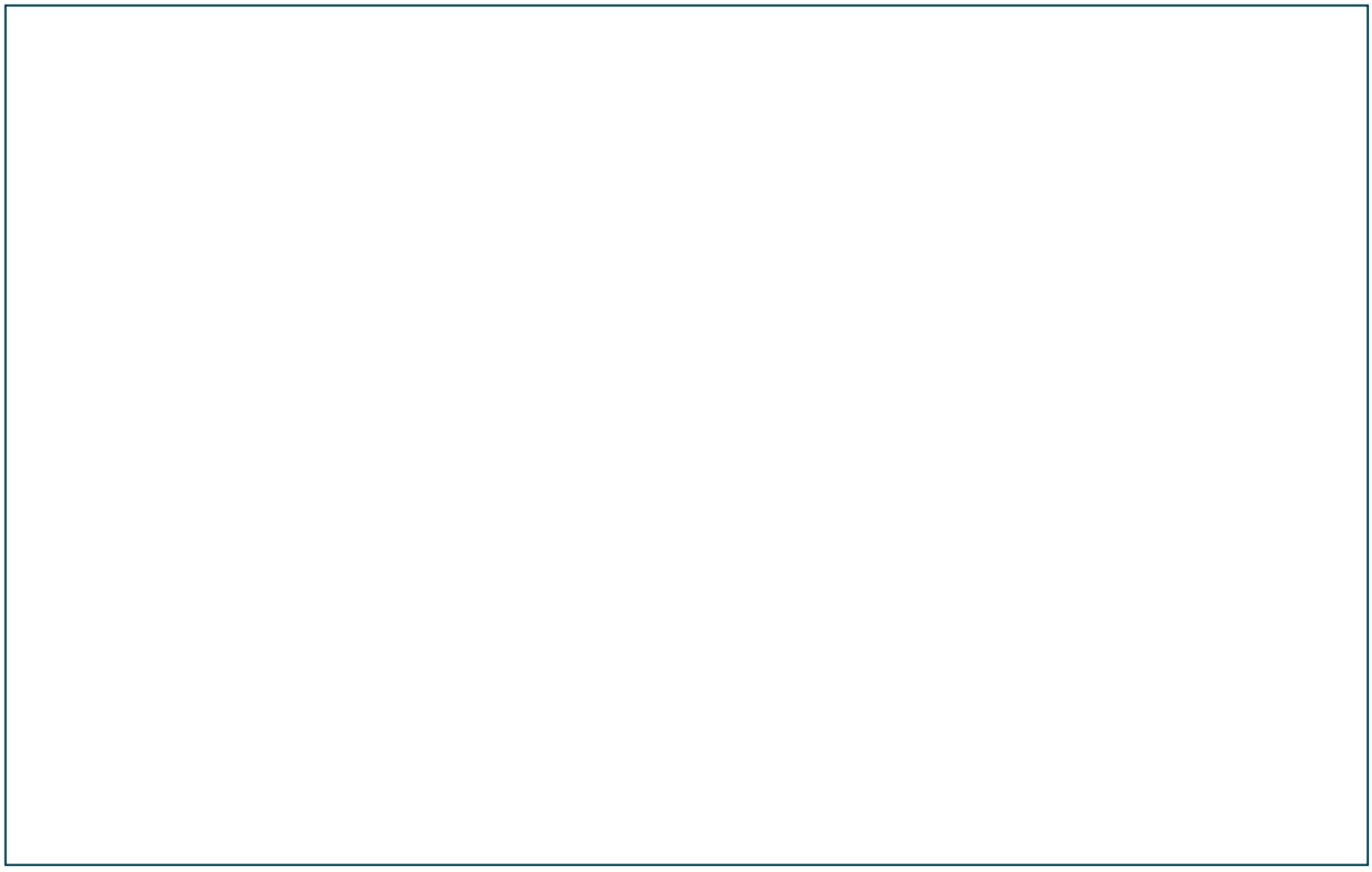
The message body can have a practically unlimited size (usually limited by what the server will accept, eg. 20MB).

The browser sends the POST body "behind the scenes" whereas the URL is visible in the address bar.

POST messages are *not* more secure than GET messages, however.

- Both are sent "plain text" over the network

Use HTTPS for encryption.



HTTP status codes

1**, eg. 102 Processing

- Informational responses – eg., processing still going on

2**, eg., 200 OK

- Successful responses

3**, eg. 301 Moved Permanently

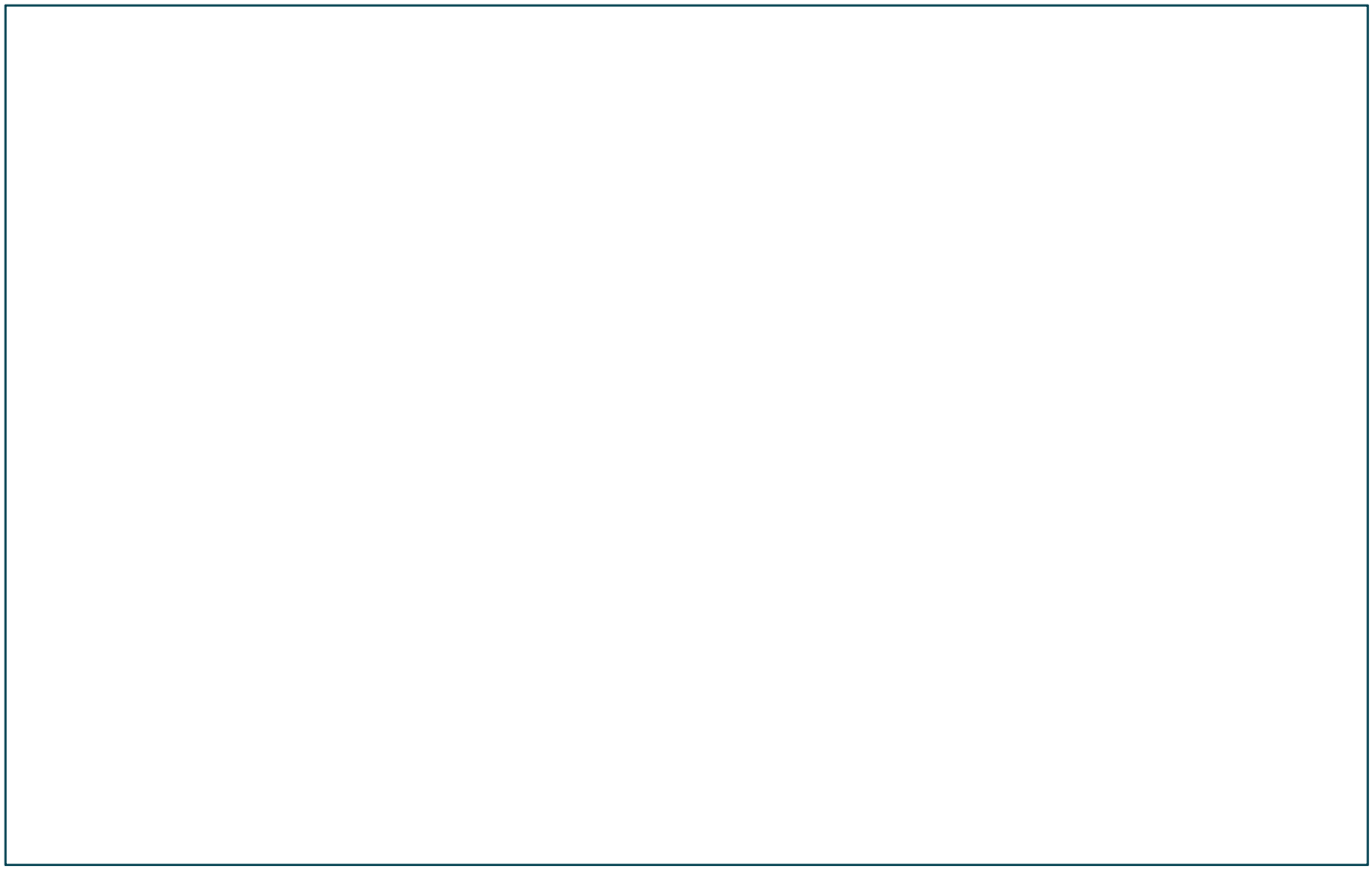
- Redirection – resource has moved to a different location

4**, eg.. 404 Not Found

- Client Error – a problem with the requesting client

5**, eg. 500 Internal Server Error

- Server Error – a problem with the responding server



RESTful APIs

Default data source for most web and mobile applications.

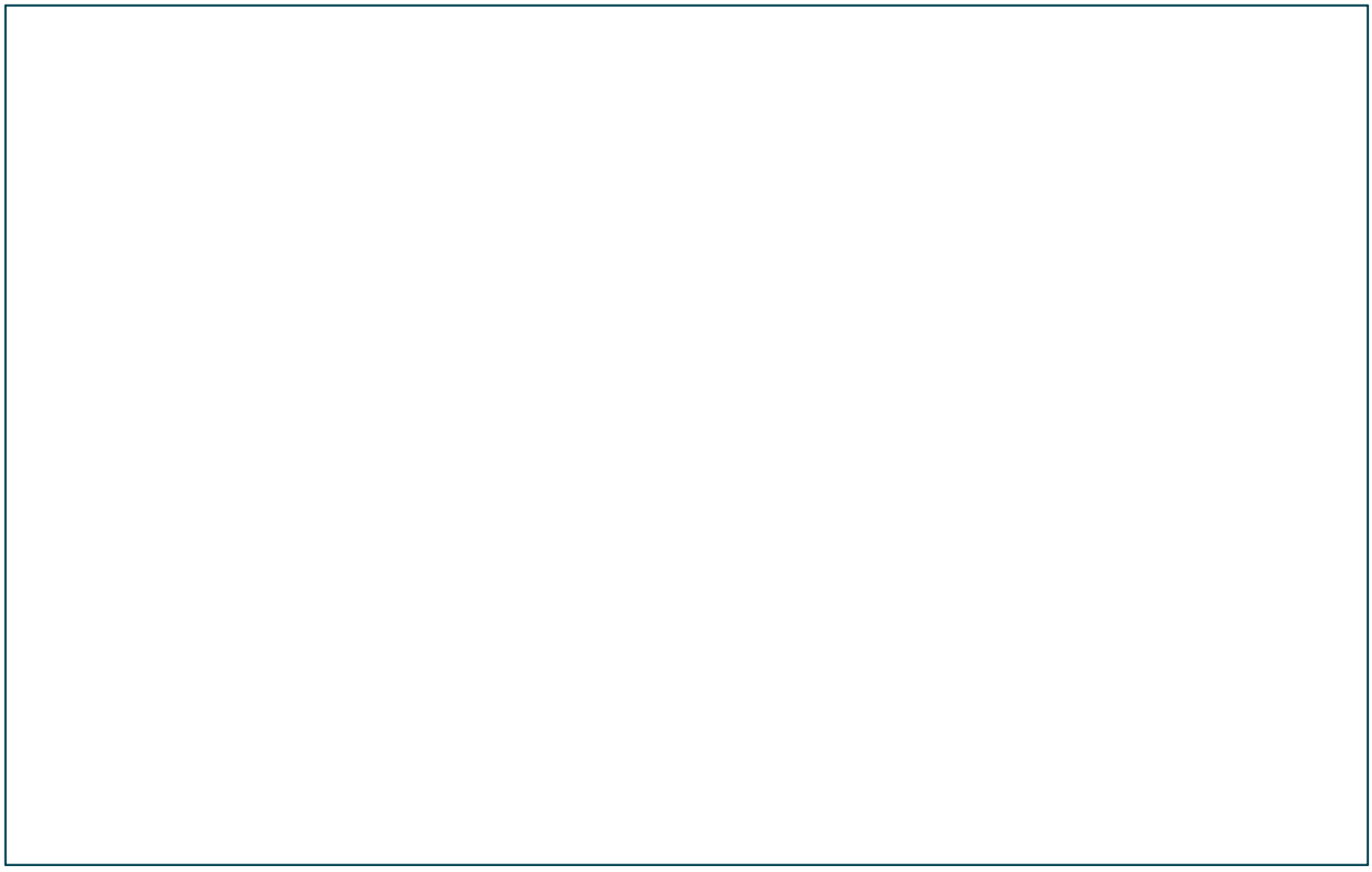
REST stands for Representational State Transfer.

- Lightweight
- Maintainable
- Scalable

Not dependent on any protocol.

- Most use HTTP or HTTPS as the underlying protocol

Can be made to provide data in JSON (desirable for JS applications) or XML.



JSON Server

Developing a solution that needs a RESTful API can be troublesome if the service is not available.
JSON Server is a "*full fake REST API with **zero-coding** in **less than 30 seconds** (seriously)*".

- Essentially needs:

A properly-formed JSON file for the data
An npm installation (either global or local)
Global is recommended by us as you will use it lots!

Once installed, point JSON server at your **.json** file

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

By default it runs on port 3000 – change this by adding a `-p <PORT_NO>` to the command

```
json-server --watch /path/to/file/filename.json
```


https://www.npmjs.com/package/json-server

CONTINUED 168

JSON Server

If you have a JSON file as shown, access is by using the format:

```
http://localhost:port/arrayName/objectId
```

You can use any type of HTTP verb request:

- GET
- POST
- PUT
- UPDATE
- DELETE

The JSON file is modified by requests to do so

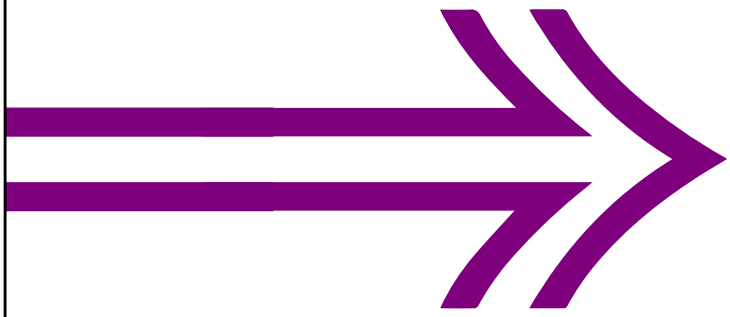
- id is immutable and autogenerated

```
{
 "posts": [
  {"id": 1, "title": "json-server" }
 ],
 "comments": [
  {"id": 1, "body": "comment", "postID": 1}
 ],
 "profile": {"name": "typicode" }
}
```

```
// So a request to:
// http://localhost:3000/posts/1
// will yield:


{"id" : 1, "title": "json-server" }
```

169

QuickLab 16 – Install JSON Server



In this QuickLab, you will:

- Install JSON Server globally
- Use a file to provide ToDo data as a fake RESTful service


React and RESTful APIs
Getting Data

React and RESTful APIs

Can make use of Fetch API and Promises.

- `fetch()` call returns a `Promise`

Promises can **resolve** or **reject**

Resolve allows for data to be returned and then passed to state (if desired).

Make initial requests for data in `useEffect` method.

- Called after component renders so placeholder can be used until data is returned from the source

```
useEffect(() => {
    fetch(`http://someRESTfulAPIURL.com`)
        .then(response => response.json())
        .then(people => setPeople(people));
}, []);
```


A Promise is a placeholder for asynchronous data that will be available immediately, some time in the future or not at all. Each .then passes the result of the previous action as the argument for the next.

For more information on Using Fetch see https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch
For more information on Promises see https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

React and RESTful APIs

ES2017 standard introduced **`async`** and **`await`** – returns a **`Promise`** still.

Allows removal of the chaining of **`then`** from promises.

Code on previous slide could have been rewritten as:

```
useEffect(() => {
  const getData = async () => {
       let response = await fetch(`http://someRESTfulAPIURL.com`);
       let people = await response.json();
       setPeople(people);
  };
  getData();
}, []);
```

Note: **`useEffect`** cannot have an asynchronous callback (at least in React v16.9)

The **`async`** function could be defined elsewhere in the code, although in this example, it is only ever used inside this **`useEffect`** hook, so it makes sense to define it here.

For more information on async/await see: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function

React and RESTful APIs

Both of these methods require the conversion of the response into JSON.

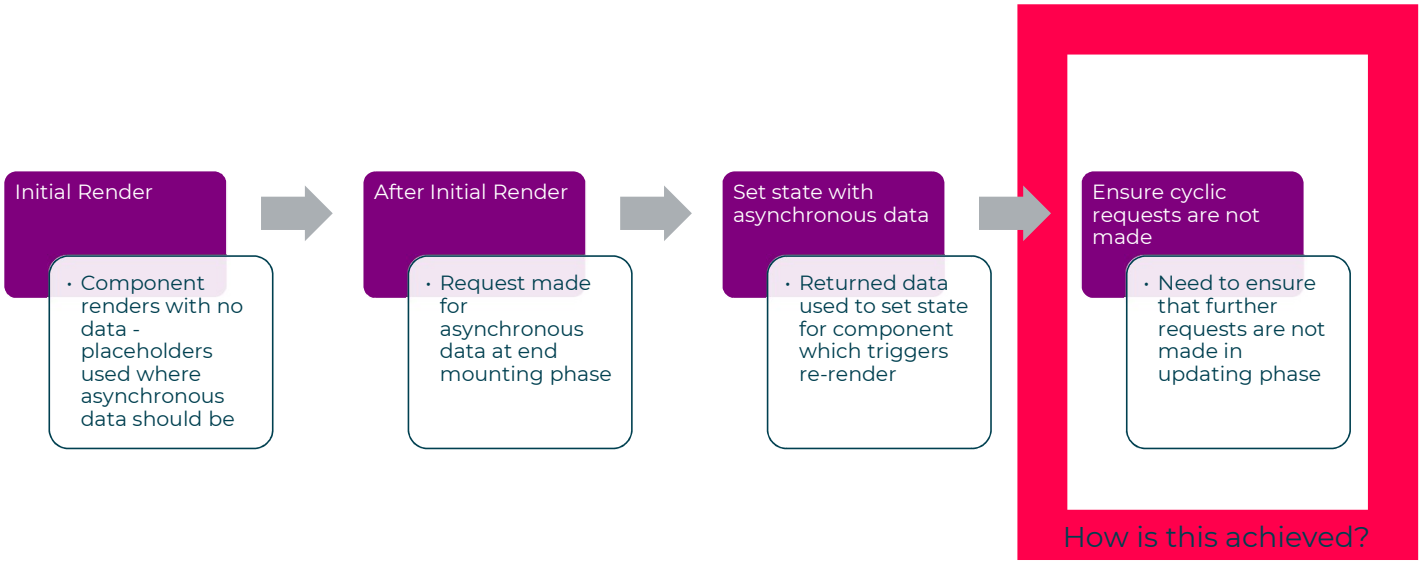
Axios is an **npm** package that simplifies the use of **`fetch/async-await`** by performing this step.

- Installed for the project using **`npm i --save axios`**
- Import **`axios`** for use, then modify the call, extracting **`.data`** from the **`response`**

```
useEffect(() => {
  const getData = async () => {
       let response = await axios.get(`http://someRESTfulAPIURL.com`);
       let people = await response.data;
       setPeople(people);
  };
  getData();
}, []);
```


For more information on async/await see: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function

Making external requests for data



175

Effect skipping

To help explain how the **`useEffect`** hook deals with stopping cyclic requests, understanding the class lifecycle method **`componentDidUpdate()`** will help.

- **`componentDidUpdate()`** takes 2 optional arguments, **`prevProps`** and **`prevState`**
- Can compare either of these values to conditionally run code

```
componentDidUpdate(prevProps, prevState) {
  if(prevState.count !== this.state.count) {
    document.title = `Button clicked ${this.state.count} times`;
  }
}
```

`useEffect` has an array of dependencies to match this behaviour.

- If all dependency values are unchanged, the effect is not run
- If any of the dependency values change, the effect runs

```
useEffect(() => {
 document.title = `Button clicked ${count} times`;
}, [count]);
```

176

A full explanation of this can be found at: https://reactjs.org/docs/hooks-effect.html#tip-optimizing-performance-by-skipping-effects

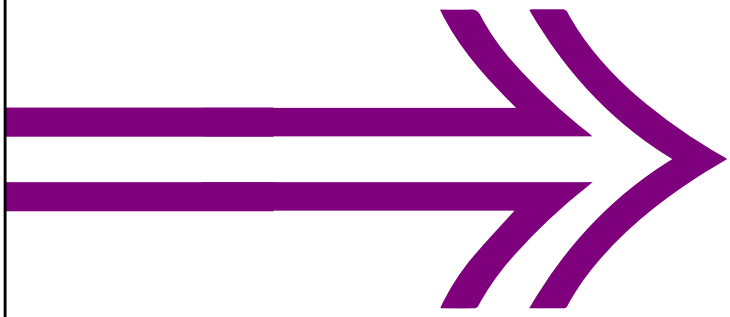
Effect skipping to avoid cyclic requests

An empty dependency array essentially tells React to only run this effect on initial render.

```
useEffect(() => {
  const getData = async () => {
       let response = await axios.get(`http://someRESTfulAPIURL.com`);
       let people = await response.data;
       setPeople(people);
  };
  getData();
}, []);
```

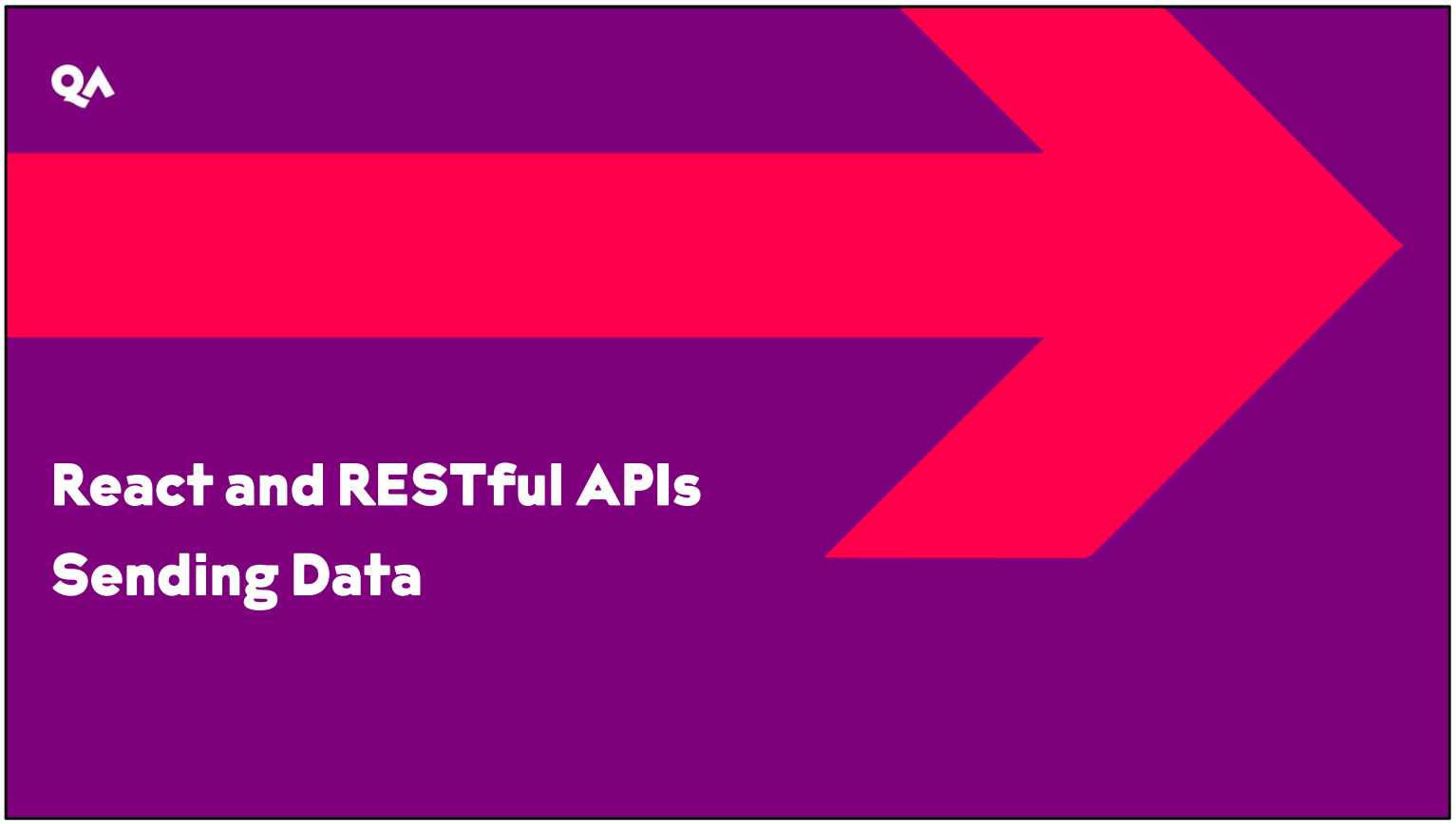

For more information on async/await see: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function

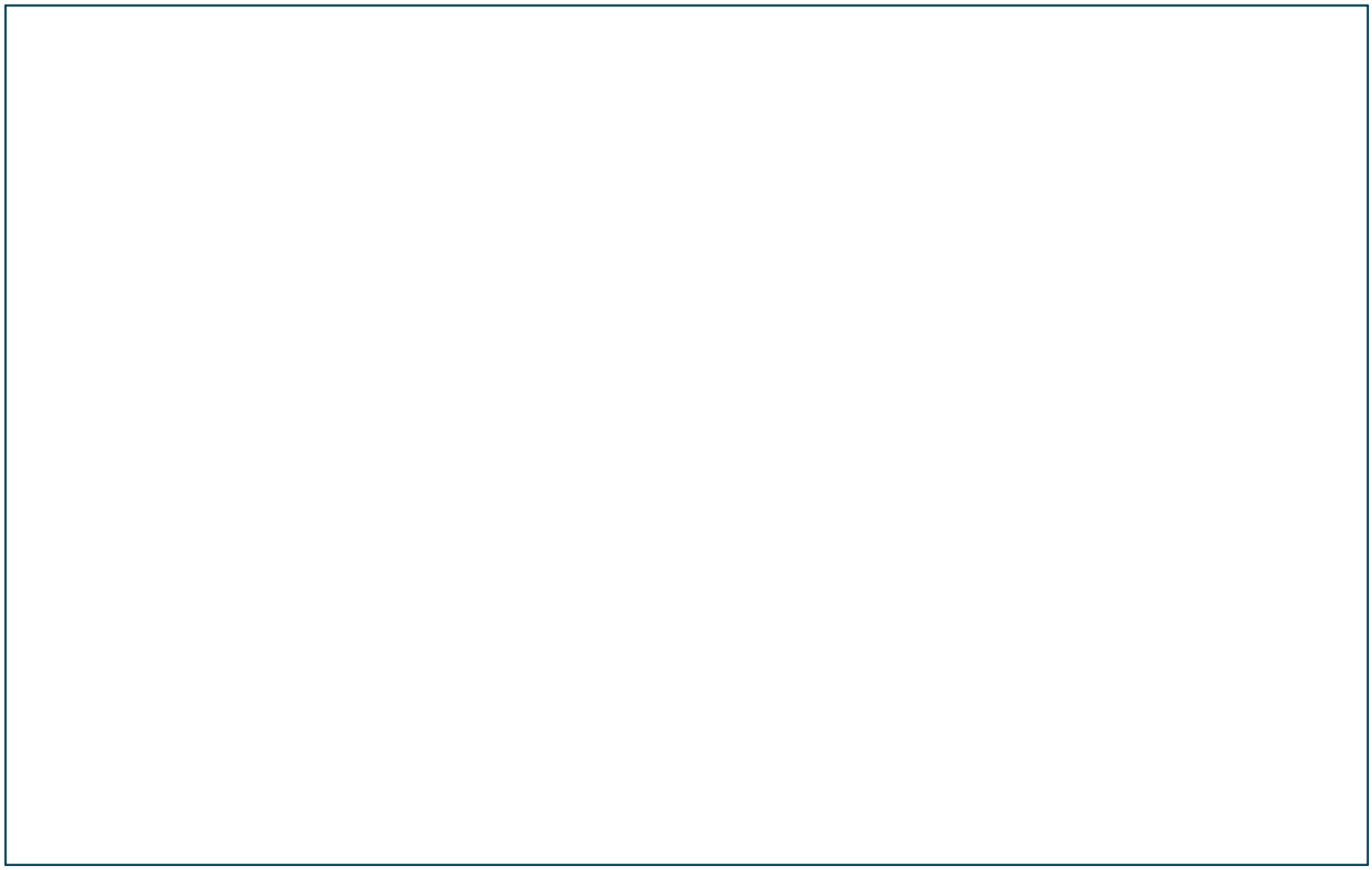
QuickLab 17a – Use an External Service – Getting Data



In this QuickLab, you will:

- Install axios and use it to retrieve an array of ToDos from an external source
- Implement the useEffect hook to facilitate the call to the external service
- Populate state with the result of the call


React and RESTful APIs
Sending Data



Sending data via POST/PUT requests

To persist data in the application, the external service needs to be able to receive new or updated data.

HTTP POST requests ask for a new entry to be created

- Sent to the URL used for ***ALL similar items***

HTTP PUT requests ask for an existing entry to be modified

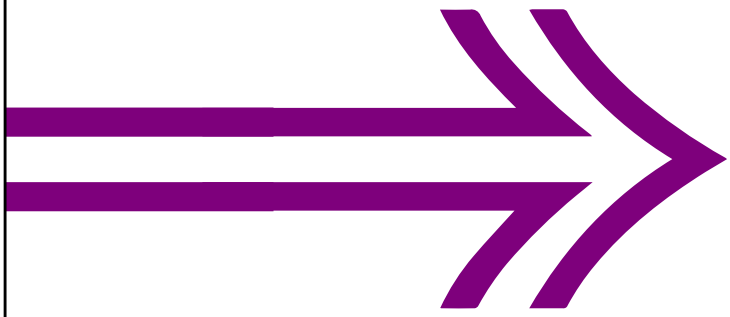
- Sent to the URL to access the ***individual item***

Both requests require a 'body' of the data to be created/updated.

```
const submit = async (data) => { await axios.post(URL_all, data); }

const update = async (data) => { await axios.put(URL_specific, data); }
```

180

QuickLab 18a – Use an External Service – Sending Data

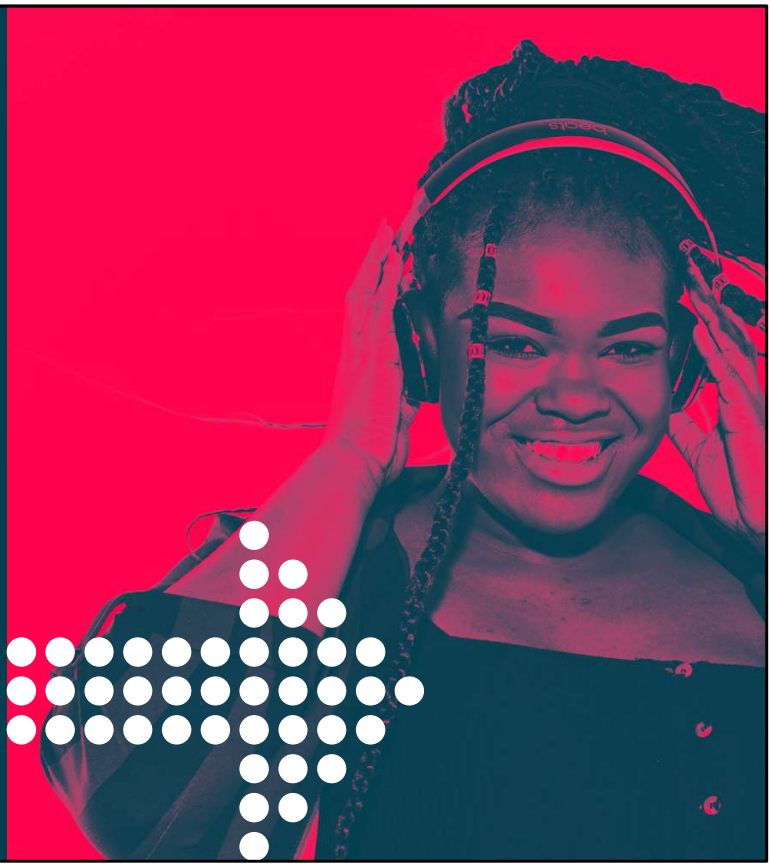


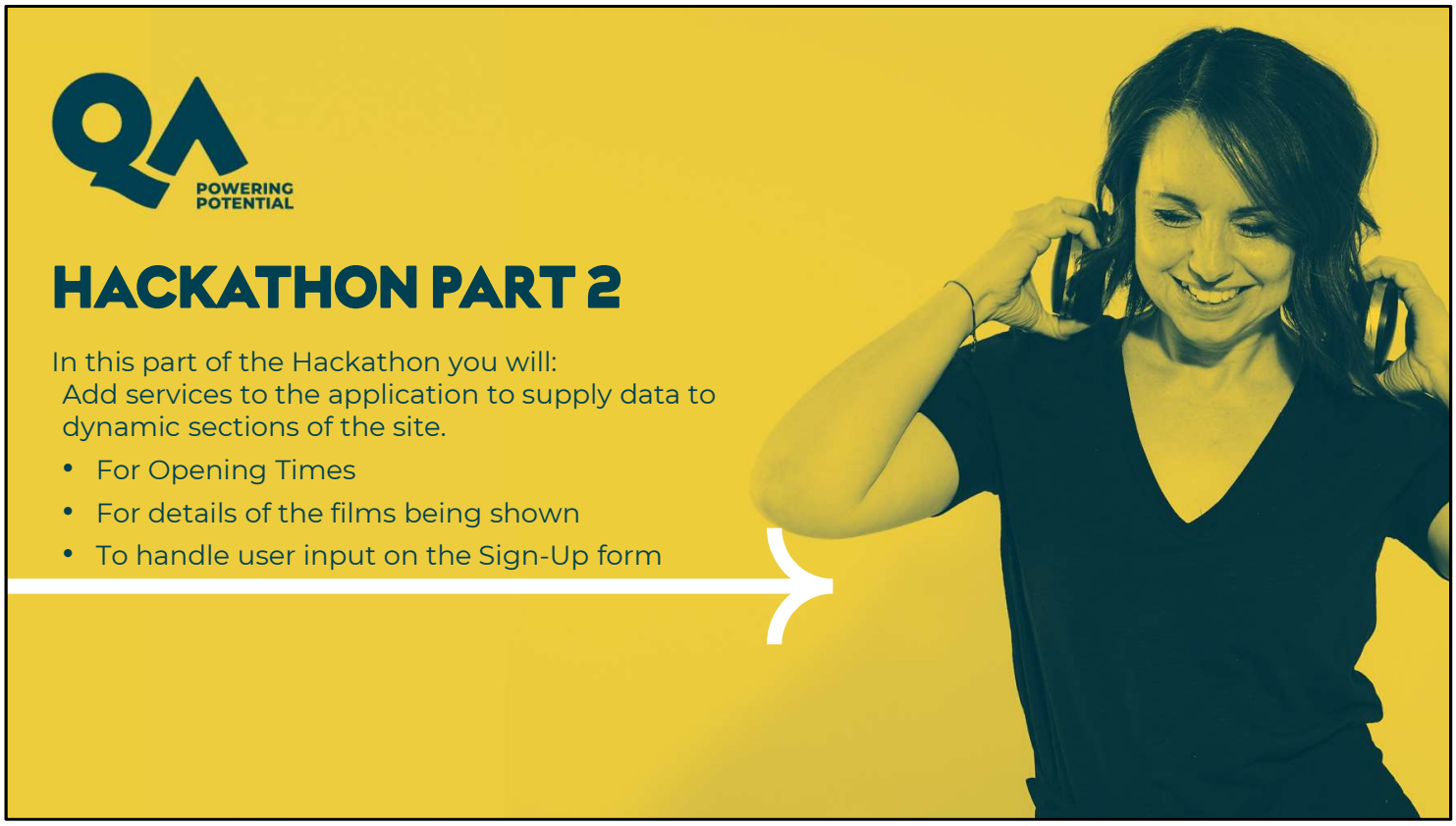
In this QuickLab, you will:

- Use a POST request to the external service to send data from the application
- Use a PUT request to update data held in the external service from with data from the application

Objectives

- To understand component life cycles
- To be able to understand different hooks available in React
- To be able to use the useEffect hook
- To understand what HTTP Verbs are used in RESTful services
- To be able to set up a mock RESTful service
- To be able to get data from an external service
- To be able to send data to an external service



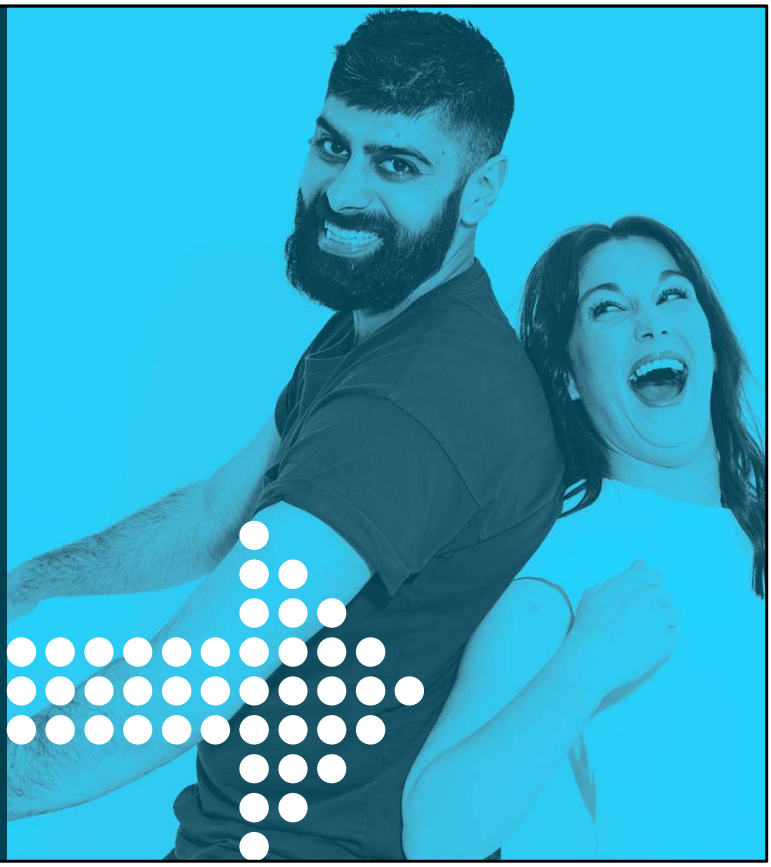


CONTINUED 183


Single Page Applications
Using React Routing
Building Web Applications Using React

Objectives

- To understand how routing can be added to an application
- To understand the roles of the router, route, switch and link components
- To be able to define what happens on a particular route
- To be able to set and read parameters on a route




Single Page Applications
and Routing Options

Single page applications

So far only looked at discrete components and pages.

Many modern applications are based on a single page.

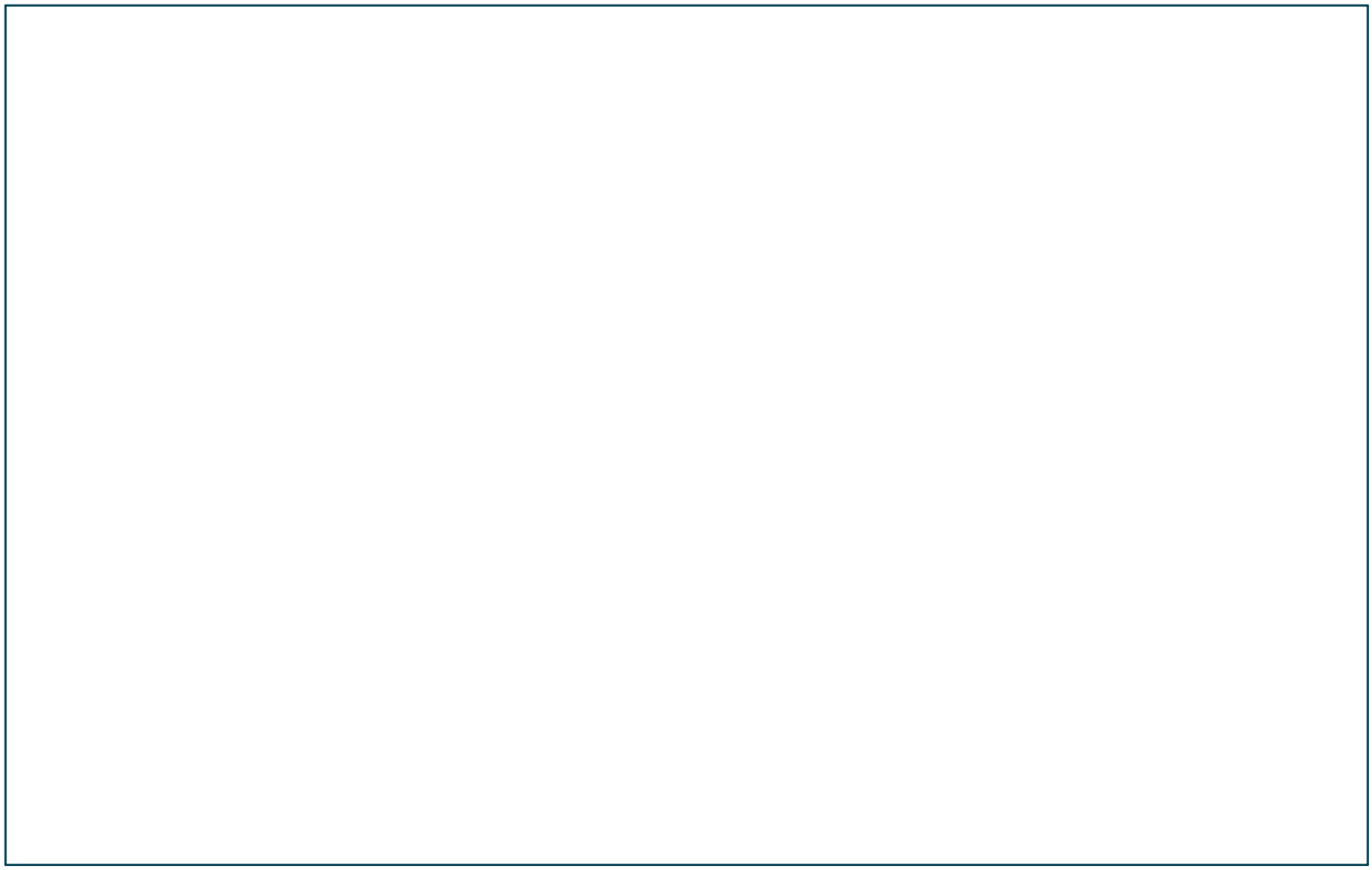
- Content within the page changes depending on user input
- Increases speed of web applications

  Only content to be changed is affected instead of whole page reloads

React-Router is standard routing library to allow this behaviour.

- From the ‘docs’:

  React Router is a powerful routing library built on top of React that helps you add new screens and flows to your application incredibly quickly, all while keeping the URL in sync with what's being displayed on the page.



Dynamic Routing v Static Routing

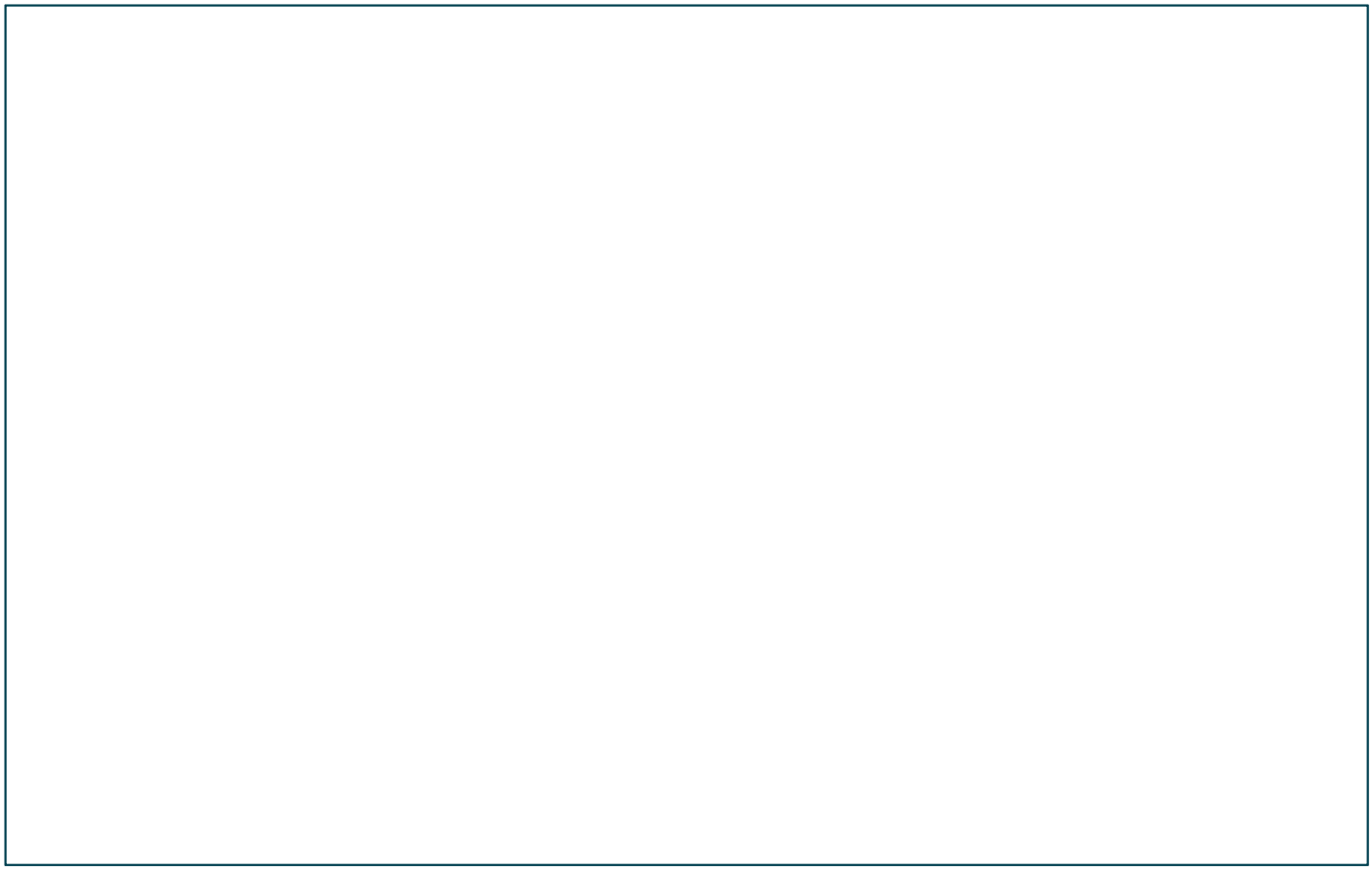
Most frameworks and libraries use static routing.

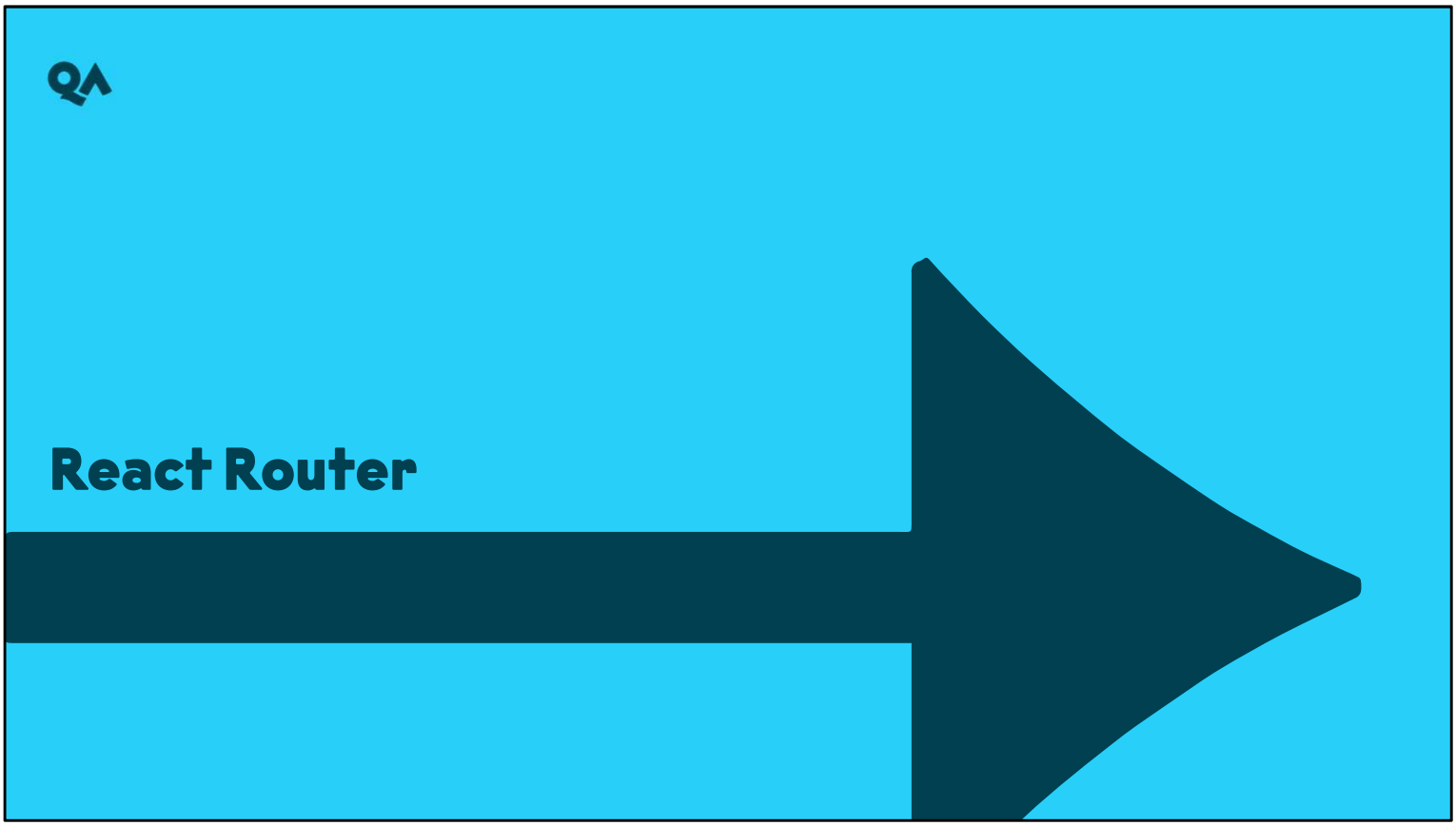
- Routes declared as part of initialisation of app before rendering
- Client side routers need routes to be declared upfront (such as in Angular)

From React Router v4, ***DYNAMIC ROUTING*** is used.

- Routing takes place as app is rendering
  Whole application is wrapped in a Router.
  Links defined within components.
  Route used to define which component should be rendered for the path.
  Route is just a component!
- Routing thought of as UI not static configuration

188




React Router

Routers

react-router-dom package has `<Router>` with 5 variations on the common low-level interface.

Typically high-level router used in app

- `<BrowserRouter>`
  Uses HTML5 history API to keep UI in sync with URL
- `<HashRouter>`
  Uses hash portion of URL to keep UI in sync
- `<MemoryRouter>`
  Keeps history of URL in memory (useful in testing and React Native)
- `<StaticRouter>`
  Never changes location – useful in server-side rendering

`<Router>` can be used to synchronise custom history with state management libraries.

For API information see: https://reacttraining.com/react-router/web/api/Router

Route

Most important to understand and learn to use.

```
<Route path render_method options />
```

Basic responsibility is to render a UI when a location matches route's `path`

- **`path`** – specifies the URL

3 ways to render from a **`<Route>`**:

- **`component`** – specified component will be created and rendered
- **`render`** – allows for convenient inline rendering and wrapping
- **`children`** – used to render whether the path matches the location or not – useful for animations

Other parameters:

- **`exact`** – match only if path matches **`location.pathname`** exactly
- **`strict`** – Boolean to represent if match should be made if trailing slash is present
- **`object`** – match path to passed location object's history (current by default)

For API information see: https://reacttraining.com/react-router/web/api/Route

Render a route with component

Simplest way to show a component on a route.

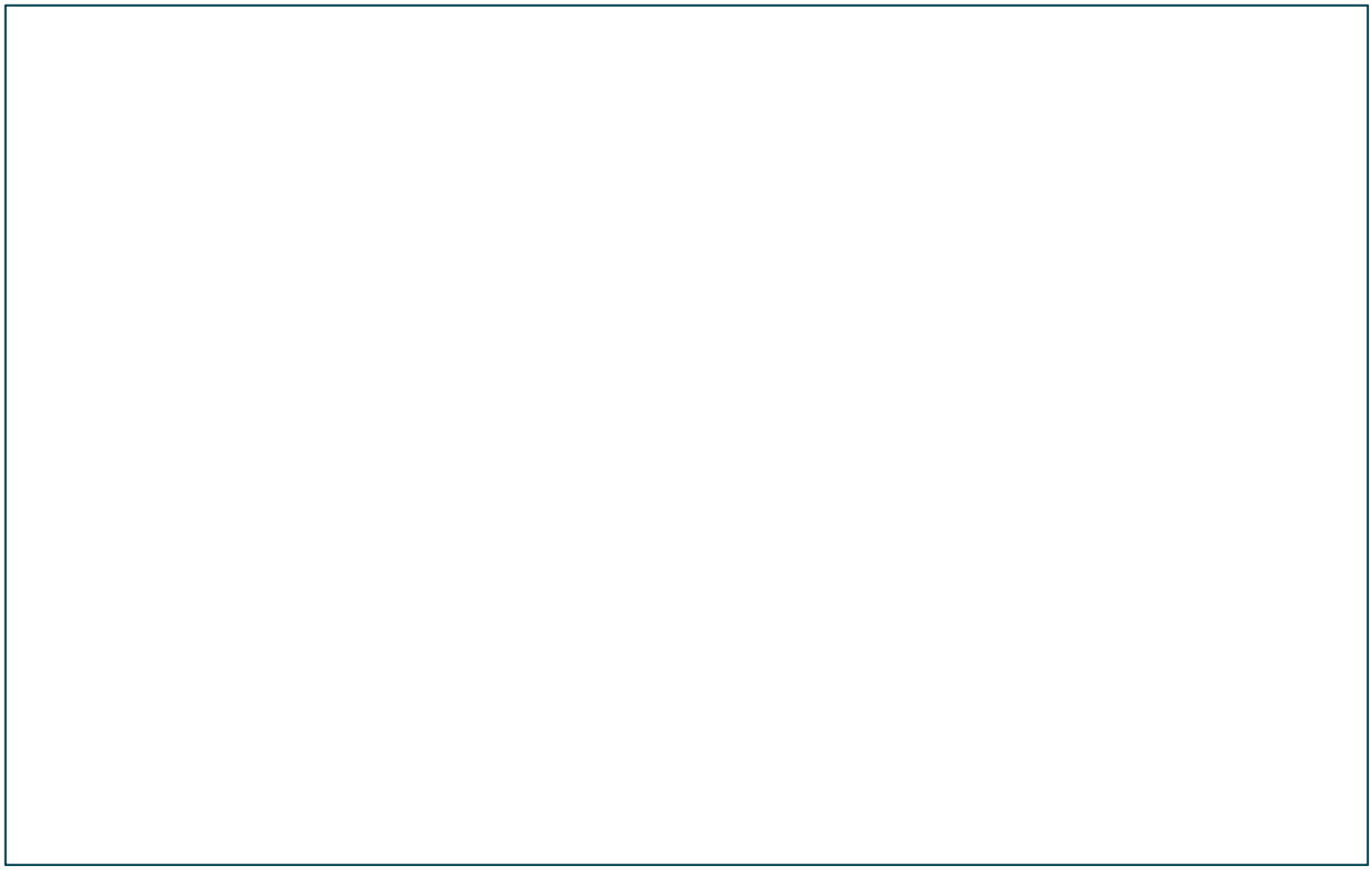
Good for components that don't need any props passed to them.

```
<Switch>
   <Route path="/" exact component={App} />
   <Route path="/subContent1" component={SubComponent1} />
   <Route path="/subContent2" component={SubComponent2} />
</Switch>
```

<Switch> is used to ensure that only the first route that matches the path is rendered.

- If not used, interesting and unexpected rendering can occur
- It is part of the **react-router-dom** package

192



Render a route with render

Needed if props need to be passed to a component to be rendered on a route.

Takes a callback that returns the desired component.

Can be mixed with other routes that have components.

No performance difference to rendering with component.

```
<Switch>
   <Route path="/" render={ props => <App {...props} newProp={newProp} /> } />
   <Route path="/subContent1" component={SubComponent1} />
</Switch>
```

193

Render a route with children

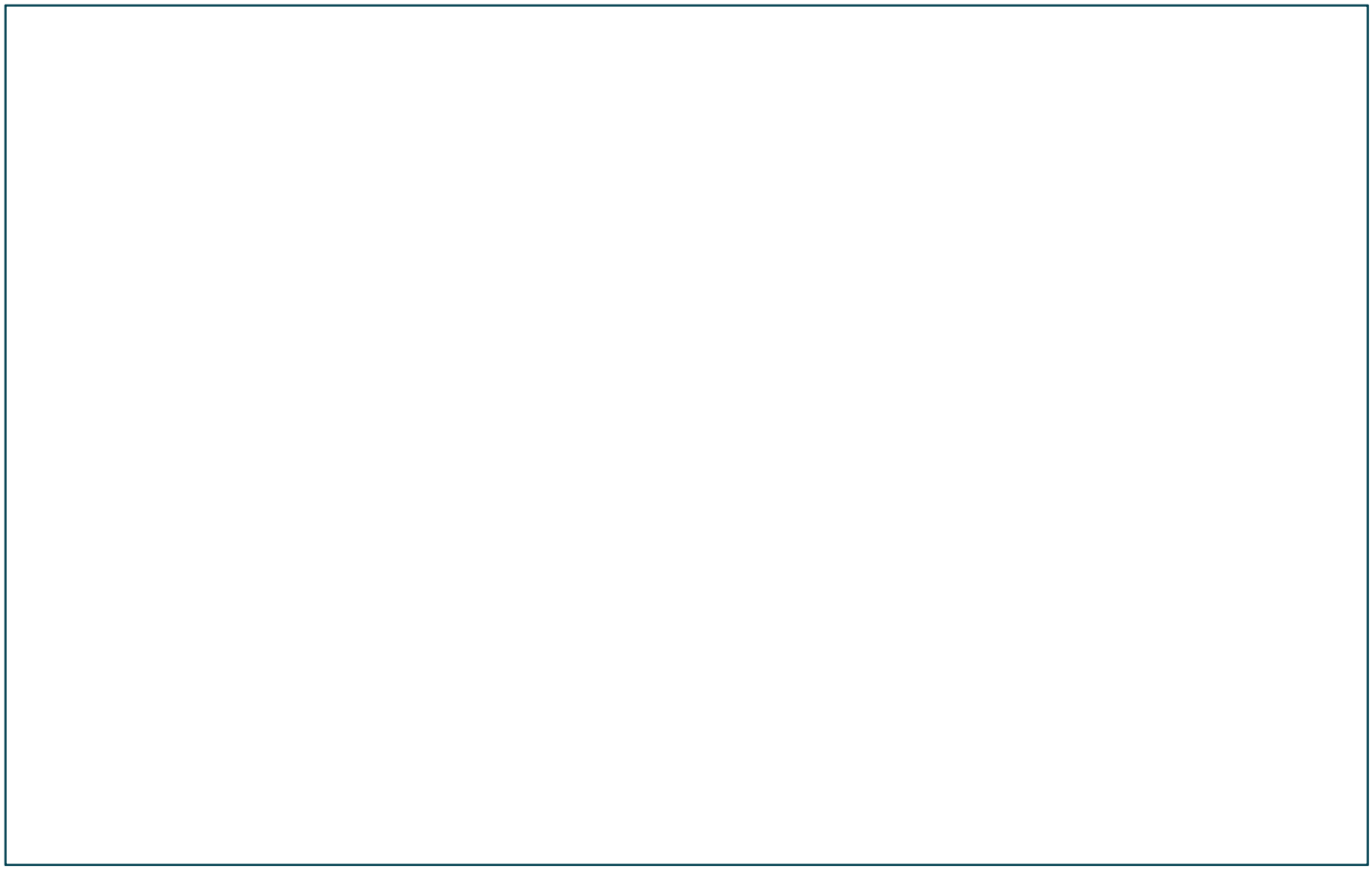
Used to render whether the path matches the location or not.

Works like render except it gets called whether there is a match or not.

Receives all same route props as component and render methods.

- When route fails to match URL, match is null
- Allow dynamic adjustment of UI based on whether the route matches

```
<>
  <Route path="/" children={({match, props}) =>
    <div className={match ? "active" : ""}>
      <SomeComponent {...props} /> }
    </div>
  }/>
</>
```

Defining and linking to routes

Provide **<Link>** components to create hyperlinks in the appropriate components.

```
<Link to="/">Home</Link>
```

- The **to** attribute can be either a **string** or an **object**

  The object has 4 properties: **pathname**, **search**, **hash** and **state**
- Other attributes include **replace**, **innerRef** and **others**

Provide **<NavLink>** when style attributes need to be added to the link when it matches the current URL object.

```
<NavLink to="/" activeClassName="selected">Home</Link>
```

- It includes all attributes from **Link** plus extras like **activeClassName**, **activeStyle**, **isActive** and and some others

The API documentation can be seen at: https://reacttraining.com/react-router/web/api/Link and https://reacttraining.com/react-router/web/api/NavLink

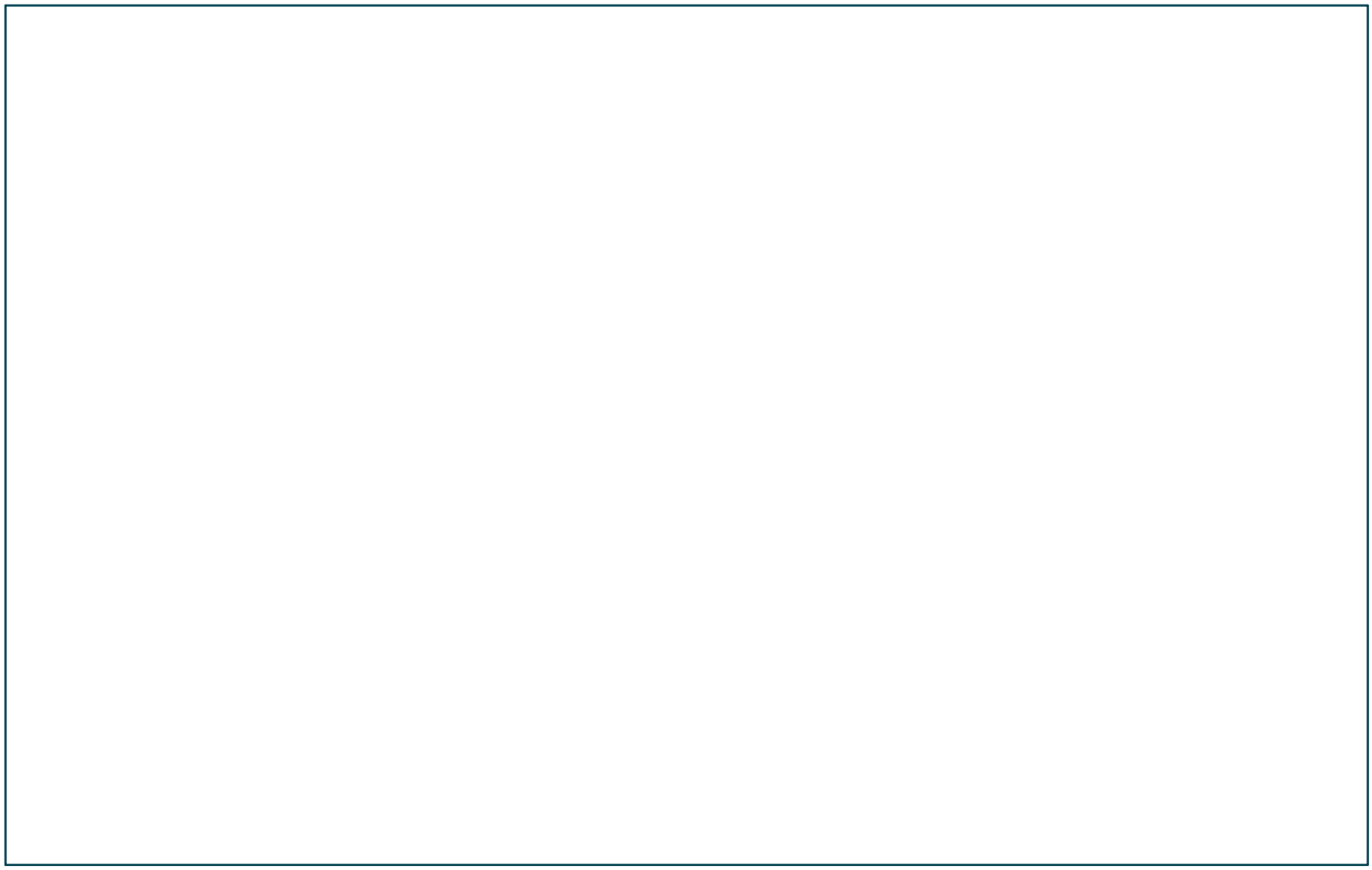
Redirecting routes

Sometimes it is desirable to redirect the application depending on circumstance.

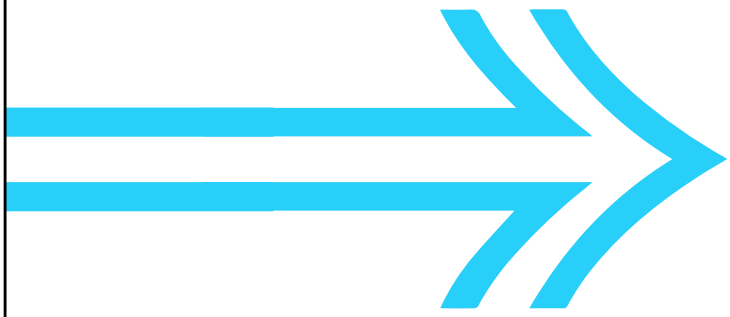
`<Redirect>` is supplied as a routing component.

- Needs a **`to`** attribute to define the path that should be redirected to
- Put in place of any markup

```
...
    <>
      {submitted ? (
        <Redirect to="/" />
      ) : (
        <p>Some other markup</p>
      )}
    </>
...
// Would render the Redirect if
// submitted is true
```

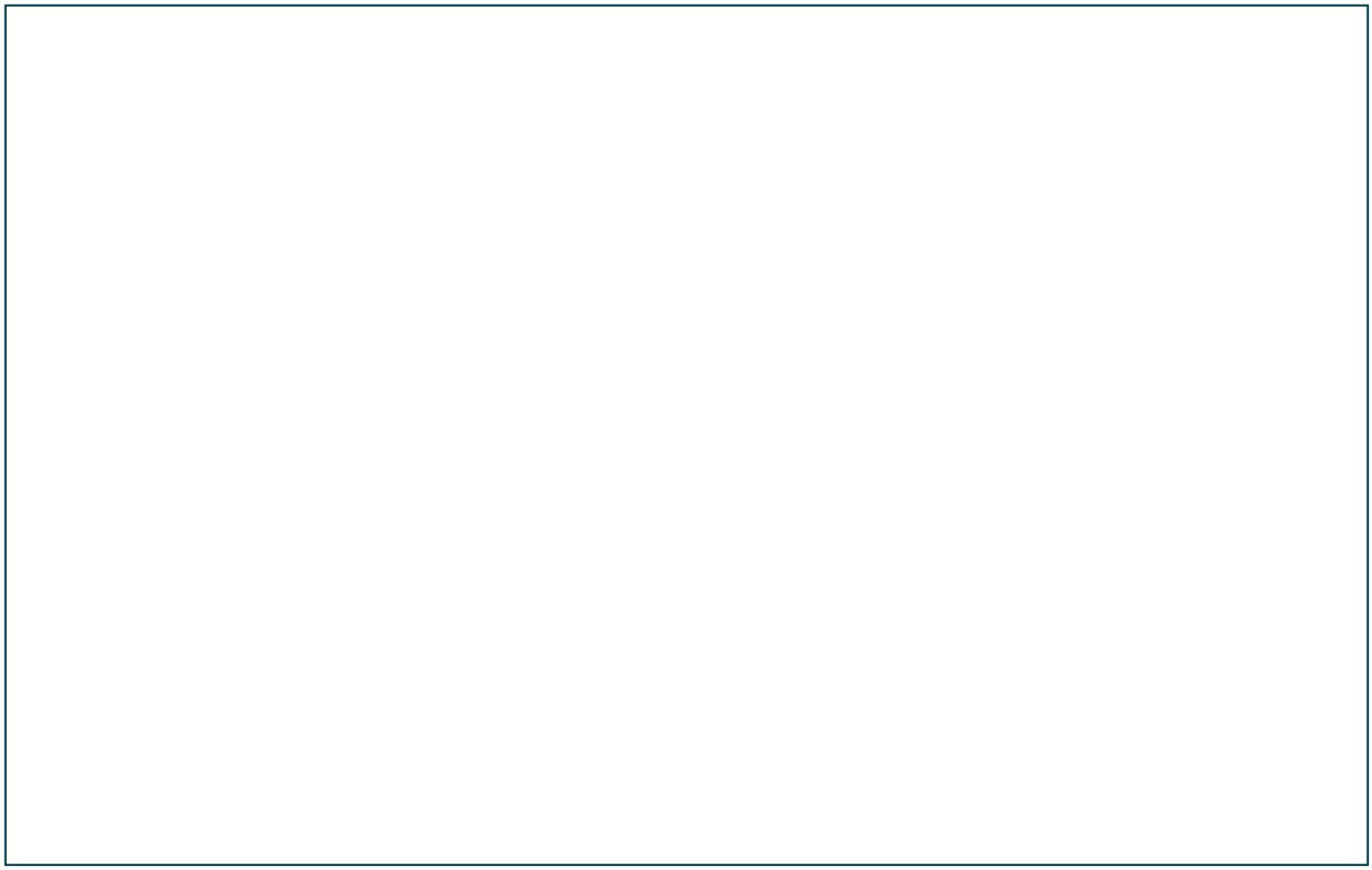
QuickLab 19 – Prepare the Main Application for Routing



In this QuickLab, you will:

- Install the react-router-dom package
- Define routes for the todo application
- Modify links within the application to use the routes defined


Parameterised Routes



Creating parameterised routes and links

Links to parameterised paths can be defined by hard coding or by supplying an expression, perhaps based on some property of an object.

```
<Link to="/content/subContent1" />

<Link to={`/content/${subContent.id}`} />
```

To define a route to a parameterised property, colon notation is used followed by the name of the parameter to use:

```
<Route path="/content/:subContentId" component={SubContent} />
```

Using parameterised routes - React Router <v5.1

Prior to ***React Router v5.1***, the following pattern would be used to access parameters in a route:

To use the **`params`** in the matched URL, **`props`** need to be passed to the rendering component and the **`match`** object accessed – in a render route its done by passing **`{...props}`** or **`{match}`**

```
const SubContent = props => (
   <h1>props.match.params.subContentId</h1>
);

// OR

const SubContent = ({match}) => (
    <h1>match.params.subContentId</h1>
);
```

Using parameterised routes - React Router v5.1+

React Router v5.1 introduced a number of HOOKS to use with routes.

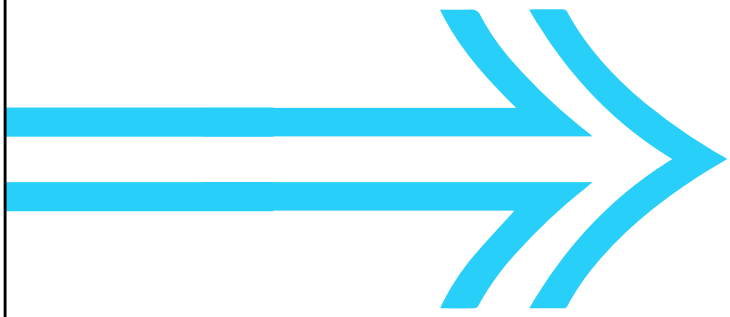
One such hook was the **`useParams`** hook – imported from **`react-router-dom.`**

To use route parameters, simply *deconstruct* the result of the **`useParams`** function.

```
import { useParams } from `react-router-dom`;

const SubContent = () => {
   const { subContentId } = useParams();
   return (
      <h1>{subContentId}</h1>
   );
};
```

Remember: Deconstructed names must match those in the object being deconstructed.

QuickLab 20 – Use Parameterised Routes

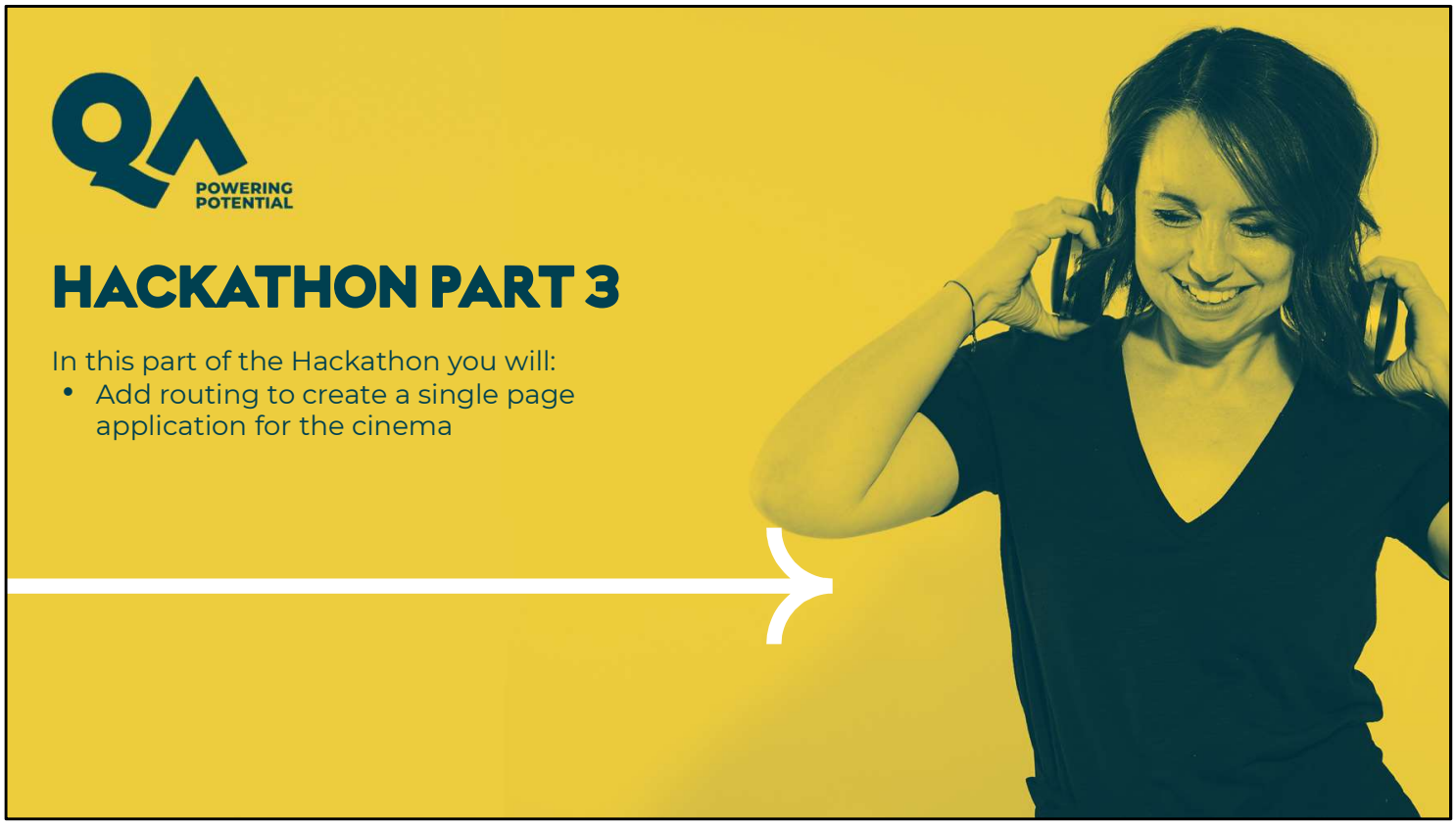


In this QuickLab, you will:

- Change the link surrounding the Edit on the AllTodos view to supply the Todo's id as part of the link
- Make the Add/Edit Todo UI recognise the parameter supplied and display the Todo for editing

Objectives

- To understand how routing can be added to an application
- To understand the roles of the router, route, switch and link components
- To be able to define what happens on a particular route
- To be able to set and read parameters on a route

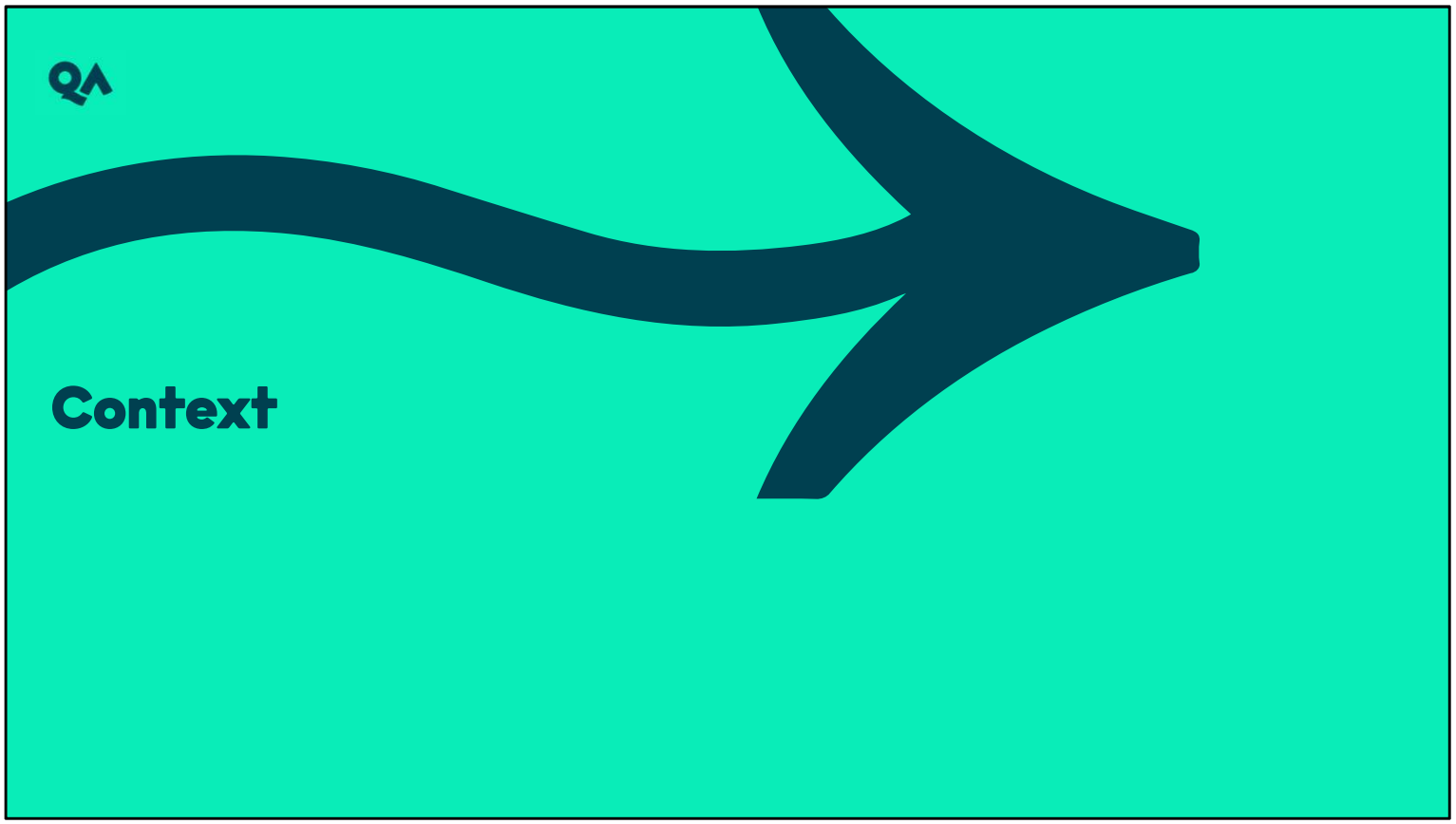


CONTINUED 204


State Management
Building Web Applications Using React

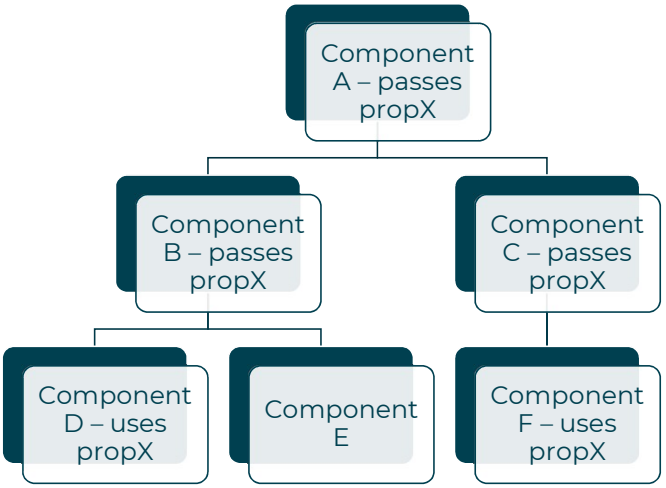
Objectives

- To understand what context is and how it is used in React
- To be able to use the useContext hook to help simplify data drill-downs
- To understand what reducers are and how they are used in React
- To be able to use the useReducer hook to enable components to dispatch actions


Context

WHY CONTEXT?

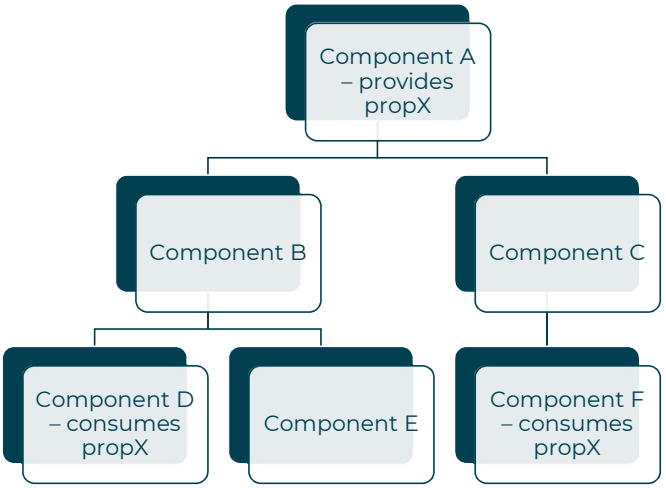
It solves a problem and cleans up code:



Passing props deep into component trees adds complexity to their parent components.

WHY CONTEXT?

Context allows component high in the tree to provide props to components lower in the tree without the need to pass through every component.



USE CASE FOR CONTEXT

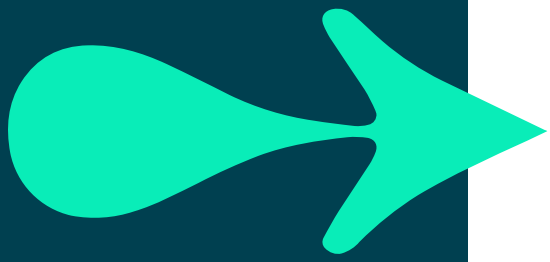
1. When multiple components across component trees need access to the same data

E.g. A set of colour styles are needed for components across an application.

Note: should not be used simply to bypass components a single tree.

210

HOW TO CREATE A CONTEXT



React provides a helper function called **`createContext`** to actually create the React context.

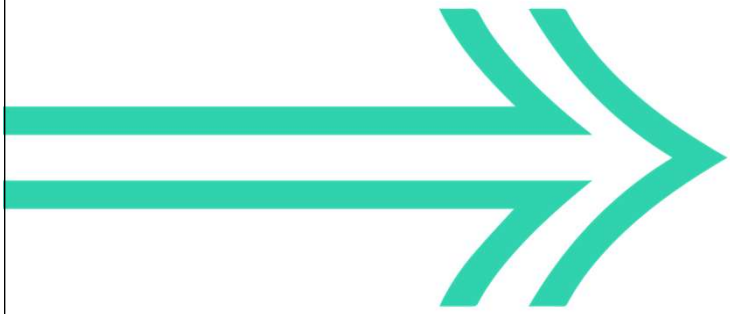
- An initial value can be set

```
import React, {createContext} from `react`;
const MyContext = createContext();
export default MyContext;
```

Context must then be 'provided' to a tree of components:

```
import React from `react`;
import MyContext `./MyContext`;
import {B, C, D, E} from `MyComponents`;

const A = () => (
  <MyContext.Provider value={myData}>
    <B>
      <D/><E />
    </B>
    <C><F /></C>
  </MyContext.Provider>
);
export default A;
```

Quick Lab 21 – Activities 1-3 - Prepare for State Management and Provide Context



In this part of the QuickLab, you will:

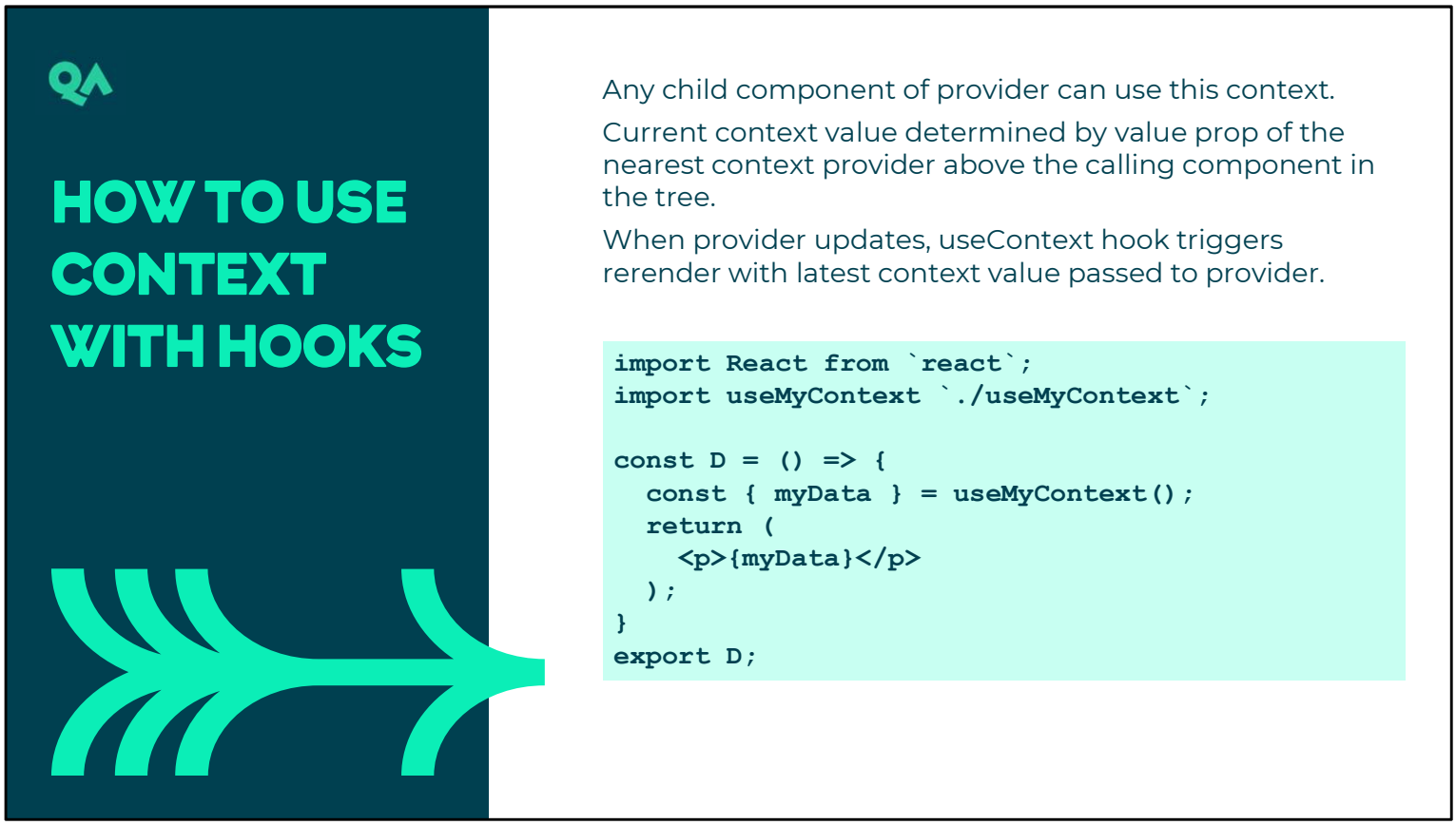
- Add providers for the state for Todos in the application

HOW TO USE CONTEXT WITH HOOKS

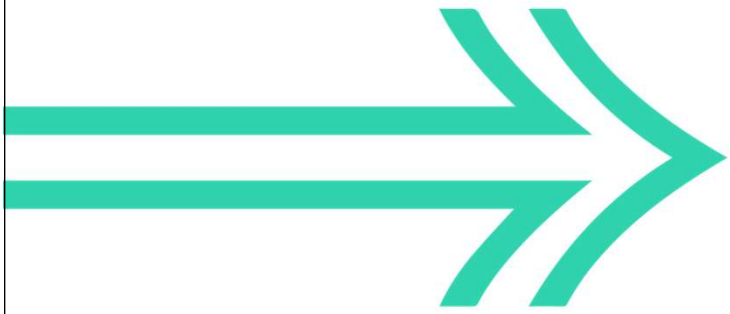
Need to provide a function to use in a component that has a context and use the useContext hook:

```
import React, {useContext} from `react`;
import MyContext from `./MyContext`;

const useMyContext = () => {
  const myContext = useContext(MyContext);
  return myContext;
}
export default useMyContext
```

useContext accepts context object created by calling createContext.

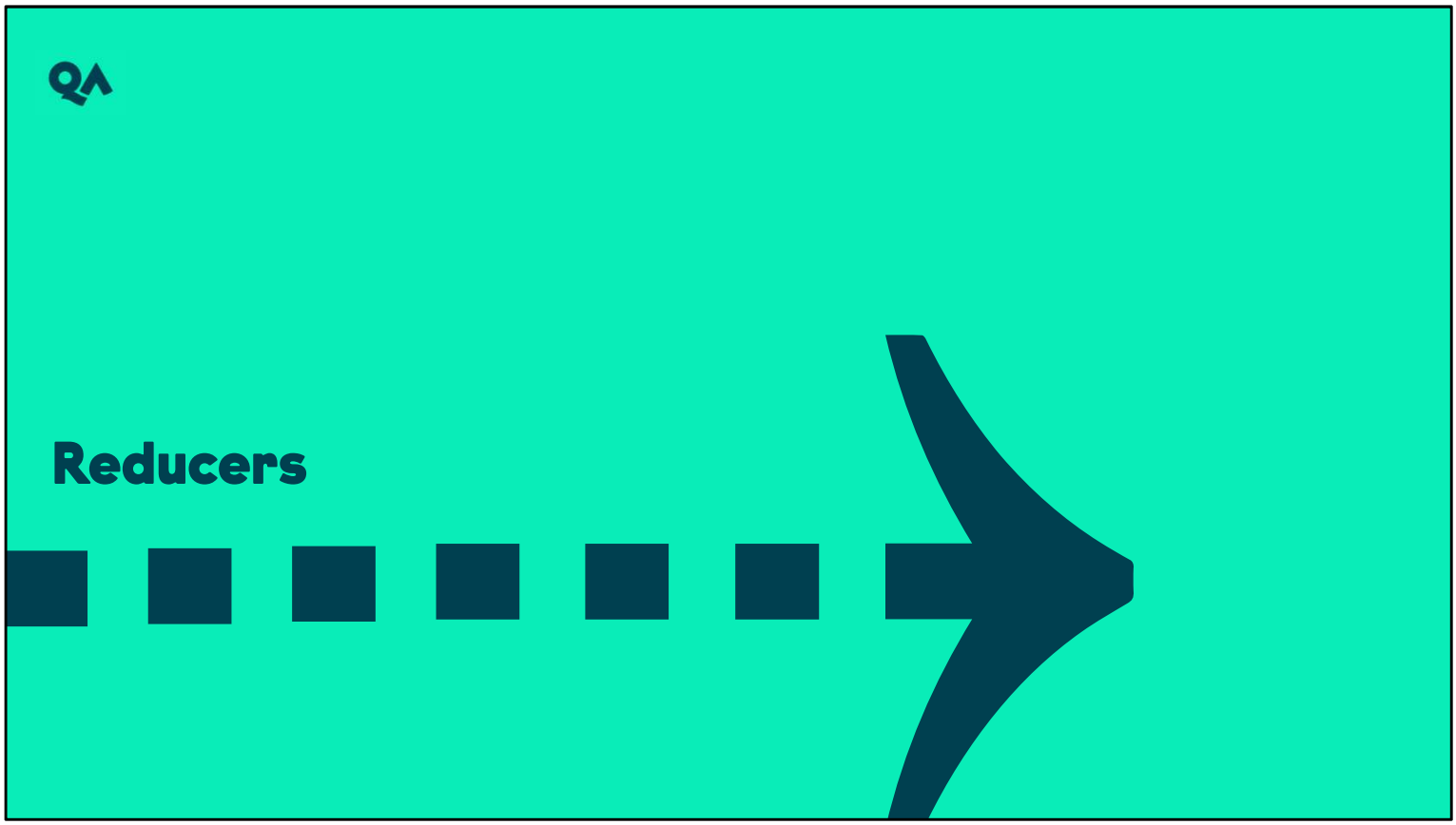

HOW TO USE CONTEXT WITH HOOKS
Any child component of provider can use this context.
Current context value determined by value prop of the nearest context provider above the calling component in the tree.
When provider updates, useContext hook triggers rerender with latest context value passed to provider.
import React from `react`;
import useMyContext `./useMyContext`;
const D = () => {
const { myData } = useMyContext();
return (
{myData}
);
}
export D;

Quick Lab 21 – Activities 4 – 5 – Consume the Context



In this part of the QuickLab, you will:

- Consume the context in the application by making the AllTodos component use the state provided by context


Reducers

WHAT IS A REDUCER?

Basically a JavaScript function!

Takes 2 arguments

- Previous state
- An action

Uses these arguments to return a completely new state.

2 basic principles for using reducers:

- State is never mutated
- Previous state can be used as part of the new state by using the spread operator …**state**

```
const myReducer = (state, action) => {
  if(action) {
    return {…state, action.payload};
  }
  return state;
}
```

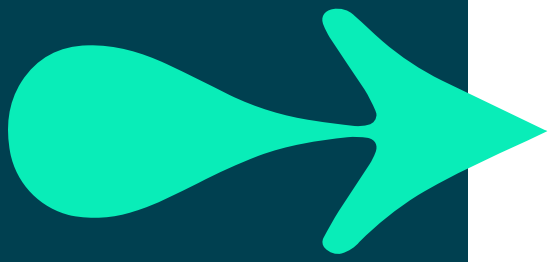
ACTION

`action` is usually an object supplied as part of a call to the reducer.

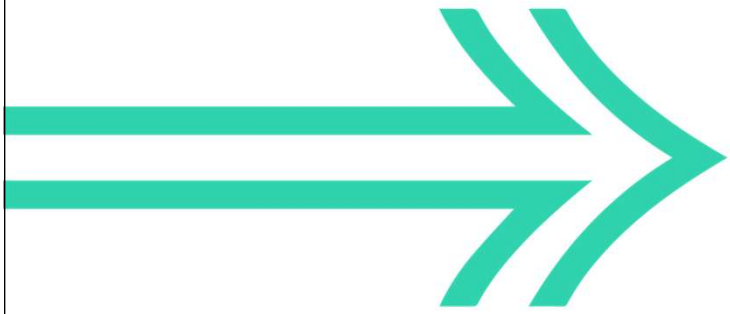
- It helps to identify the action that should be taken at this point to produce the new state

Action usually has at least 2 properties:

- `type` – a string to identify the action by
- `payload` – the data that has been dispatched as part of the update call

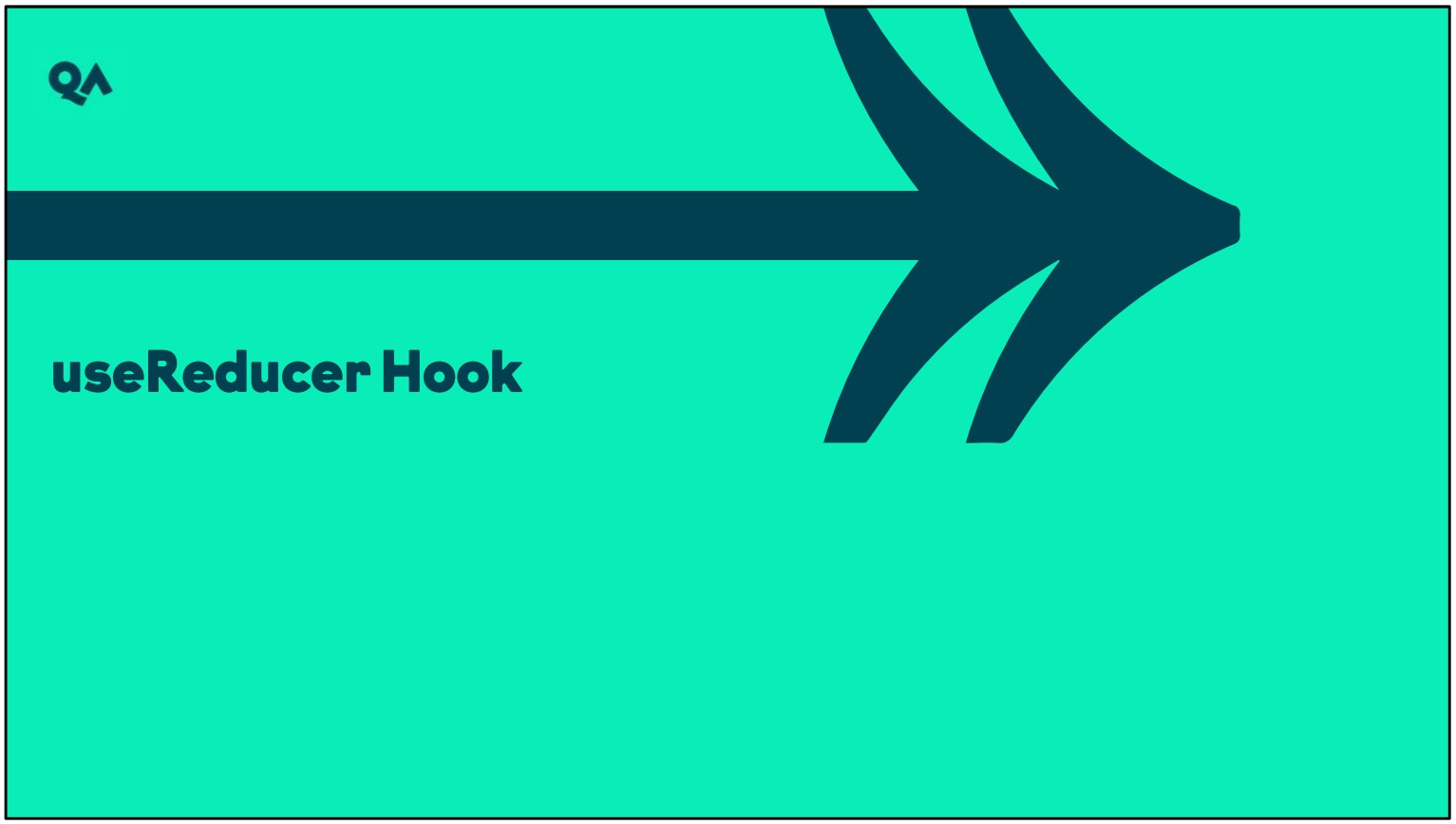


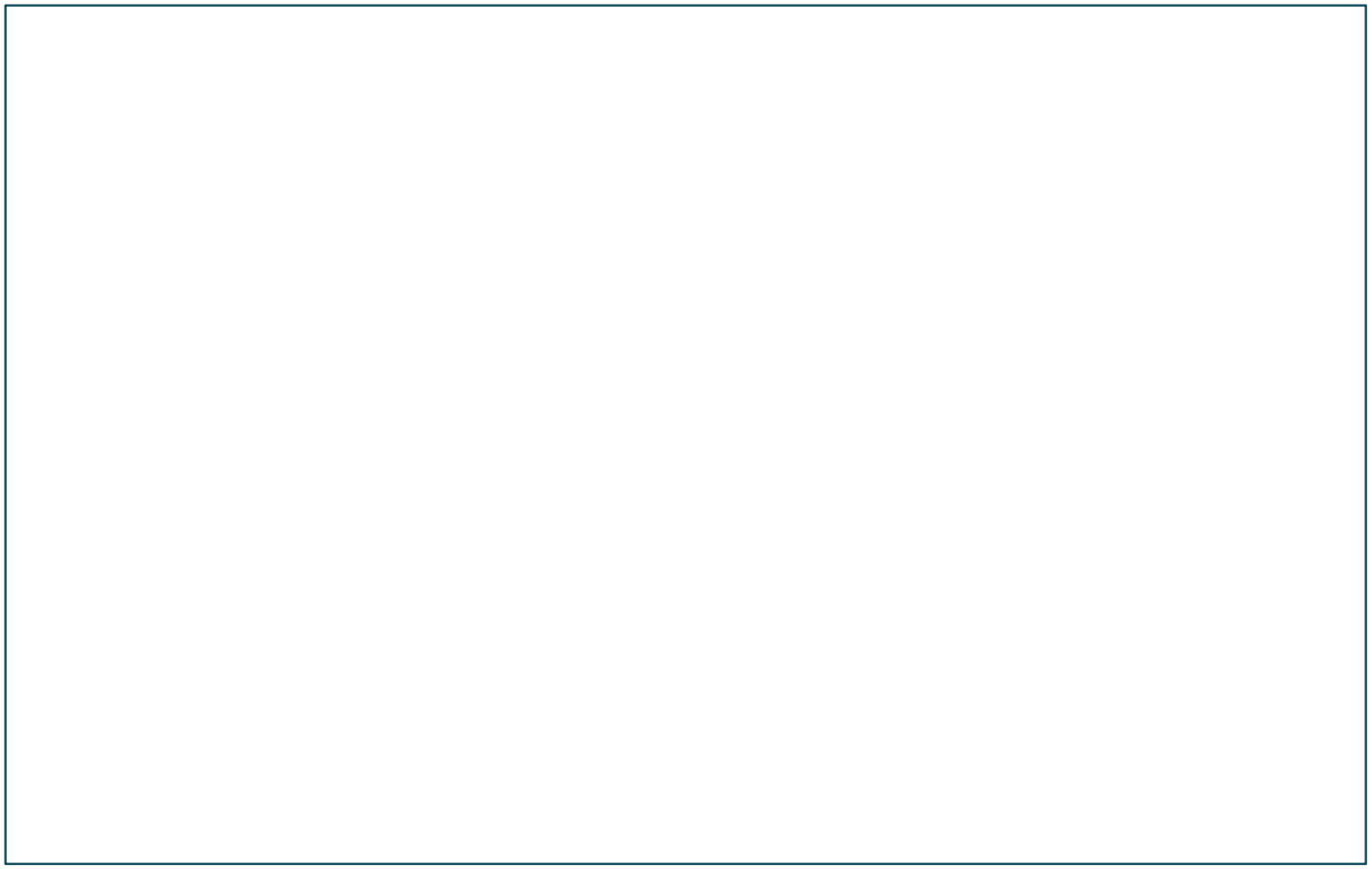
Quick Lab 22 – Activity 1 – Create a Reducer



In this part of the QuickLab, you will:

- Create a reducer function to handle state updates for Todos


useReducer Hook



USEREDUCER HOOK

useReducer can be used as an alternative to the **useState** hook.

It needs to be passed a reducer function.

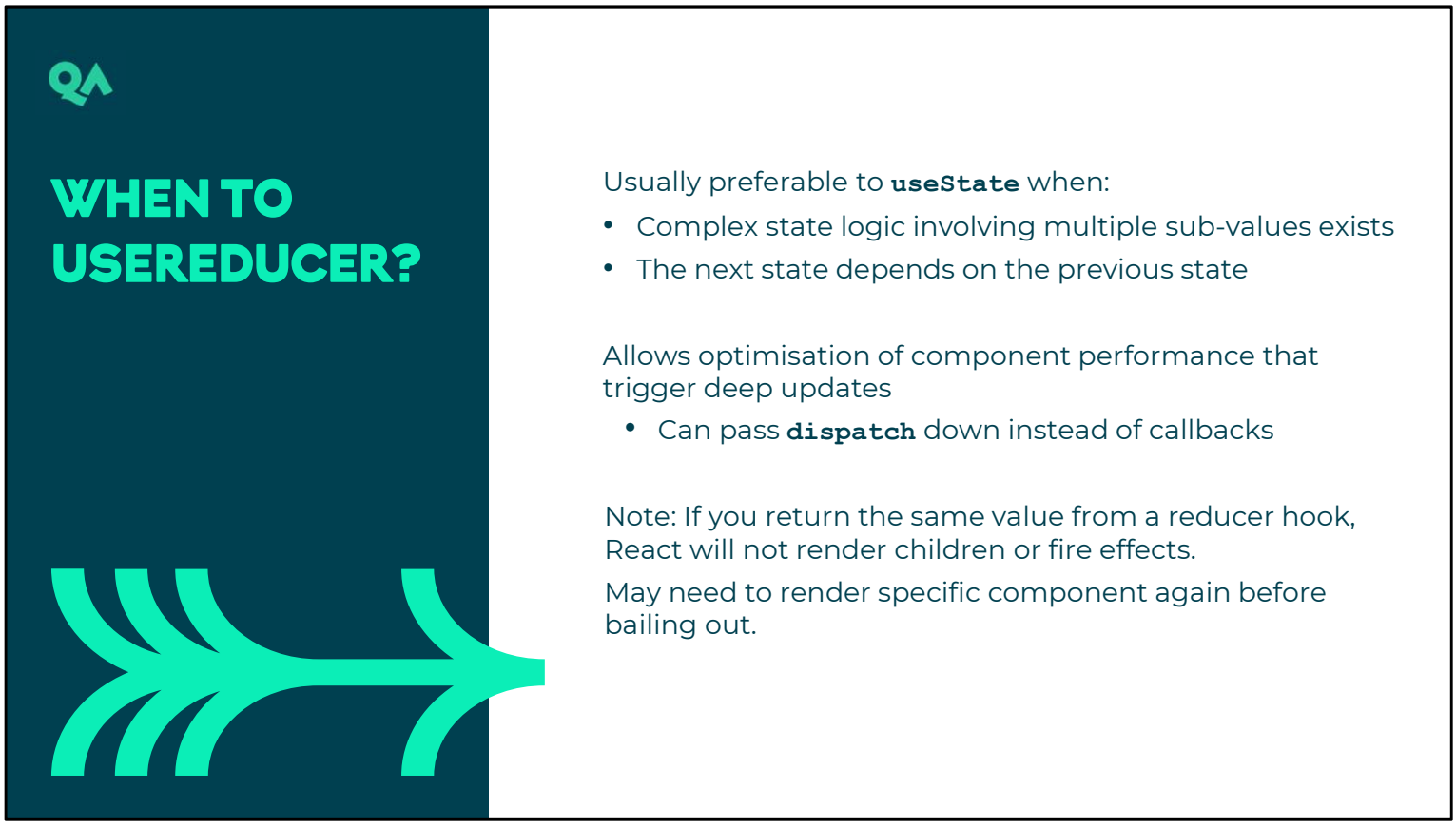
- Can optionally be passed an initial state

```
const [state, dispatch] =
  useReducer(myReducer, initialValue);
```

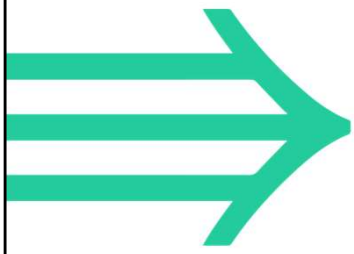
state is paired with a **dispatch** function that can be called

- **dispatch** is used to send **action** objects

```
const getData = async () => {
  const payload = await // some data call
  dispatch({type: `getData`, payload});
}
```


WHEN TO USEREDUCER?
Usually preferable to useState when:
• Complex state logic involving multiple sub-values exists
• The next state depends on the previous state
Allows optimisation of component performance that trigger deep updates
• Can pass dispatch down instead of callbacks
Note: If you return the same value from a reducer hook, React will not render children or fire effects.
May need to render specific component again before bailing out.

Other notable hooks



`useCallback` – Returns a memoized callback – only changes if one of the dependencies has changed.

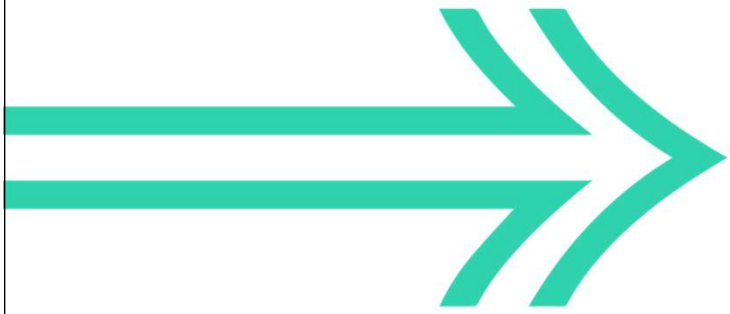
`useMemo` – Returns a memoized value – only recomputes memoized value when one of the dependencies changes – performance optimisation technique.

`useLayoutEffect` – identical to **`useEffect`** but fires synchronously after all DOM mutations – can read layout from DOM and synchronously re-render.

`useDebugValue` – can be used to display a label for custom hooks in the React Developer Tools.

- Not recommended to add to every hook – most valuable for custom Hooks that come as part of shared libraries

Quick Lab 22 – Activities 2-6 – Consume Dispatch Context and use it From a Submit



In this part of the QuickLab, you will:

- Create context for the dispatch and supply this to whoever wants it
- Consume the dispatch context where it is needed (i.e the TodoForm)
- Add routes back into app to allow adding and editing

Objectives

- To understand what context is and how it is used in React
- To be able to use the useContext hook to help simplify data drill-downs
- To understand what reducers are and how they are used in React
- To be able to use the useReducer hook to enable components to dispatch actions

POWERING POTENTIAL

Course Outcomes

By the end of the course, you will be able to:

- Create React components
- Perform some simple tests
- Think in React
- Add state and props to an application
- Add inverse data flow to an application
- Use some common React hooks
- Use external services to provide data
- Set up a single page application
- Use context and reducers

POWERING POTENTIAL

Hope you enjoyed this training course.

POWERING POTENTIAL

Please complete your evaluation

Code: QAREACTJS

PIN: