Vumbula React

A beginner's guide to ReactJS with ES6

First Edition

By Edmond Atto & John Kagga

Vumbula React

Edmond Atto and John Kagga

Vumbula React
by Edmond Atto and John Kagga

Editors: John Paul Seremba, Cecilia Caroline Nalubega
Production Editor: Bridget Mendoza
Cover Designer: John Paul Seremba

June 2019: First Edition.

Revision History for the First Edition:
2019-06-21 First release

Table of Contents

Preface

Who This Book Is For

This book is for someone that has some coding experience, we do not go into the basics of programming but jump straight into React. It would do you good to know a bit of Javascript before jumping into this book.

What's in This Book?

- In Chapter One, get introduced to React with ES6. If you are new to React, simply need a refresher, or need a gentle introduction to the ES6 features that are most frequently used throughout this book.
- In Chapter Two, get introduced to React components. They are the building blocks of any React Application you will build.
- In Chapter Three, get introduced to State in React. Understanding State and how it works will unlock your ability to build powerful components.
- In Chapter Four, get introduced to handling User Input in React. Understanding how to handle user input (primarily via forms) will unlock your ability to build interactive applications.
- In Chapter Five, get introduced to working with Routing in React. Create protected routes, nest routes and custom create routes.

We believe that by the time you have gone through this book, you will be well equipped to create a react project from scratch.

Conventions Used in This Book

This icon signifies a tip

This icon signifies a note, giving some information.

Getting and Using the Code Examples

The code used in this book can all be found here, https://github.com/vumbula/vumbula-react. It is organised on a chapter basis and should be easy to follow. There are links to the code in the different chapters.

This book is here to help you get your job done. In general, if the example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from this book does require permission. Answering a question by citing this book and quoting example code does not

require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

How to Contact Us

You can contact us

John Kagga: Twitter | LinkedIn | Email
Edmond Atto: Twitter | LinkedIn | Email
Bridget Mendoza: Twitter | LinkedIn | Email

Acknowledgements

The journey that led us to write this book has been a serendipitous one. We were two people who wanted to share our knowledge of React with the world. Initially, we had set out to write a series of articles on the subject but as we wrote more and more, we found that we had a lot more to say than articles would allow and so, the book *Vumbula React* was born.

This book would not have been possible without the support of Bridget Mendoza and her tireless efforts to get us across the finish line. When she happened upon us, all we had were a couple of standalone chapters of what we once dreamed would be a book. She took it upon herself to take up the production of what we present to you now as *Vumbula React*. Many thanks to her for being our Northstar throughout this process.

Chapter One
A Beginner's Guide to React With ES6

What is React?

React is a JavaScript library for building user interfaces, that is maintained by Facebook as an Open Source project under the MIT license. Over the past year or so, it has gained widespread popularity in the developer community as well as a large community of contributors. At the time of publishing this book, React was the **3rd** most starred library and/or Framework on Github (behind Bootstrap and Vue)with **131.099k** stars. It is safe to say then, that you deciding to learn React is a great decision.

It has to be emphasised that React is **NOT** a framework; it is a library! This is a common misconception many developers have as they start out on their React journey. As a library, React depends on other libraries such as React Router, Redux, Prop-types etc. to extend its already extensive capabilities.

React deals entirely with the look (user interface) of your web application, that is to say, it is purely presentational. React Native which is beyond the scope of this book, helps you to design user interfaces for Android and iOS apps. Refer to the official documentation for more information on React Native.

Is React really for me?

If you're interested in building modular interfaces for web applications, then the short answer is yes, React really is for you.

Because React is a JavaScript library, you will get the most out of this book if you know the basics of JavaScript, specifically the modern features of ECMAScript 6 (ES6 or JavaScript 2015) that we'll be using over the course of the book.

Do not worry if you do not know much about JavaScript. Towards the end of this introductory chapter, there is an overview of the JavaScript concepts that will be used in this book.

You are also encouraged to brush up on the basics of JavaScript from here and continue with this book when you're all caught up.

Why React?

React offers up a variety of benefits that have driven its rise in popularity. Let us review a couple of them real quick:

⇒ **Getting started is easy**

React is basically JavaScript, so, as long as you know its basics, you're good to go! The React API is quite simple to use and you will be able to create your first component with very limited markup.

⇒ **Easy DOM (Document Object Model) manipulation** In the past, using the actual DOM API was a pain, a fact that made it difficult for developers to manipulate the DOM. React solves this problem by providing a virtual DOM (in memory) that acts as an agent between the developer and the real DOM. The virtual DOM is a lot more user-friendly for developers.

⇒ **Speed**

Because of React's virtual DOM, it has a pretty cool way of handling changes to a web page; React is constantly listening for changes to the virtual DOM. It keeps a record of the actual DOM tree and when a change is detected on the virtual DOM, React calculates the differences between the two, it *reacts* to this change by making the changes and re-rendering only the elements on the DOM that have changed. This precision is what makes React lightning quick.

⇒ **React is declarative**

React allows you to describe what the application interface should look like as opposed to you describing how it should build the UI. React makes it such that you are not concerned with the details of how the different UI elements are created and rendered, giving you more time to think about what look you want to achieve with your interfaces as opposed to the tiny details of how to make that happen. Declarative programming is becoming more advanced and bringing more and more exciting features to the space. This post by Tyler Mcginnis is a good place to start finding more information on declarative programming vs imperative programming.

⇒ **React makes use of reusable components**

A component is simply a function/class that returns a section of your interface. Building an application with React allows you to reuse components in different sections of your application. It follows the pattern of creating a component once and declaring it in multiple locations as required. This helps you write a lot more maintainable code as it's easy to make cascading changes throughout your application.

⇒ **Unidirectional data flow**

React applications are built as a combination of parent and child components. As the names suggest, each child component has a parent and a parent component will typically have one or more child components. Components receive data via props and in the case of a parent and child component, props are passed down from the parent to the child. Data (props) is never passed up from the child to the parent, hence the phrase, unidirectional data flow. This is a powerful concept because it leads to a more predictable application and creates a single source of truth so that any changes to the parent's state propagate to all its children consistently.

⇒ **Powerful type-checking using PropTypes**

With the power of PropTypes, React allows you to protect your components from abuse (and catch bugs early) by strictly and efficiently enforcing type-checking on the props(props are to components what arguments are to functions) passed to them without the need to add the complexity that comes with using TypeScript or flow for type-checking in your project.

As of React 15.5.0, PropTypes is no longer part of the React core package but is used as a separate dependency.

⇒ **Large community** React's popularity ensures an ever-growing community around it, which means, there's a ton of resources out there to help you as you grow your skills. More importantly, packages such as React-router and React-redux that extend React's capabilities are actively maintained.

As you start out, these concepts may seem numerous and confusing; do not lose steam if everything doesn't make sense right now. During the course of reading the book, things will get a lot clearer. Reviewing earlier sections of the book as you go along will help you to further internalise the content.

Setting up your first React Application

There are numerous ways and tools out there to help you set up a React app, however, throughout this book, the official create-react-app CLI tool created by the Facebook team will be used. According to the official React documentation, create-react-app is the best way to start building a new React single page application. It sets up your development environment so that you can use the latest JavaScript features, provides a nice developer experience, and optimizes your app for production, all without requiring any configuration on your part.

Before the create-react-app CLI tool was created by the Facebook team, developers had to deal with setting up the applications with the right set of dependencies such as babel, react-dom as well as code linters. Additionally, they would have to create a custom webpack configuration file for every React app they worked on. With the CLI tool, developers can instantly jump into active development with minimal bootstrapping.

Before you can get started enjoying the create-react-app goodness, you'll need to make sure you have `Node >= 6` installed on your machine. If you do not have Node installed on your machine, no worries, this guide will have you set up in no time.

Create-react-app is available for Windows, Linux and MacOS so, you should be covered. Still not sure if you should use create-react-app? Reading this should help.

Enough talk, let's get you started and walk you through the simple process of creating a React app with create-react-app.

⇒ Install create-react-app globally

```
npm install -g create-react-app
```

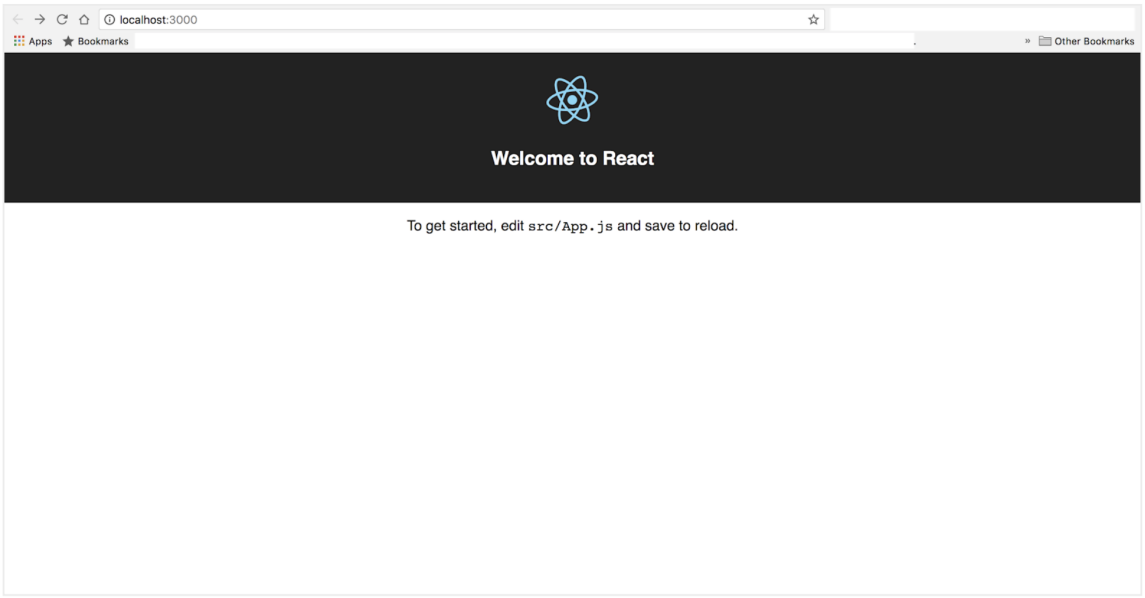
⇒ Run create-react-app by passing it the desired name of your app

```
create-react-app my-new-app
```

⇒ cd into your new app's directory and start your brand new app

```
cd my-new-app
npm start
```

⇒ Navigate to `localhost:3000` in your browser



Congratulations! You just created your first React application. You're well on your way to building bigger and better things.

Commonly used ES6 Features

Throughout the rest of this book, a number of ES6 features will be used consistently. If you do not have prior experience with ES6 features, this brief introduction will come in handy. If you're comfortable with ES6 features, skip this section and head to chapter 2 to get started writing your first component.

let and const

`let` and `const` are two new keywords that were introduced in ES6 for declaring variables. When used to declare variables, they are scoped to the block and not the function; this means they are only available within that block. Variables declared with let can be re-assigned but cannot be redeclared within the same scope whereas those declared by const must be assigned an initial value but cannot be redeclared within the same scope.

In summary, use `let` when you plan on re-assigning new values to the variable and `const` if you're not planning to re-assign a variable. See an example of using `let`

```
let name = 'Edmond';
name = 'Atto';
console.log(name);
```

Output

```
Atto
```

The spread operator

The spread operator denoted by ⋯ is used to expand iterable objects into multiple elements as shown in the example below.

```
const cities = ["Kampala", "Nairobi", "Lagos"];
console.log(...cities);
```

Output

```
Kampala Nairobi Lagos
```

The spread operator can also be used to combine multiple arrays into one array containing all array elements as shown below.

```
const east = ["Uganda", "Kenya", "Tanzania"];
const west = ["Nigeria", "Cameroon", "Ghana"];

const countries = [...east, ...west];
console.log(countries);
```

Output

```
[ 'Uganda', 'Kenya', 'Tanzania', 'Nigeria', 'Cameroon', 'Ghana' ]
```

Template literals

Before ES6, strings were concatenated using the `+` operator as shown in the example below.

```
const student = {
 name: 'John Kagga',
 city: 'Kampala'
};

let message = 'Hello ' + student.name + ' from ' + student.city;
console.log(message);
```

Output

```
Hello John Kagga from Kampala
```

ES6 introduced template literals which are essentially string literals that include embedded expressions. They are denoted by backticks instead of single or double quotes. The template literals can contain placeholders which are represented by `${expression}`. The quotes and `+` operator are dropped when using template literals as shown in the rewrite of the above example below.

```
let message = `Hello ${student.name} from ${student.city}`;
```

Output

```
Hello John Kagga from Kampala
```

Default function parameters

ES6 introduced a way of adding default values to the function's parameter list as shown below.

```
function greet(name = 'Fellow', greeting = 'Welcome') {
 return `${greeting} ${name}`;
}

console.log(greet());
console.log(greet('Kagga'));
console.log(greet('Mike', 'Hi'));
```

Output

```
Welcome Fellow
Welcome Kagga
Hi Mike
```

A default parameter is created when an equal (=) is added and whatever the parameter should default to if an argument is not provided (this parameter) can be any JavaScript data type.

Destructuring

In ES6, data can be extracted from arrays and objects into distinct variables using destructuring. Here are a couple of examples

1. Extracting data from an array

⇒ Before ES6

```
const points = [20, 30, 40];

const x = points[0];
const y = points[1];
const z = points[2];

console.log(x, y, z);
```

Output

```
20 30 40
```

⇒ With ES6

The above example can be changed to use destructuring in ES6 as shown below.

```
const points = [20, 30, 40];

const [x, y, z] = points;

console.log(x, y, z);
```

Output

```
20 30 40
```

The [ ] represent the array being destructured and x, y, z represent the variables where the values from the array are to be stored. You do not have to specify the array indexes because they are automatically implied. During destructing, some values can be ignored for example the y value can be ignored as shown below.

```
const [x, , z] = points
```

2. Extracting data from an object

⇒ Before ES6

```
const car = {
 type: 'Toyota',
 color: 'Silver',
 model: 2007
};

const type = car.type;
const color = car.color;
const model = car.model;

console.log(type, color, model);
```

Output

```
Toyota Silver 2007
```

⇒ With ES6

```
const car = {
 type: 'Toyota',
 color: 'Silver',
 model: 2007
};

const {type, color, model} = car;

console.log(type, color, model);
```

Output

```
Toyota Silver 2007
```

The `{ }` represent the object to be destructed and `type, color, model` represent the variables where to store the properties from the object. There is no need of specifying the property from where to extract the value from because `car` already contains a property called `type` and the value is automatically stored in the `type` variable.

As with array destructuring, object destructing enables extraction of only the values needed at a given time. The example below shows the extraction of only the `color` property from the car object.

```
const {color} = car;
console.log(color);
```

Output

```
Silver
```

Object literal Shorthand

ES6 provides a new way of initialising objects without code repetition, making them concise and easy to read. Prior to ES6, objects were initialised using the same property

names as the variable names assigned to them as shown below:

```
let type = 'Toyota';
let color = 'Silver';
let model = 2007;

const car = {
 type: type,
 color: color,
 model: model
};

console.log(car);
```

Output

```
{ type: 'Toyota', color: 'Silver', model: 2007 }
```

Looking closely at the above example, it is clear that `type:type,` `color:color` and `model:model` seem redundant. The good news is that you can remove those duplicate variable names from object properties if the properties have the same name as the variables being assigned to them as shown below.

```
let type = 'Toyota';
let color = 'Silver';
let model = 2007;

const car = {
 type,
 color,
 model
};
console.log(car);
```

Output

```
{ type: 'Toyota', color: 'Silver', model: 2007 }
```

Arrow functions

ES6 introduced a new kind of functions called arrow functions which are very similar to regular functions in behaviour but different syntactically.

As an example, follow the steps below to convert the given regular function into an arrow function.

```
function (name) {
  return name.toUpperCase();
}
```

- remove the function keyword
- remove the parentheses
- remove the opening and closing curly braces
- remove the return keyword
- remove the semicolon
- add an arrow (⇒) between the parameter list and the function body

The result

```
name => name.toUpperCase();
```

Using arrow functions

As opposed to regular expressions which can either be function declarations or function expressions, arrow functions are always expressions which can only be used where expressions are valid. Arrow functions can be stored in a variable, passed as an argument to a function or stored in an object's property.

Parentheses and arrow function parameters

If an arrow function parameter list has one element, there is no need for wrapping that element in parentheses.

```
name => `Hello ${name}!`
```

But, if there are two or more items in the parameter list or zero items, the list has to be wrapped in parentheses as shown below.

```
const hello = () => console.log('Hello React!'); //zero parameters
hello();

const location = (name, city) => console.log(`${name} is from ${city}.`);//two
parameters
location('John', 'kampala');
```

Block body syntax

When there is need to have more than one line of code in the arrow function body, the *block body syntax* has to be used. With the block body syntax, curly braces have to be used to wrap the function body and a `return` statement has to be used to actually return something from the function as shown below.

```
name => {
  name = name.toUpperCase();
  return `${name.length} characters make up ${name}'s name`;
};
```

Benefits of using arrow functions

Arrow functions may be preferred because of the following:-

- short syntax
- they are easy to write and read
- they automatically return when their body is a single line of code.

Classes

ES6 introduced classes that are simply a mirage that hides the fact that prototypal inheritance goes on under the hood. These classes are unlike those in class-based languages like Java. Below is an example of an ES6 class.

```
class Animal {
 constructor(numLegs) {
   this.numLegs = numLegs;
   this.mammal = false;
 }

 isMammal() {
   this.mammal = true;
 }
}
```

When a new object is constructed from the `Animal` class the constructor will run and the variables inside it will be initialised.

Benefits of using classes

With the new class syntax, less code is required to create a function. The function contains a clearly specified constructor function and all the code needed for the class is contained in its declaration.

ES6 also introduced two new keywords, `super` and `extends` which are used to extend classes.

Classes in javascript are still functions and their behavior is not the same as those in object-oriented programming languages such as Java.

This was a brief, high-level introduction to the ES6 features that will be used throughout the book. It is not meant as a replacement for any fully-fledged ES6 resources out there. Refer to this resource to learn more about ES6 features.

Chapter Two
Understanding React Components

What are components?

Components are the building blocks of any React app and a typical React app will have many of these. Simply put, a component is a JavaScript class or function that optionally accepts inputs i.e. properties(props) and returns a React element that describes how a section of the UI (User Interface) should appear.

Your first component

```
const Greeting = () => <h1>Hello World today!</h1>;
```

This is a **functional component** (called Greeting) written using ES6's arrow function syntax that takes no props and returns an `h1` tag with the text **"Hello World today!"**

In Chapter 1, you learnt how to set up a React App using the create-react-app tool. We'll take a step back momentarily and use a basic setup to learn the basics of components. You can find the starter app here and clone it to your computer.

In order to run the code examples in this chapter on your machine, you first have to install a server globally using nodeJs. Below is the command to install the `http-server` on your machine. Open your terminal and run:-

```
npm install http-server -g
```

Open the *index.html* file within the **Chapter 2/starter-code** folder in your text editor and add the *Greeting* component where you see the instructions to do so. Below is a code snippet of how your *index.html* file should look like after this change.

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>Get Started with Vumbula React</title>
</head>

<body>
<div id="root">
    Loading...
</div>
<script src="https://unpkg.com/@babel/standalone/babel.js"></script>
<script src="https://unpkg.com/react/umd/react.development.js"></script>
<script src="https://unpkg.com/react-dom/umd/react-
dom.development.js"></script>
<script type="text/babel">
  // Add your brand new component here
  const Greeting = () => <h1>Hello World today!</h1>
  ReactDOM.render(
      <Greeting />,
      document.getElementById('root')
  );
</script>
</body>
</html>
```

Within the **starter-code** folder run the command below to start the server:-

```
http-server .
```

Open the URL within the terminal in your browser and you should see the text "**Hello World Today!**".

In case you make changes to the code and they are not shown in the browser even after refresh. Try hard refreshing that tab or page.

You did it! You created and rendered your first component. Let's take a closer look to help us understand what just happened.

⇒ The React script allows us to write React components

⇒ The ReactDOM script allows us to place our components and work with them in the context of the DOM

⇒ The Babel script allows us to transpile ES6 to ES5. Some browsers have limited support for ES6 features; transpiling our ES6 to ES5 allows us to use the modern features of ES6 in our design without having to worry about compatibility. Notice that the React code is wrapped in a script tag with a type of `text/babel`.

```
ReactDOM.render(<Greeting />, document.getElementById('root'));
```

Translating the line of code above to English would sound something like this;

Use ReactDOM's render method to render the Greeting element into the DOM in a container with the id of root.

When naming a React component, it is convention to capitalize the first letter. This is important because it enables React to differentiate between the native HTML tags such as div, h1, span etc and custom components like `Greeting`.

A different way to write components

So far, you've written a functional component, a fitting name since it really was just a function. Components can also be written using ES6 classes instead of functions. Such components are called **class components**. Go ahead and convert the functional *Greeting* component to a class component like so:

```
class Greeting extends React.Component {
  render(){
    return <h1>Hello World Today!</h1>;
  }
```

Replacing the functional component in index.html with your new class component and refreshing your browser should also render "**Hello World Today!**" which means everything is working well.

Functional (Stateless) Vs Class (Stateful) components

By now, you've created both a functional and class component. In this section, we'll take a closer look at the differences as well as situations in which you might prefer to use one type over another.

Functional components

These components are purely presentational and are simply represented by a function that optionally takes props and returns a React element to be rendered to the page.

Generally, it is preferred to use functional components whenever possible because of their predictability and conciseness. Since they are purely presentational, their output is always the same given the same props. You may find functional components referred to as *stateless*, *dumb* or *presentational* in other literature. All these names are derived from the simple nature that functional components take on.

⇒ **Functional** because they are basically functions

⇒ **Stateless** because they do not hold and/or manage state

⇒ **Presentational** because all they do is output UI elements

A functional component in it's simplest form looks something like this:

```
const Greeting = () => <h1>Hi, I'm a dumb component!</h1>;
```

Class Components

These components are created using ES6's class syntax. They have some additional features such as the ability to contain logic (for example methods that handle onClick events), local state (more on this in the next chapter) and other capabilities to be explored in later sections of the book. As you explore other resources, you might find class components referred to as *smart*, *container* or *stateful* components.

⇒ **Class** because they are basically classes

⇒ **Smart** because they can contain logic

⇒ **Stateful** because they can hold and/or manage local state

⇒ **Container** because they usually hold/contain numerous other (mostly functional) components

Class components have a considerably larger amount of markup. Using them excessively and unnecessarily can negatively affect performance as well as code readability, maintainability and testability.

A class component in its simplest form:

```
class Greeting extends React.Component {
  render(){
    return <h1>Hi, I'm a smart component!</h1>;
  }
}
```

How do I choose which component type to use?

Use a class component if you:

⇒ need to manage local state

⇒ need to add lifecycle methods to your component

⇒ need to add logic for event handlers

Otherwise, **always** use a functional component.

As you start out, you will not always know whether to use class or functional components. Many times, you will realise after a while that you chose the wrong type. Do not be discouraged, making this choice gets easier as you create more components. Until then, one helpful tip is, class components that only have markup within the render body can safely be converted to functional components.

Props

In the previous chapter, having reusable components was listed as a benefit of using React, this is true because components can accept props and return a customised React element based on the props received.

Looking at the *Greeting* component you created earlier, it is clear that it's not a very useful component to have. In real-world situations, you will often need to render components dynamically depending on the situation. You, for example, might want the *Greeting* component to append your application's current user's name to the end of the greeting to have an output like "**Hello Steve**" as opposed to having it render "**Hello World Today!**" every time. Perhaps, you're always saying hello world, and the world never says hello back.

Props are React's way of making components easily and dynamically customisable. They provide a way of passing properties/data down from one component to another, typically from a parent to a child component (unidirectional dataflow).

It's important to note that props are **read-only** and that a component must **never** modify

the props passed to it. As such, when a component is passed props as input, it should always return the same result for the same input.

All React components should act like pure functions with respect to their props.

Now that you know about props, make use of them in the *Greeting* component to render a greeting with a custom name appended to it.

Make changes to the code between the script tags in your *index.html* document to make it look like this:

```
const Greeting = props => <h1>Hello {props.name}</h1>;
   ReactDOM.render(
     <Greeting name={'Edmond'}/>,
     document.getElementById('root')
   );
```

This renders the text "**Hello Edmond**" to the screen. Go ahead and play around with this by switching out the name for yours.

Using props added some new syntax to your app. Let's take a closer look and understand what is going on here.

⇒ An **argument** (props) is passed to the functional component. Recall that since a single argument is being passed to the arrow function, the parentheses are unnecessary. Passing this argument lets the component know to expect some data to be passed to it (in this case, the name of our app's user)

⇒ Within **ReactDOM.render**, the name you want to be rendered to the screen is passed in by specifying `propName={propValue}` within the component's tag.

⇒ In the `h1` tag, `{}` are used to print the name that is added to the props object when it's passed in via the component's tag. Notice that the name attribute is accessed using the dot syntax.

There is no limit to how many *props* can be supplied to a component.

Using Props with Class Components

Adding props to class components is a very similar process to the one used in the functional component above. There are two notable changes:

⇒ Props is not passed as an argument to the class

⇒ The `name` attribute is accessed using `this.props.name` instead of `props.name`

```
class Greeting extends React.Component {
     render(){
     return <h1>Hello {this.props.name}</h1>;
     }
   }
ReactDOM.render(
     <Greeting name={'Edmond'}/>,
     document.getElementById('root')
   );
```

Challenge

*Make changes that make it possible for the **Greeting** component to take **name**, **age** and **gender** props and render this information to the page.*

HINT: *Pass 3 attributes (name, gender and age) to your component within* `ReactDOM.render()` *and alter your* `h1` *text to accommodate your new data. Remember to access the attributes using the right syntax e.g.* `props.gender` *for functional components and* `this.props.gender` *for class components*

Default props

These offer another way to pass props to your component and as the name suggests, default props are used by a component as default attributes in case no props are explicitly passed to the component.

As a fallback, default props are helpful in enabling you offer a better user experience through your app, for example, considering the *Greeting* component from previous examples, using default props ensures that a complete greeting is always rendered even if the name attribute has not been explicitly passed to the component.

```
// Greetings Component
const Greeting = props => <h1>Hello {props.name}</h1>;

// Default Props
Greeting.defaultProps = {
  name: "User"
};

ReactDOM.render(
    <Greeting/>,
    document.getElementById('root')
);
```

By altering the *Greeting* component, as shown above, you now have "**Hello User**" being rendered in your browser if you do not pass the name attribute to the component.

*Passing a `name` attribute as a prop to the **Greeting** component overwrites the default props.*

Composing Components

Up until now, you've only created a single component, however, when building real products, you will often have to build multiple components.

React allows you to reference components within other components, allowing you to add a level(s) of abstraction to your application.

Take for example a user profile component on a social network. We could write this component's structure like so:

```
UserProfile
    |-> Avatar
    |-> UserName
    |-> Bio
```

In this case, *UserProfile*, *Avatar*, *UserName* and *Bio* are all components. The *UserProfile* component is composed of the *Avatar*, *UserName* and *Bio* components. This concept of component composition is quite powerful as it enables you to write highly modular and reusable components. For example, the *UserName* component can be used in many parts of the web application and in case it ever needed to be updated, changes would only be made to the *UserName* component and the changes would reflect everywhere with the application where it is used.

```
//Avatar component
const Avatar = () => <img src="http://i.pravatar.cc/300"/>;

//Username component
const UserName = () => <h4>janedoe</h4>;

//Bio component
const Bio = () =>
    <p>
      <strong>Bio: </strong>
      Lorem ipsum dolor sit amet, justo a bibendum phasellus proodio
      ligula, sit
    </p>;

// UserProfile component
const UserProfile = () =>
    <div>
      <Avatar/>
      <UserName/>
      <Bio/>
    </div>;

ReactDOM.render(
    <UserProfile/>,
    document.getElementById('root')
);
```

In the code snippet above, the *Avatar*, *UserName* and *Bio* components are defined within the *UserProfile* component. Try and do this on your own using the *index.html* file from previous examples.

> ***Functional components** can be referenced within **class components** and vice versa. However, it is not often that you will reference a class component within a functional component; class components typically serve as container components.*

Project One

At this point, you have learned enough of the basics. Get your hands dirty by following up with this first project and in case you get blocked, get out the Github repository for this chapter for the solution.

Let's get started

Clone the repository and `cd` into the **chapter 2** folder that contains the code for this chapter. Then fire up a text editor or IDE of your choice, though VSCode or Webstorm are recommended and follow the steps below.

- Create a *project folder* to hold the project files.
- Create an *index.html* page
- Create a *src folder* to hold the JavaScript files.
- Create an *index.js* file within the src folder
- Add a `div` with an *id* of root to the body of the *index.html*.
- Add the react, react-dom and babel scripts
- Link to the *index.js* script below the babel script at the bottom of the html page.
- Within *index.js* create a presentational component called *Application*.
- Copy all the html within the body of the html template in the *starter-code* folder within the *chapter 2* folder apart from the script tags.
- Paste this html within the `<> </>` tags (fragments). We use tags because a React component only accepts one element and all the rest/siblings must be nested within the one parent element.
- We need to clean up the html code and turn it into JSX that React can understand.
- Let us start by removing all the html comments like this one `<!-— Navbar -→`
- Rename all **class** instances to **className**, then close all img tags like so `<img className="card-img-top" src="img/8.png"/>` and also close the horizontal rule `<hr/>`

> ***class** is a reserved name in React, hence, the requirement to change all class instances to **className**.*

- Copy the *img* folder and paste it at the root of the project folder.
- Head back to the *index.html* file and add the *Bootstrap 4 CSS* link tag in the head section.
- Whoop…Whoop…You can now open the html page in the browser and see your new React application.

> *Here is the code up to this point.*

Great work so far, you are moving on well but you are not yet done. You need to break the main component down further into smaller components so that your code is clean and easy to maintain. Let us get back to work.

Into components

Start by creating the *Nav* component. You can try it on your own and then cross-check your work by reading through the steps below.

- Below the *Application* component, create a new functional component with a name Nav.
- Copy the `<nav> </nav>` JSX into the *Nav* component as shown below.

```
const Nav = () =>
  <nav className="navbar fixed-top navbar-expand-lg navbar-dark bg-primary">
    <div className="container">
      <button className="navbar-toggler"
              type="button"
              data-toggle="collapse"
              data-target="#navbarNavAltMarkup"
              aria-controls="navbarNavAltMarkup"
              aria-expanded="false"
              aria-label="Toggle navigation">
        <span className="navbar-toggler-icon"></span>
      </button>
      <div className="collapse navbar-collapse"
        id="navbarNavAltMarkup">
        <div className="navbar-nav">
          <a className="nav-item nav-link"
             href="#home">Home
            <span className="sr-only">(current)</span>
          </a>
        </div>
      </div>
    </div>
  </nav>;
```

- Delete the nav code from the *Application* component JSX and replace it with `<Nav/>` element as shown here.
- Open up the application again in the browser and everything should still be the same.
- Let's move on to the second presentational component, the *Jumbotron*.
- Create an arrow function with a name `Jumbotron`.
- Copy the *jumbotron* code and paste it into the `Jumbotron` function.
- Delete the jumbotron code from the *Application* component's JSX and replace it with the `<Jumbotron/>` element.

It is now your turn, go on and create the *Toys* and *Footer* functional components and then reference them within the *Application* component. Be sure to follow similar steps as before.

Nothing about the page in the browser should change after you are done. When you are done cross-check your solution with this.

We have done a good job up to this point, you may have realized that our JSX is not DRY. Meaning there is a lot of repetition specifically in the `<Toys/>` component, the toy cards are repeated for every toy. We can leverage the power of reusability that React components possess to clean this up.

- First, create an array called toys to hold objects containing the toy name, description and image number as shown below.

```
const toys = [
  {
    name: 'Toy One',
    description: `Lorem Ipsum is simply dummy text of the printing
   and typesetting industry. Lorem Ipsum has been
   the industry's standard dummy text ever since the 1500s.`,
    image: '1'
  },
  {
    name: 'Toy Two',
    description: `Lorem Ipsum is simply dummy text of the printing
   and typesetting industry. Lorem Ipsum has been
   the industry's standard dummy text ever since the 1500s.`,
    image: '2'
  },
  {
    name: 'Toy Three',
    description: `Lorem Ipsum is simply dummy text of the printing
   and typesetting industry. Lorem Ipsum has been
   the industry's standard dummy text ever since the 1500s.`,
    image: '3'
  },
  {
    name: 'Toy Four',
    description: `Lorem Ipsum is simply dummy text of the printing
   and typesetting industry. Lorem Ipsum has been
   the industry's standard dummy text ever since the 1500s.`,
    image: '4'
  },
  {
    name: 'Toy Five',
```

```
    description: `Lorem Ipsum is simply dummy text of the printing
   and typesetting industry. Lorem Ipsum has been
   the industry's standard dummy text ever since the 1500s.`,
    image: '5'
  },
  {
    name: 'Toy Six',
    description: `Lorem Ipsum is simply dummy text of the printing
   and typesetting industry. Lorem Ipsum has been
   the industry's standard dummy text ever since the 1500s.`,
    image: '6'
  },
  {
    name: 'Toy Seven',
    description: `Lorem Ipsum is simply dummy text of the printing
   and typesetting industry. Lorem Ipsum has been
   the industry's standard dummy text ever since the 1500s.`,
    image: '7'
  },
  {
    name: 'Toy Eight',
    description: `Lorem Ipsum is simply dummy text of the printing
   and typesetting industry. Lorem Ipsum has been
   the industry's standard dummy text ever since the 1500s.`,
    image: '8'
  },
];
```

- Create a functional component that accepts props as an argument and name it Card. Copy and paste one card's JSX code into it from the *Toys* component as shown below.

```
const Card = props =>
    <div className="col-md-6 col-lg-3">
      <div className="card mb-3">
        <img className="card-img-top" src={`img/${props.toy.image}.png`}/>
        <div className="card-body">
          <h4 className="card-title text-center">{props.toy.name}</h4>
          <p className="card-text">
            {props.toy.description}
          </p>
        </div>
      </div>
    </div>;
```

- We need to make a few changes to the card so that it is reusable by adding

placeholders which will be replaced by the actual data to be rendered. *The toy name is replaced by `{props.toy.name}` where `toy` is a prop object passed into the component from the toys array.

- The description is replaced by `{props.toy.description}`.
- The image src is replaced by a string template literal which accepts `{props.toy.image}` to make up the image path.
- Let us make use of our new *Card* component by refactoring our *Toys* component. First, delete all the cards within a div with a **class** of `row` in the *Toys* component. Then change the function signature to accept *props* as its only argument.
- In order to display all the cards again in the *Toys* component, we make use of the map function to loop through the toys array passed into it as props. This *map* function accepts a callback function that accepts two arguments, the **item** in the array and its **index**. This callback returns a *Card* component which accepts a **toy** has its props. React also requires us to add a key to elements that are being looped over so that it can easily keep track of them and the changes applied to them making it easy for it to know what elements to re-render when the underlying data changes. Therefore the index of the toy object within the array acts as the key in this case as shown below.

```
const Toys = props =>
    <>
      <h1 id="toys"
          className="display-4 my-4 text-center text-muted">Toys
      </h1>
      <div className="row">
        {props.toys.map((toy, index) => <Card key={index} toy={toy}/>)}
      </div>
    </>;
```

- Before you can test out the changes there is one more thing to do, otherwise, the toys won't show on the page.

We need to pass in the `toys` array as props to the *Application* component so that the *Toys* component can get access to them. This is shown in the snippet below.

```
ReactDOM.render(
<Application toys={toys}/>,
document.getElementById('root')
);
```

Finally, within the *Application* component, we also need to pass the toys array as props down to the *Toys* component as shown below. Recall from chapter one that data flow in React is unidirectional.

```
<Toys toys={this.props.toys}/>
```

Now open the page again in the browser to view the changes we have made. You will realize that nothing changes in the browser, we still get to see our page design as it was, but now it is fully optimized with React.

At this point, you know how components work and how you can make use of them to develop modular React code that represents different sections of your user interface. Here is the final code for this project. The next chapter explains the aspect of state in a React application, do not miss it.

Chapter Three
Understanding State in React

In earlier chapters, we've dealt mostly with functional React components that do not require state management. This chapter's main focus is on state, its management and components that utilise it in React.

What is State?

State is a JavaScript object that stores a component's dynamic data and determines the component's behaviour. Because state is dynamic, it enables a component to keep track of changing information in between renders and for it to be dynamic and interactive.

State can only be used within a class component. If you anticipate that a component will need to manage state, it should be created as a class component and not a functional one.

State is similar to `props` but unlike `props`, it is private to a component and is controlled solely by the said component. In the examples from previous chapters, the behaviour of components has primarily depended on the `props` that are passed down to them. In those cases, the components that receive the props have no control over them because props are read-only.

In **Project One** from Chapter 2, `toys` were passed as `props` to the *Application* component, and then down to the *Toys* component. For the *Toys* component to gain control over the `toys` data, it should first be converted into a class component and the `toys` data should be added into `state`.

It is worth mentioning that state in React is **immutable**, that is to say, state should never be altered/changed directly but rather, changes should be made to a copy of the current version of the state. This has benefits such as providing the ability to review the state at different points in time and for apps to hot reload (automatic reloading of the page in the browser when you make changes in the code).

Adding State to a Class Component

```
class Greeting extends React.Component {
  render(){
    return <h1>I'm a component in need of some state!</h1>;
  }
}
```

Adding state to the *Greeting* component above involves defining within the class component, a constructor function that assigns the initial state using `this.state`.

```
class Greeting extends React.Component {
  constructor(props) {
   super(props);
     // Define your state object here
     this.state = {
       name: 'Jane Doe'
     }
   }
   render(){
     return <h1>Hello { this.state.name }</h1>;
   }
}
```

Notice that the constructor accepts `props` as an argument, which are then passed to `super()`. Adding `super()` is a **must** when using the constructor.

Passing `props` is not necessary unless you are making use of them in the component. From the *Greeting* component above, it's not necessary to pass `props` to either the `constructor` or `super()`, that is to say, the component can be written like so:

```
class Greeting extends React.Component {
  constructor() {
    super();
    // Define your state object here
  }
  // Define your render method here
}
```

However, the React docs recommend that you **always** pass `props` in order to guarantee compatibility with potential future features

State is accessed using `this.state` as seen in the *Greeting* component's `h1` tag.

State is initiated using `this.state`, however, all subsequent changes to state are made using this.setState. Using this.setState ensures that the components affected by the change in state are re-rendered in the browser.

Investigating State using React Developer tools

One way to accelerate your understanding of React is to make use of the React devtools created by the team at Facebook. The power of React devtools is most apparent when you need to debug your React app by doing a deep dive into the code. The tools enable you to investigate how React is working below the surface when the app is rendered in the

browser.

Installing the React Developer tools

The devtools are available for download on both Mozilla Firefox Add-ons and the Chrome Web Store. Follow the appropriate link to install the devtools depending on which browser you have installed on your computer.

Throughout the rest of this book, Chrome will be used as the browser of choice. In order to confirm successful installation of the devtools on Chrome, open the developer tools window using `Cmd+Opt+I` on a `Mac or Ctrl+Alt+I` on a windows PC. You should now see a React tab.

Using the React Devtools

With the *Greeting* component from earlier in this chapter rendered in your browser, open the developer tools and navigate to the React tab. You should see something similar to this

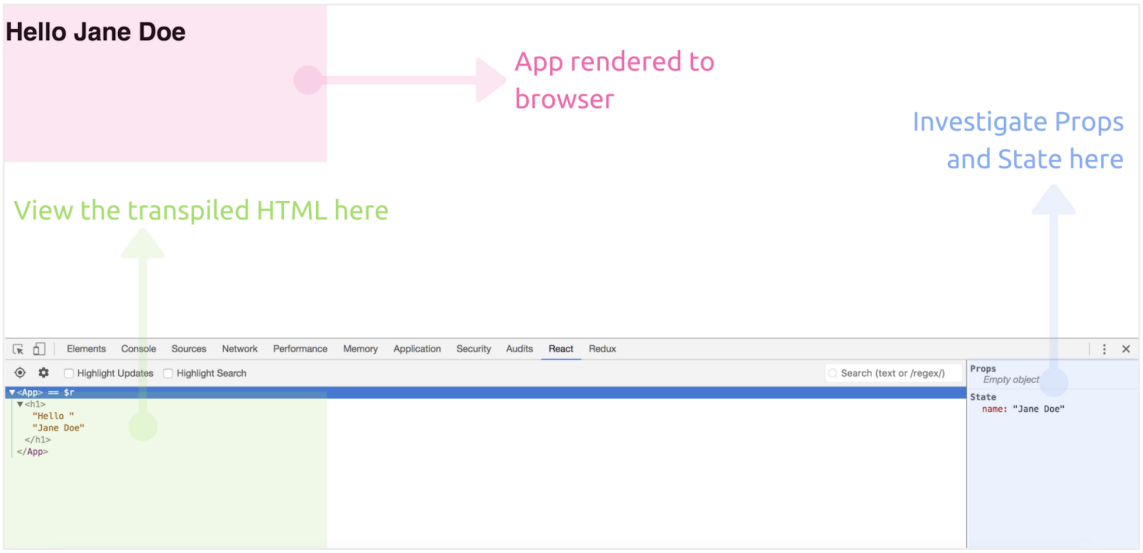


Figure 3-1: ReactDev Tools

Mastery of the React DevTools will enable you to gain a better understanding of React's inner workings and to quickly debug React applications.

Project Two

To better understand the basic use cases of state in React, we shall build a simple application that allows us to create and render records.

What we'll build

A React app that enables us to keep track of our friends' `names` and `ages`. The app provides

a form that we shall use to enter their details. It then renders our friends' details in beautiful Bootstrap 4 cards.

The finished application looks like this:

Figure 3-2: Finished Application

Getting started

Download or clone the projects' starter files from the repository to your computer so that you can follow along.

In order to run the code examples in this chapter on your machine, you have to first install a server globally using NodeJS. Below is the command to install the `http-server` on your machine. Open your terminal and run:-

```
npm install http-server -g
```

After the installation is done, `cd` into the **Chapter 3** folder then the **starter-code** folder. Within there run the command below to start the server:-

```
http-server .
```

In case you make changes to the code and they are not shown in the browser even after a refresh, try hard refreshing that tab or page.

Below is what will be shown in the browser when you open the localhost url displayed in your terminal.

Figure 3-3: Current Application state

Inside `src/index.js`, there's a simple class component that renders JSX for a form with `name` and `age` fields, and a `save` button.

Adding state to the component

In order to display the `names` and `ages` added to the application, we need to add state to our *Application* component. We'll start by adding default state which contains a dummy name and age which will display whenever the page is rendered in the browser.

We do this by initiating state with `this.state` inside the component's `constructor` method like so:

```
constructor(props) {
  super(props);
  this.state = {
    data: [
      {
        name: 'John',
        age: 20
      }
    ]
  }
}
```

If you are following along, copy and paste the snippet into the *Application* component just before the render function.

Rendering data from state

To render the state data, a *Card* presentation component is defined with the functionality to display the `name` and `age` from the props passed to it as shown below.

```
const Card = props =>
  <div className="col-md-6 col-lg-3">
    <div className="card mb-3">
      <div className="card-body">
        <p className="card-title">
          <span>Name: </span>{props.info.name}
        </p>
        <p className="card-text">
          <span>Age: </span>{props.info.age}
        </p>
      </div>
   </div>
 </div>;
```

Add this *Card* component to the *index.js* file below the *Application* component but before the `ReactDOM` code.

To display the data in state, we need to access the data array using `this.state.data` and then use JavaScript's `map` function to loop through the array so that each of its elements is rendered on the page.

```
<div className="row">
  {
    this.state.data.map(info => <Card info={info}/>)
  }
</div>
```

The statement containing the *Card* component is wrapped within a Bootstrap row so that it is displayed within the Bootstrap grid and placed just after the second `<hr/>` within the class component's `render` function.

A card is then displayed in the browser as shown below.

Figure 3-4: Card displayed in browser

Checking the console within the developer tools window reveals errors as shown in the screenshot below.

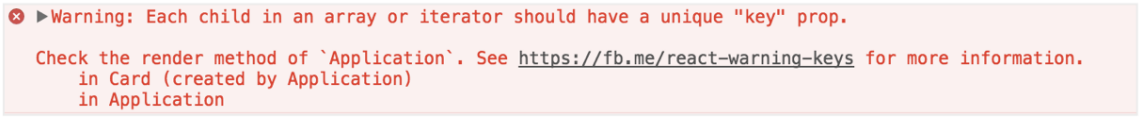


Figure 3-5: Errors in console

This means that we need to give each `Card` element a `key` so that React can identify each `Card` and know what to do when changes occur to any one of them. This can easily be fixed using the `map` function.

The `map` function accepts a function that accepts two arguments, the array element (`info`) and its `index`; this means that we can use the `index` as a key to the *Card* component.

Alter the code to match the code within the snippet below.

```
<div className="row">
  {
    this.state.data.map(
      (info, index) => <Card key={index} info={info}/>
    )
  }
</div>
```

This should clear the error in the console.

Using `index` as a `key` in a map function typically works well for small applications whose data is not that dynamic. However, as applications and data sources get larger, using the `index` as a `key` becomes unreliable. In these cases, it's recommended to use a truly unique `key`, for example, an `id`. In the project above, every object in state can be assigned an `id` field and this `id` can then be used as the key like so; `key={info.id}`.

Adding form data to state

The application is incomplete without the functionality to add new `names` and `ages` via the form. This requires knowledge of **handling user input**, a topic that is covered in the next chapter.

Chapter Four
Handling User Input in React

Majority of the applications you will build will have the bulk of their functionality centered around detecting and responding to user input. This could be via a click or more likely, a form. This chapter will focus on forms and making sure you have a good understanding of how forms are used in React.

Forms are the most common way to receive input from a user, for example, forms are used to collect users' login details. When the user clicks the login button, these details are submitted and they may then be handed over to the application's backend (authentication) service for processing. Depending on whether the login was successful or not, the frontend is updated accordingly. Consequently, forms also make it possible for users to update already existing information such as their username on a social media site when they believe they've found a cooler one.

When working with forms in React, two types of components are typically used:

⇒ Controlled components

⇒ Uncontrolled components

Controlled Components

HTML form elements are unique because, by default, they maintain some internal state. Specifically, form elements such as the `<input>` and `<textarea>` maintain and update their own internal state. For this reason, we have to think more carefully about how we use them in React.

In chapter 3, it is pointed out that a component's mutable data is stored in its state property. It makes sense then, to combine the HTML forms' *"natural"* abilities with React's state to make React's state the singular data source.

This combination creates a situation where the component that renders a form also **controls** what action is taken upon user input. For this reason, such a component is called a **controlled component**. Inputs that live inside controlled components are known as **controlled inputs**.

Here's an example of a controlled component that renders a login form. For demonstration purposes, we shall use the `username` field.

```
class LoginForm extends React.Component {

 constructor(props){
   super(props);
   this.state = { username: '' };
 }

 handleChange = event => {
   this.setState({ username: event.target.value });
 };

 render() {
   return (
     <React.Fragment>
       <form>
         <label htmlFor="username">username</label>
         <input
           type="text"
           name="username"
           value={this.state.username}
           onChange={this.handleChange}
         />
       </form>

       <h3>Your username is: {this.state.username}</h3>
     </React.Fragment>
   );
 }
}
```

In the above example, the *LoginForm* component is set up with a state object containing a `username` property. This property will hold the value/text entered by the user.

Initially, there is no text displayed in the input field because its value attribute is set to `this.state.username` which is initialised with username set to an empty string.

When the user clicks on the input field and starts typing, each keystroke triggers the `onChange` event handler. The `handleChange` function is then called and the current value *(text)* in the input is saved to state using `setState()`.

`setState()` causes the component to rerender and the text displayed in the input field is now fetched from `this.state.username`. The text in the `h3` is also updated upon the re-render.

This flow ensures that the `input`, `h3` and `state` are always in sync since the state object is

the single source of truth for the component.

Using controlled components ensures that:

⇒ the inputs (username field in this example) and the data (state) are always in sync

⇒ the UI (`h3` tag in this example) and the data (state) are always in sync.

Working with multiple inputs

It is unlikely that an application will have a form with just one field. Let's add an extra field to the *LoginForm* component from before to explore how to implement multiple controlled inputs with minimal markup.

```
class LoginForm extends React.Component {

 constructor(props){
   super(props);
   this.state = { username: '', password: '' };
 }

 handleChange = ({ target }) => {
    this.setState({ [target.name]: target.value });
 };

 render() {
   return (
     <React.Fragment>
       <form>
         <label htmlFor="username">username</label>
         <input
           type="text"
           name="username"
           value={this.state.username}
           onChange={this.handleChange}
         />
         <label htmlFor="password">password</label>
         <input
           type="password"
           name="password"
           value={this.state.password}
           onChange={this.handleChange}
         />
       </form>
       <h3>Your username is: {this.state.username}</h3>
     </React.Fragment>
   );
 }
}
```

In order to use a single `handleChange` function for multiple inputs, each input field is given a name attribute. The `handleChange` function is altered to perform a different action depending on the input `target`. Here, we use the power of ES6's computed property name `[name]: value` to update the state key corresponding to a particular input's `name` attribute.

Uncontrolled Components

In **uncontrolled** components, form data is handled by the DOM, unlike controlled components in which the form data is handled by a React component.

Uncontrolled components leverage the fact that HTML form elements maintain their own internal state. When dealing with uncontrolled inputs, state management via a React component is not required.

In uncontrolled components, form data is accessed using **refs**. Think of a **ref** as a tag that you receive when you check your bag in at the airport (just go with it). When your flight lands, you present your tag which serves as a reference to which bag is yours. The person at the bag desk takes your tag and returns minutes later with your bag. Similarly, HTML forms know which data belongs to which input field and by assigning an input a ref, you can then retrieve its value later.

In this analogy, you can only retrieve your bag after the flight has landed. Similarly, you can only use refs to fetch form data **after** a form has been submitted.

Here is an example of an uncontrolled component.

```
class LoginForm extends React.Component {

 handleSubmit = event => {
   event.preventDefault();
   alert('Your username is: ' + this.input.value);
 };

 render() {
   return (
     <form onSubmit={this.handleSubmit}>
       <label htmlFor="username">username</label>
       <input
         type="text"
         name="username"
         ref={(input) => this.input = input}
       />
     </form>
   );
 }
}
```

In this example, notice that the input has a `ref` attribute. The input element is passed as input to the arrow function and is then assigned to `this.input`.

When the form is submitted, the `handleSubmit` function is fired and at this point, the text entered by the user can be accessed using `this.input.value`.

Using Default Values in Controlled Components

In cases where the user needs to update an already existing value, for example, a profile update, the input field should display the pre-existing value and remain editable.

During a React component's render lifecycle, a form's `value` attribute will always override the `value` attribute in the DOM. Consequently, you can use React to set the initial value and leave subsequent updates as uncontrolled.

```
class LoginForm extends React.Component {

 handleSubmit = event => {
   event.preventDefault();
   alert('Your username is: ' + this.input.value);
 };

 render() {
   return (
     <form onSubmit={this.handleSubmit}>
       <label htmlFor="username">username</label>
       <input
         type="text"
         name="username"
         defaultValue="cool-guy"
         ref={(input) => this.input = input}
       />
     </form>
   );
 }
}
```

In the *LoginForm* component above, the input field initially renders with the text cool-guy because of the value passed to the `defaultValue` attribute. Alternatively, a value from state can be passed in here.

Upon submission of the form, the input field's value attribute overrides the defaultValue.

Controlled Vs Uncontrolled

Using controlled components is widely viewed as the preferred way to work with forms in React. This because they are more powerful than uncontrolled components and offer a number of benefits, that is to say:

⇒ The inputs, data and UI are always in sync

⇒ They allow for instant field validation

⇒ They allow for custom input formatting before submission, for example, converting all entered email addresses to lowercase before form submission

That said, using controlled components where forms with numerous input fields are involved can be tedious. This is because you would be required to write `onChange` handlers covering every possible way your data can change and channel all input data through a React component.

In situations where the form you are dealing with is relatively simple and only requires submission of user data with no dynamic UI updates during user input, or input formatting before submission, uncontrolled components could be a better choice.

Project Two (Continued)

You now know enough about forms to build out the rest of the features for project three from the previous chapter.

Adding form data to state

It is time to use the form on top of the page to add `names` and their corresponding `ages`. To do this, some changes need to be made to the class component, particularly to the form element.

In order to get the `name` and `age` entered by the user, we need to add a `ref` attribute to the `name` and `age` input elements. The `ref` attribute accepts a callback which receives the underlying DOM element as its argument.

The `ref` callback is then used to store a reference to the text input of the DOM element within an instance variable, in our case the instance variables are the `name` and `age` as shown in the code snippet below.

```
<form className="form-inline">
 <input
     type="text"
     className="form-control mb-2 mr-sm-2 mb-sm-0"
     placeholder="Name"
     ref={input => this.name = input}/>
 <div className="input-group mb-2 mr-sm-2 mb-sm-0">
   <input
     type="text"
     className="form-control"
     placeholder="Age"
     ref={input => this.age = input}/>
 </div>
 <button
     type="submit"
     className="btn btn-primary">Save
 </button>
</form>
```

Add the `ref` lines to your name and age input elements to match the code shown above.

Now that we have a reference to the text entered by the user, we need to add it to state when the save button is clicked. To do this effectively, we need to add an `onSubmit` event handler to the form element which will be called when the **save** *(submit)* button is clicked. This `onSubmit` handler attribute expects a function which will be executed when the **save** button is clicked.

Therefore, define an arrow function with a name of `onSubmit` that accepts event as its only argument within the *Application* component. Within the `onSubmit` function prevent the default button behaviour (of reloading the page when it is clicked) by adding `event.preventDefault()`.

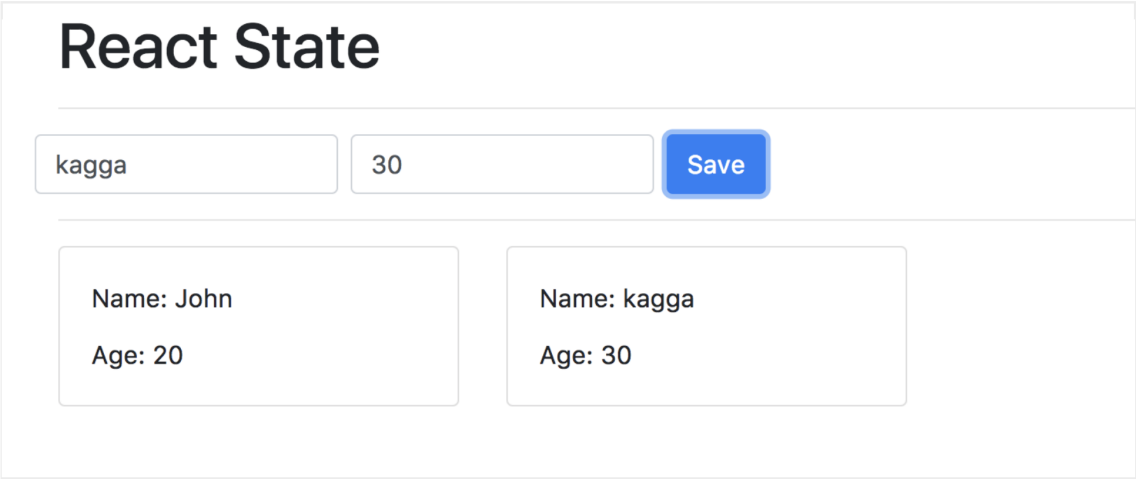
We also need to get the `name` and `age` text entered by the user from the instance variables we set in the `ref` callbacks. After all that we update the component state using `this.setState` as shown in the code snippet below.

```
onSubmit = event => {
 event.preventDefault();
 const name = this.name.value;
 const age = this.age.value;
 const info = {name: name, age: age};
 const data = this.state.data;
 data.push(info);
 this.setState({
   data: data
 });
};
```

Finally, add the `onSubmit` attribute to the form element and `this.onSubmit` as its value referencing the onSubmit function defined within the component.

```
<form className="form-inline" onSubmit={this.onSubmit}>
```

Now open the *index.js* file in the browser and type `kagga` in the name input field and `30` in the age input field then click the **Save** button. A new card will be added on the page as shown in the image below.



State immutability

You can now add the form data into state and display it on the page.

But wait...did you see it? It is okay if you did not. In the introduction of chapter 3, it was pointed out that state in React should never be mutated that to say, should be immutable.

Looking back at the `onSubmit` function, we mutated state when we used the `push` method on the data array from `this.state.data`.

The right way to update state is to create a new data array and then update the state with that new array. This can be achieved in many ways but we are going to use the ES6 spread operator to create a new data array and also add the new info object containing the `name` and `age` from the form as shown in the code snippet below. Make the necessary changes to the earlier code.

```
onSubmit = event => {
 event.preventDefault();
 const name = this.name.value;
 const age = this.age.value;
 const info = {name: name, age: age};
 const data = [...this.state.data, info];
 this.setState({
   data: data
 });
};
```

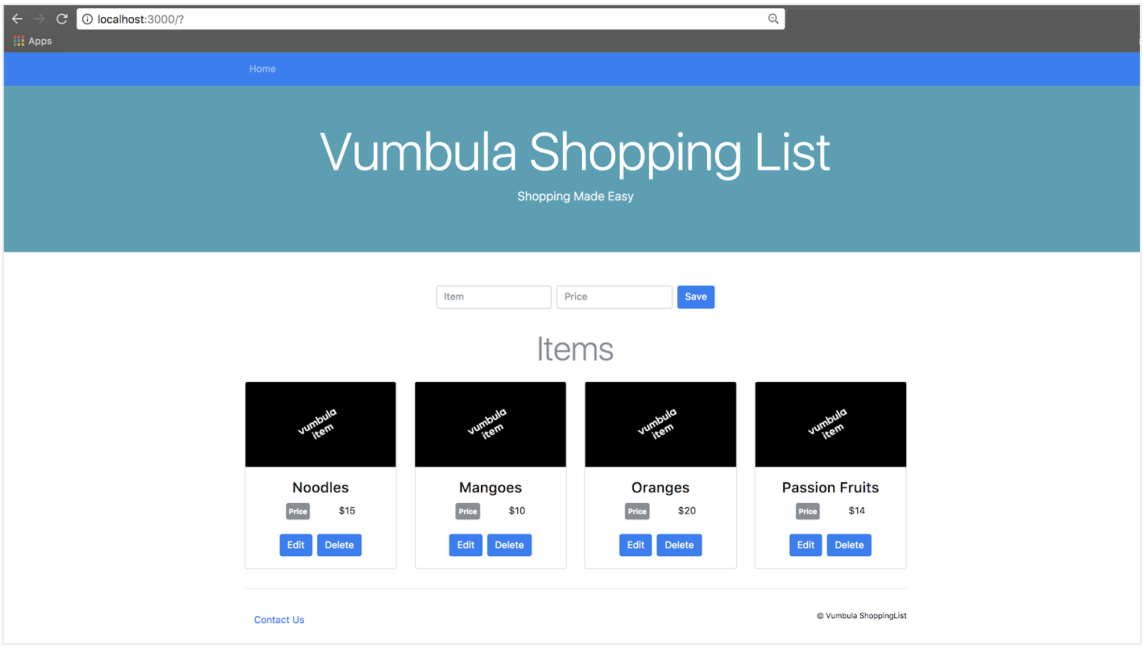
With the above changes, we still get the same results With the added benefit of state immutability. Find all code for this section here.

Project Three: Building a Shopping List App

In this project, we shall combine everything we've learnt until this point as we build our shopping list app. Our app will allow for CRUD functionality.

The starter files are available for download here, in there, you will find TODOs to guide you if you would like to attempt the project on your own. The final code for the project is also available here and has solutions to all the TODOs in the starter code.

After cloning the repository, `cd` into your project directory and install the dependencies by running `npm install` or `yarn install`. Run `npm start` or `yarn start` to view the project in the browser and make sure that everything is working well.



In the src folder, you will find a file `App.js` that contains an *App* component. Inside the *App* component, there are functional components which include *Nav*, *Jumbotron*, *AddItem* and *Footer*.

Adding Items to the Shopping List

* Add `name` and `price` as properties to the state object. These will hold the new item before it is saved to state. * After the `name` and `price` have been saved as a new item in the items array that is within state, they are reset to their defaults.

Destructure the `name` and `price` from the state object and pass them as props to the *AddItem* component.

```
const {name, price} = this.state;
<AddItem
   name={name}
   price={price}
/>
```

- Also within the *AddItem* component, destructure the `name` and `price` within the function argument parentheses.
- Add a value attribute to both the `name` and `price` input elements with the variables destructed from the component argument list as necessary.
- Add an attribute name to both the `name` and `price` input elements with string values of name and price respectively.
- We need to check the type of the props we are passing to the *AddItem* component. To

do this we use the `prop-types` package. Follow the steps below to add type checking. *Import `PropTypes` from the `prop-types` package, note that this package is already installed, it is part of the dependencies in the `package.json` but not bundled with React. It should always be installed separately using `npm`.

Add a `propTypes` object for `name` and `price` with a type of string and mark them as required as shown below.

```
AddItem.propTypes = {
 name: PropTypes.string.isRequired,
 price: PropTypes.string.isRequired,
};
```

At this point, your component should look like this:

```
import React from 'react';
import PropTypes from 'prop-types';

export const AddItem = ({name, price}) => (
 <div className="row justify-content-center">
   <form className="form-inline">
     <input
       type="text"
       className="form-control mb-2 mr-sm-2"
       placeholder="Item"
       value={name}
       name="name"
     />
     <div className="input-group mb-2 mr-sm-2">
       <input
         type="text"
         className="form-control"
         placeholder="Price"
         value={price}
         name="price"
       />
     </div>
     <button type="submit" className="btn btn-primary mb-2 pxy-4">Save</
button>
   </form>
 </div>
);

AddItem.propTypes = {
 name: PropTypes.string.isRequired,
 price: PropTypes.string.isRequired,
};
```

In order to get the `name` and `price` the user types into the input fields, we need to add an `onChange` event listener to both the `name` and `price` inputs.

- Create an arrow function called `handleInputChange` which accepts `event` as its own argument within the *App* component.
- Within the function, use the passed in `event` parameter to get the `target` input element; from the `target` get the `value` and `name` of the input element.

Use the `setState` function to add the `name` and/or `price` to the `name` and `price` properties in the state object.

```
handleInputChange = event => {
 const target = event.target;
 const value = target.value;
 const name = target.name;
 this.setState({
   [name]: value
 });
};
```

Define an `onChange` prop on the *AddItem* component with a value of `this.handleInputChange`

```
<AddItem
   name={name}
   price={price}
   onChange={this.handleInputChange}
/>
```

- In the `AddItem` file and component, add the `onChange` prop to the list of destructured elements in the function argument list.
- Add `onChange` to the propTypes object as a required function.
- Add an `onChange` attribute to both input elements with the value of the `onChange` prop.

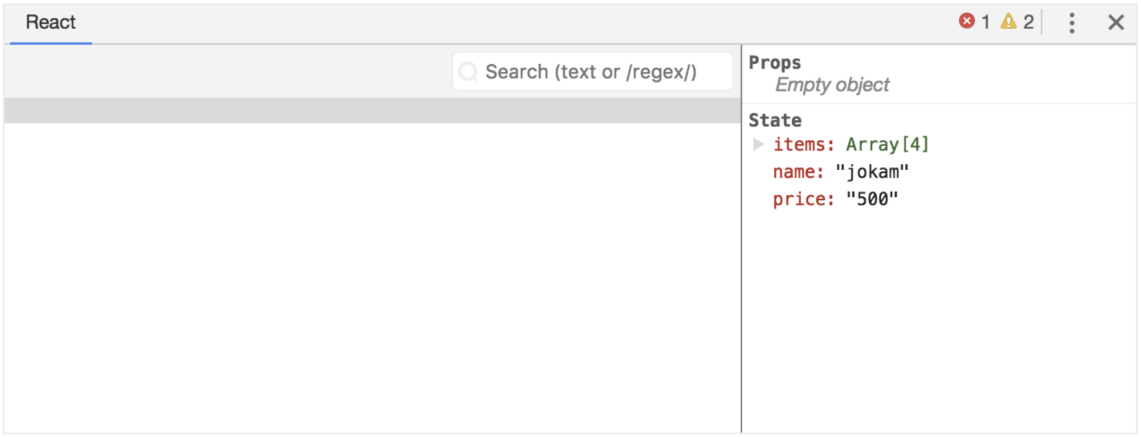
At this point your *AddItem* component should look like this

```
import React from 'react';
import PropTypes from 'prop-types';

export const AddItem = ({name, price, onChange}) => (
 <div className="row justify-content-center">
   <form className="form-inline">
     <input
       type="text"
       className="form-control mb-2 mr-sm-2"
       placeholder="Item"
       value={name}
       name="name"
       onChange={onChange}
     />

     <div className="input-group mb-2 mr-sm-2">
       <input
         type="text"
         className="form-control"
         placeholder="Price"
         value={price}
         name="price"
         onChange={onChange}
       />
     </div>
     <button type="submit" className="btn btn-primary mb-2 pxy-4">Save</
button>
   </form>
 </div>
);
AddItem.propTypes = {
 name: PropTypes.string.isRequired,
 price: PropTypes.string.isRequired,
 onChange: PropTypes.func.isRequired,
};
```

- Head over to the browser, let's check out our progress.

Open the React developer tools and look at the state section. Notice that within the state section, the *name* and *price* properties have empty strings as their values. image::vr-chap4-2.png[]

As you type into the `name` or `price` input fields, the state updates with each keystroke.



- The `name` and `price` now need to be added to the items array in state, so that they are rendered when the **Save** button is clicked. Let's do this now.
- Define an arrow function called `addItem` which accepts `event` as its only argument
- Within it call `preventDefault()` on event, to prevent the default behaviour of the button.
- Use destructing to get the set `name` and `price` from state.
- Since an `id` is needed when saving an item to be used as a key, get the length of the existing items array in state. Then, use the ternary operator to either increment the `id` of the last element in the items array or to use `1` as the `id` if the items array is empty.

Use the `setState` function to add the new item to the items array. Remember not to mutate state. Use the spread operator for the existing items within the array and the `Object.assign` function for adding the new item to the array. Set the `name` and `price` back to their defaults as shown below.

```
addItem = event => {
  event.preventDefault();
  const {name, price} = this.state;
  const itemsInState = this.state.items;
  const itemsArrayLength = itemsInState.length;
  const id = itemsArrayLength
             ?
             (itemsInState[itemsArrayLength - 1].id + 1)
             :
             1;
  this.setState({
   items: [
     ...itemsInState,
     Object.assign({}, {
       id,
       name,
       price
     })
   ],
   name: "",
   price: ""
 })
};
```

- Define an `onSubmit` prop on the *AddItem* component with a value of `this.addItem` within the *App* component.
- Within the *AddItem* component, add an `onSubmit` to the list of destructured elements in the function argument list.
- Add `onSubmit` to the proptypes object as a required function.
- Add an `onSubmit` attribute to the form with the value of `onSubmit`.
- Moment of truth, open the app in your browser. At this point, you should be able to view the added item when you click the save button.

The final working code for this section can be found here.

Editing/Updating the Items on the Shopping List

In this section, we are going to tackle editing and updating the items. The general idea is to click the edit button so that the `name` and `price` fields turn into input fields, thus giving the user the ability to modify their content. After modifying the `name` and `price` the user can then click the `save` button in order for the `name` and `price` to revert to their display mode. The starter code for this section can be found here.

Let's get started

- Define an arrow function with a name of `toggleItemEditing` which accepts `index` as its only argument. The `index` will be used to find the item to be edited.
- Within this function use the `setState` method and within it, define the item's `key`. To set its value, loop through the `items` array and when the item with the passed in `index` is found, add an `isEditing` property with a value of `!item.isEditing`. This will toggle the `isEditing` boolean accordingly. The function implementation is as shown below.

```
toggleItemEditing = index => {
 this.setState({
   items: this.state.items.map((item, itemIndex) => {
     if (itemIndex === index) {
       return {
         ...item,
         isEditing: !item.isEditing
       }
     }
     return item;
   })
 });
};
```

- Add `toggleEdit` as a prop to the *ItemCard* component and define an arrow function that calls the `toggleItemEditing` function passing it the `index` as the argument.
- This function acts as a callback and will only execute when a button is clicked.

```
toggleEditing = {() => this.toggleItemEditing(index)}
```

- Within the *ItemCard* component, add an `onClick` attribute to the edit button with the `toggleEditing` prop as its value.
- Use the `isEditing` property of the item to toggle between showing **Edit** or **Save** as the button text. Do not forget to add `toggleEditing` to the list of propTypes in the *ItemCard* component.

```
<button
   type="button"
   className="btn btn-primary mr-2"
   onClick={toggleEditing}
>
 {item.isEditing ? "Save" : "Edit"}
</button>
```

- At this point, clicking the edit button in the browser will toggle its text between **Save** and **Edit**.
- Also note that the `isEditing` property changes its value whenever the Edit button is clicked as shown below in the React devtools.

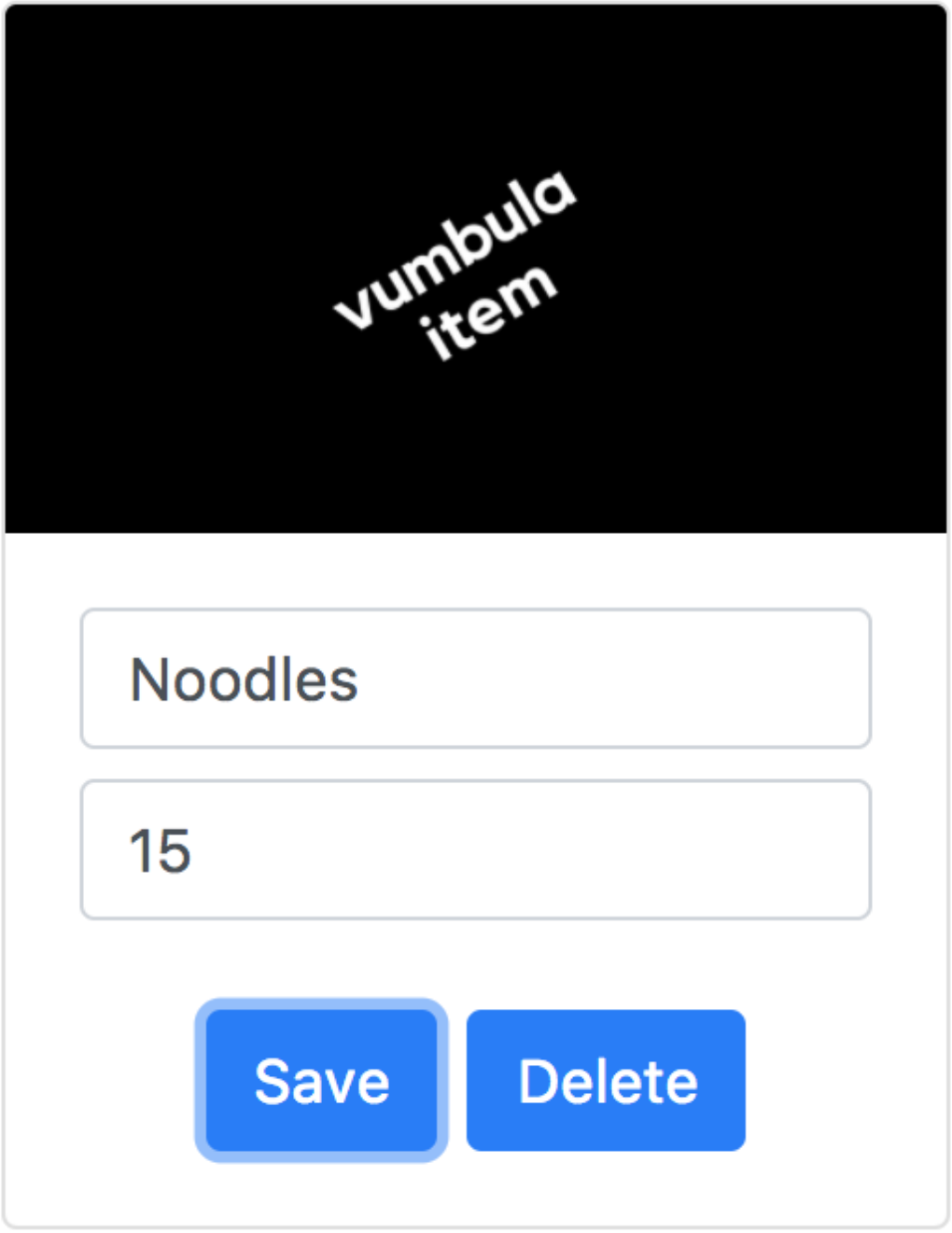
- Now, we use the `item.isEditing` property to either render the input fields or display the `name` and `price` of the item within the card body.

Also, add the `value` attribute to the `name` input element with a value of `item.name` and also the `price` input element with item.price.

```
<div className="card-body">
 {item.isEditing
     ?
     <div className="mb-4">
       <input
           type="text"
           name="name"
           className="form-control mb-2 mr-sm-2"
           placeholder="Item"
           value={item.name}
           required
       />
       <input
```

```
            type="number"
            name="price"
            className="form-control"
            placeholder="Price"
            value={item.price}
            required
          />
        </div>
        :
        <div>
          <h4 className="card-title text-center">
            {item.name}
          </h4>
          <div className="row justify-content-center mb-4">
            <p className="card-text">
              <span className="badge badge-secondary py-2 mr-5">Price</span>
              <span>${item.price}</span>
            </p>
          </div>
        </div>
  }

  <div className="row justify-content-center">
    <div>
      <button
          type="button"
          className="btn btn-primary mr-2"
          onClick={toggleEditing}>
        {item.isEditing ? "Save" : "Edit"}
      </button>
      <button
          type="button"
          className="btn btn-primary">
        Delete
      </button>
    </div>
  </div>
 </div>
```

At this point, when you open the app in the browser and click the `edit` button, the input fields should be visible. Clicking the `save` button should cause them to disappear as shown below.



Attempting to type into the `item-name` *and* `item-price` *fields will not work at this point. This is because we are using* ***controlled inputs*** *but the inputs do not have* `onChange` *event handlers. Fixing this is fairly forward, we need to write a function to handle the editing functionality.*

- Within the *App* component define an arrow function with a name of `handleItemUpdate` which accepts an `event` and `index` as its only arguments.
- This function is similar to one we defined above that was updating the state with the

`name` and `price` of an item before it was saved into the items array. The difference is that we use the `setState` function to find the item with the passed in `index` and update its `name` and/or `price` with the new values. We use the spread operator to populate the already existing item properties.

We return the item after updating it as shown below.

```
handleItemUpdate = (event, index) => {
 const target = event.target;
 const value = target.value;
 const name = target.name;
 this.setState({
   items: this.state.items.map((item, itemIndex) => {
     if (itemIndex === index) {
       return {
         ...item,
         [name]: value
       }
     }
     return item;
   })
 });
};
```

By now, you know the flow. Go ahead and add an `onChange` prop to the *ItemCard* component with the above function as its value.

Add the passed in prop to the *ItemCard* component argument list and use an arrow function which accepts an `event`. This function returns this prop as the value to the `onChange` attribute to both the `name` and `price` input elements, passing it the `event` and `index` as shown below.

```
onChange = {event => onChange(event, index)}
```

*The `onChange` prop name can have any name. Here `onChange` is used for simplicity, but the `onChange` attribute on the input elements **CANNOT** have any other name*

The *ItemCard* component looks like this in the end.

```
export const ItemCard = ({toggleEditing, item, image, onChange, index}) => (
 <div className="col-md-6 col-lg-3">
   <div className="card mb-3">
```

```
      <img className="card-img-top" src={image}/>
      <div className="card-body">
        {item.isEditing
          ?
          <div className="mb-4">
            <input
              type="text"
              name="name"
              className="form-control mb-2 mr-sm-2"
              placeholder="Item"
              value={item.name}
              onChange={event => onChange(event, index)}
              required
            />
            <input
              type="number"
              name="price"
              className="form-control"
              placeholder="Price"
              value={item.price}
              onChange={event => onChange(event, index)}
              required
            />
          </div>
          :
          <div>
            <h4
              className="card-title text-center">
              {item.name}
            </h4>
            <div
              className="row justify-content-center mb-4">
              <p className="card-text">
                <span className="badge badge-secondary py-2 mr-5">Price</span>
                <span>${item.price}</span>
              </p>
            </div>
          </div>
        }

        <div className="row justify-content-center">
          <div>
            <button
              type="button"
              className="btn btn-primary mr-2"
              onClick={toggleEditing}>
              {item.isEditing ? "Save" : "Edit"}
```

```
            </button>
            <button
              type="button"
              className="btn btn-primary">
              Delete
            </button>
          </div>
        </div>
      </div>
    </div>
  </div>
);


ItemCard.propTypes = {
  image: PropTypes.string.isRequired,
  item: PropTypes.shape({
    name: PropTypes.string.isRequired,
    price: PropTypes.string.isRequired
  }),
  toggleEditing: PropTypes.func.isRequired,
  onChange: PropTypes.func.isRequired,
};
```

The app should now permit update of the `name` and/or `price` of any item successfully. Find the final code for this section here.

Deleting an Item from the Shopping List

Deleting an item should be straightforward, the idea is that when a user clicks the **Delete** button, an item is removed from the items array in state. Here is the starter code for this section.

Let's get started adding the delete functionality.

- Define an arrow function, `onDelete` that takes `index` as its only argument.
- Within the function, call the `setState` function and define an object with items as a property key and the value being an empty array.
- Within the array, use the spread operator to populate the array with items from the zeroth index to the item before the passed in index using the `slice` method.

At this point, only part of the array is being included in the new array using the spread operator. To add the remaining part of the array without the item with the passed in index (item to be deleted), the spread operator and the slice method are used again to get the items at the `index` passed in `+ 1` as shown below.

```
onDelete = index => {
 this.setState({
   items: [
     ...this.state.items.slice(0, index),
     ...this.state.items.slice(index + 1)
   ]
 });
};
```

- Moving on, define an `onDelete` prop on the *ItemCard* component with its value being an arrow function that calls the `onDelete` function in the *App* component, passing it the `index` of the item to be deleted.
- Within the *ItemCard* component destructure the `onDelete` prop in the components argument list.
- Go on and add `onDelete` to the components `propTypes`.

Finally, add an `onClick` attribute to the **delete** button with the `onDelete` prop as its value as shown below.

```
<button
   type="button"
   className="btn btn-primary"
   onClick={onDelete}>
 Delete
</button>
```

Save all your changes and open the app in a browser, when you click on the delete button that item card should be deleted and thus, disappear. Find the final code for this section here.

Chapter Five
Handling Routing in React

Routing is the ability to move between different parts of an application when a user enters a URL or clicks an element (link, button, icon, image etc) within the application.

Up until this point, you have dealt with simple projects that do not require transitioning from one view to another, thus, you are yet to interact with Routing in React.

In this chapter, you will get introduced to routing in a React application. To extend your applications by adding routing capabilities, you will use the popular React-Router library. It's worth noting that this library has three variants:

⇒ **react-router**: the core library

⇒ **react-router-dom**: a variant of the core library meant to be used for web applications

⇒ **react-router-native**: a variant of the core library used with react native in the development of Android and iOS applications.

Often, there is no need to install the core react-router library by itself, but rather a choice is made between react-router-dom and react-router-native, depending on the situation. Both react-router-dom and react-router-native import all the functionality of the core react-router library.

The scope of this book is in the realm of web applications so we can safely choose react-router-dom. This library is installed in a project by running the command below in the project directory

```
npm install --save react-router-dom
```

Routers

The react-router package includes a number of routers that we can take advantage of depending on the platform we are targeting. These include `BrowserRouter`, `HashRouter`, and `MemoryRouter`.

For the browser-based applications we are building, the `BrowserRouter` and `HashRouter` are a good fit.

The `BrowserRouter` is used for applications which have a dynamic server that knows how to handle any type of URL whereas the `HashRouter` is used for static websites with a server that only responds to requests for files that it knows about.

Going forward, we shall use the `BrowserRouter` with the assumption that the server running our application is dynamic. Worth noting is that any router expects to receive only one child. Take the example below

```
ReactDOM.render(
  <BrowserRouter>
    <App/>
  </BrowserRouter>,
  document.getElementById('root'));
```

In this example, the `<App/>` component is the child to the `<BrowserRouter>` and should be the only child. Now, the routing can happen anywhere within the `<App/>` component, however, it is considered good practice to group and place all the routes in the same place. More on this later.

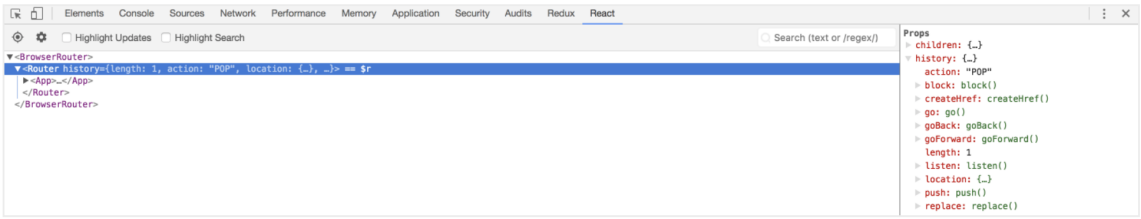
History

Each router creates a history object that it uses to keep track of the current location and re-renders the application whenever this location changes. For this reason, the other React Router components rely on this `history` object being present; which is why they need to be rendered inside a router.

The `BrowserRouter` uses the HTML5 history API to keep the user interface in sync with the URL in the browser address bar.

The `history` object created by the Router contains a number of properties and one of the location property whose value is also an object. The location property is one we shall put a lot of emphasis on in this chapter as the rest are beyond the scope of this book.

When the earlier example is rendered in the browser, you should be able to see the created `history` object within the React DevTools window as shown below.



The `location` object within the history object is shaped like so

```
{ pathname, search, hash, state }
```

The `location` object properties are derived from the application URL.

Routes

The `<Route/>` component is one of the most important building blocks in the React Router package. It renders the appropriate user interface when the current location matches the route's `path`. The `path` is a prop on the `<Route/>` component that describes the pathname that the route should match as shown in the example that follows

```
<Route path="/items" />
```

This route is matched when the pathname is `/items` or, all other paths that start with `/items/` for example `/items/2`. If the intention is to strictly match only `/items`, the `<Route/>` component accepts an exact prop. Adding this ensures that only the pathname that exactly matches the current location is rendered. Below is an example that uses the **exact** prop.

```
<Route exact path="/items" />
```

When a `path` is matched, a React component should be rendered so that there's a change in the UI.

It is also worth noting that the **Path-to-RegExp** package is used by the react-router package to turn a path string into a regular expression and matched against the current *location*.

The `<Route/>` component provides three props that can be used to determine which component to render:

⇒ component

⇒ render

⇒ children

Component Prop

The `component` prop defines the React element that will be returned by the Route when the `path` is matched. This React element is created from the provided component using **React.createElement**. Below is an example using the **component** prop.

```
<Route
  exact
  path="/items"
  component={Items}
/>
```

In this example, the *Items* component will be returned when the `path` matches the current *location*.

Render Prop

The `render` prop provides the ability for inline rendering and passing extra props to the element. This prop expects a function that returns a React element when the current *location* matches the route's `path`. Below are examples demonstrating the use of the `render` prop on a *Route* component.

```
<Route
  exact
  path="/items"
  render={() => (<div>List of Items</div>)}
/>
```

In the example above, when the current location matches the `path` exactly, a React element is created and the string **List of Items** is rendered in the browser.

```
const cat = {category: "food"}
<Route
  exact path="/items"
  render={props => <Items {...props} data={cat}/>}
/>
```

In the second example, data represents the extra props that are passed to the Items component. Here, `cat` is passed in as the extra prop.

Children Prop

The `children` prop is similar to the `render` prop since it always expects a function that returns a React element. The major difference is that the element defined by the `child` prop is returned for all paths irrespective of whether the current location matches the path or not.

```
<Route children={props => <Items {···props}/>}/>
```

In this case, `Items` component is **always** rendered.

Switch

The react-router library also contains a `<Switch/>` component that is used to wrap multiple `<Route/>` components. The `Switch` component **only** picks the **first** matching route among all its children routes.

The next example demonstrates how multiple routes behave in the absence of the `Switch` component.

```
<Route
 path="/items"
 render={() => (<div><em>List of items</em></div>)}
/>
<Route
 path="/items/2"
 render={() => (<div>Item with id of 2</div>)}
/>
```

In the browser, when you navigate to `/items/2`, the React elements in both `Route` components will be rendered as shown below

```
List of items
Item with id of 2
```

This could be the intended behaviour, where the first component displays the title and the other routes with the same base path render different UIs.

Let's modify the example above and include the `<Switch/>` component and observe the behaviour when we navigate to `/items/2`.

```
<Switch>
  <Route
    path="/items"
    render={() => (<div><em>List of items</em></div>)}
  />
  <Route
    path="/items/2"
    render={() => (<div>Item with id of 2</div>)}
  />
</Switch>
```

In the browser, only `List of Items` will be rendered. This is because the `Switch` component matches only the first path that matches the current *location*. In this example, the route **/items** was matched when **/items/2** was entered in the browser's address bar.

Link

The react-router package also contains a `<Link/>` component that is used to navigate the different parts of an application by way of hyperlinks. It is similar to HTML's anchor element but the main difference is that using the *Link* component does not reload the page but rather, changes the UI. Using an anchor tag would require that the page is reloaded in order to load the new UI. When the *Link* component is clicked, it also updates the URL.

Let's explore the use of the *Link* component further by creating an app that allows us to navigate between **categories** and **items**.

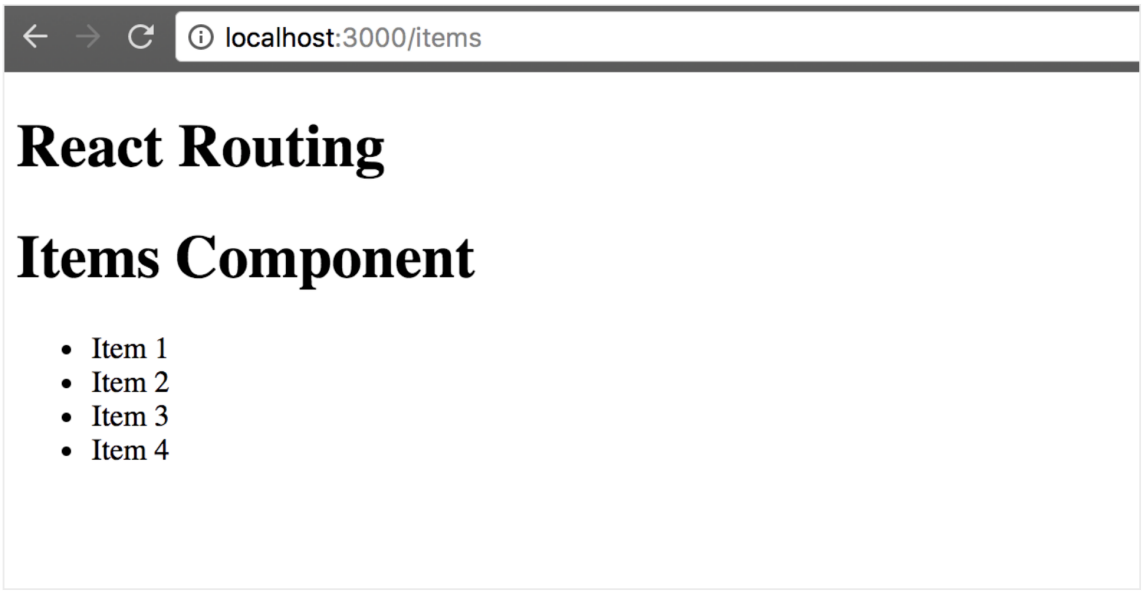
```
export const Home = () => (
  <div>
    Home Component
    <ul>
      <li>
        <Link to="/items">Items</Link>
      </li>
      <li>
        <Link to="/category">Category</Link>
       </li>
    </ul>
  </div>
);
```

The *Home* component contains links to *Items* and *Categories* components.

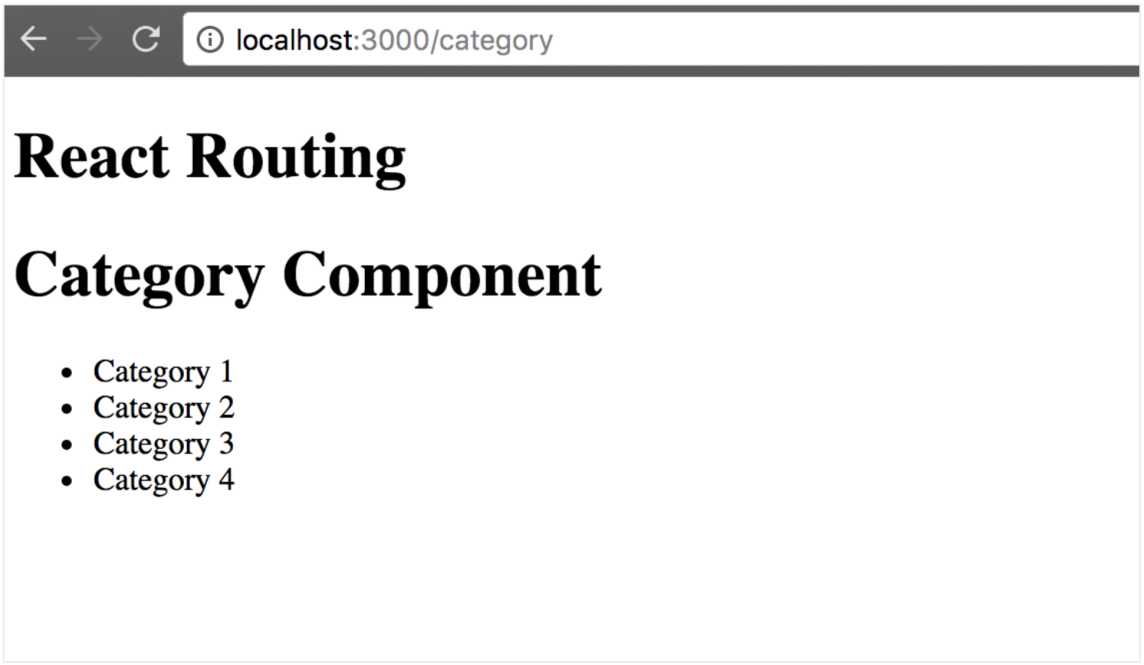
The `<Link/>` component uses `to` as a prop to define the *location* to navigate to. This prop can either be a string or a *location* object. If it is a string, it is converted to a *location* object. Note that the pathname **must** be absolute.

To get the example set up on your machine, clone the project here and run `npm install && npm start`. The rendered page should look like this

Clicking on the **Items** link triggers a UI change and updates the URL in the address bar as well.



Similarly, clicking on the **Category** link trigger a UI change and updates the URL in the address bar.



Nested Routing

You now have an understanding of how the `<Route/>` component and path work. We can now move on to nested routing in a React application.

When the router's `path` and *location* are successfully matched, a `match` object is created. This object contains information about the URL and the path. This information can be accessed as properties on the match object.

Let's take a closer look at the properties:

⇒ `url` : A string that returns the matched part of the URL

⇒ `path` : A string that returns the route's path

⇒ `isExact` : A boolean that returns true if the match was exact

⇒ `params` : An object containing key-value pairs that were matched by the **Path-To-RegExp** package.

You can try this out using Route tester to match routes to URLs.

In order to successfully achieve nested routing, we shall use `match.url` for nested Links and `match.path` for nested Routes.

Let's explore the use of nested routing by working on an example. Clone the project here and run `npm install && npm start` to get it set up and fired up.

This example contains four components;

⇒ *Header* component which contains the Home, Items and Category links

⇒ *Home* component which contains dummy data

⇒ *Items* component which contains a list of dummy items

⇒ *Category* component which demonstrates nested routing and dynamic routing

We shall focus on the *Category* component since it contains the nested and dynamic routing.

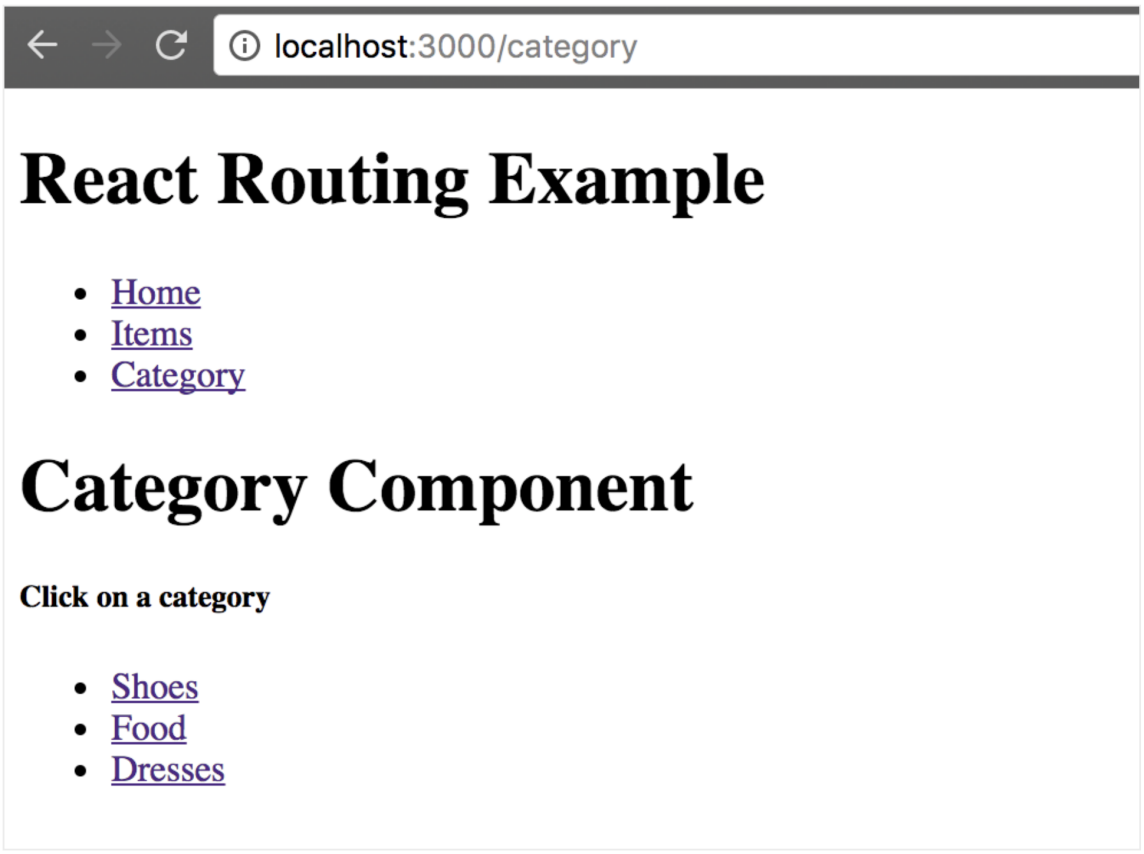
```
export const Category = ({match}) => (
  <div>
  <h1>Category Component</h1>
  <h5>Click on a category</h5>
  <ul>
    <li>
      <Link to={`${match.url}/shoes`}>Shoes</Link>
    </li>
    <li>
      <Link to={`${match.url}/food`}>Food</Link>
    </li>
    <li>
      <Link to={`${match.url}/dresses`}>Dresses</Link>
    </li>
  </ul>
);
```

Based on the code snippet above, when the `Category` link is clicked, a route `path` is matched and a `match` object is created and sent as a prop to the *Category* component.

Within the *Category* component, the match object is destructured in the argument list and links to the three categories are created using `match.url`.

Template literals are used to construct the value of the prop on the *Link* component to the different `/shoes`, `/food` and `/dresses` URLs.

Opening the example in the browser and clicking on the category link reveals three different categories. When any one of these categories is clicked, the URL updates, however, there is no change in the UI.



In order to fix this bug and ensure that the UI changes when a category link is clicked, we create a dynamic route within the *Category* component that uses `match.path` for its `path` prop and then dynamically change the UI.

```
<Route
  path={`${match.path}/:categoryName`}
  render={props =>
                (<div>
                  {props.match.params.categoryName} category
                 </div>
                )
         }
/>
```

Looking closely at the value of the `path` prop in the code snippet above, you can see that we use `:categoryName`, a variable within the **pathname**.

`:categoryName` is the path parameter within the URL and it catches everything that comes after `/category`.

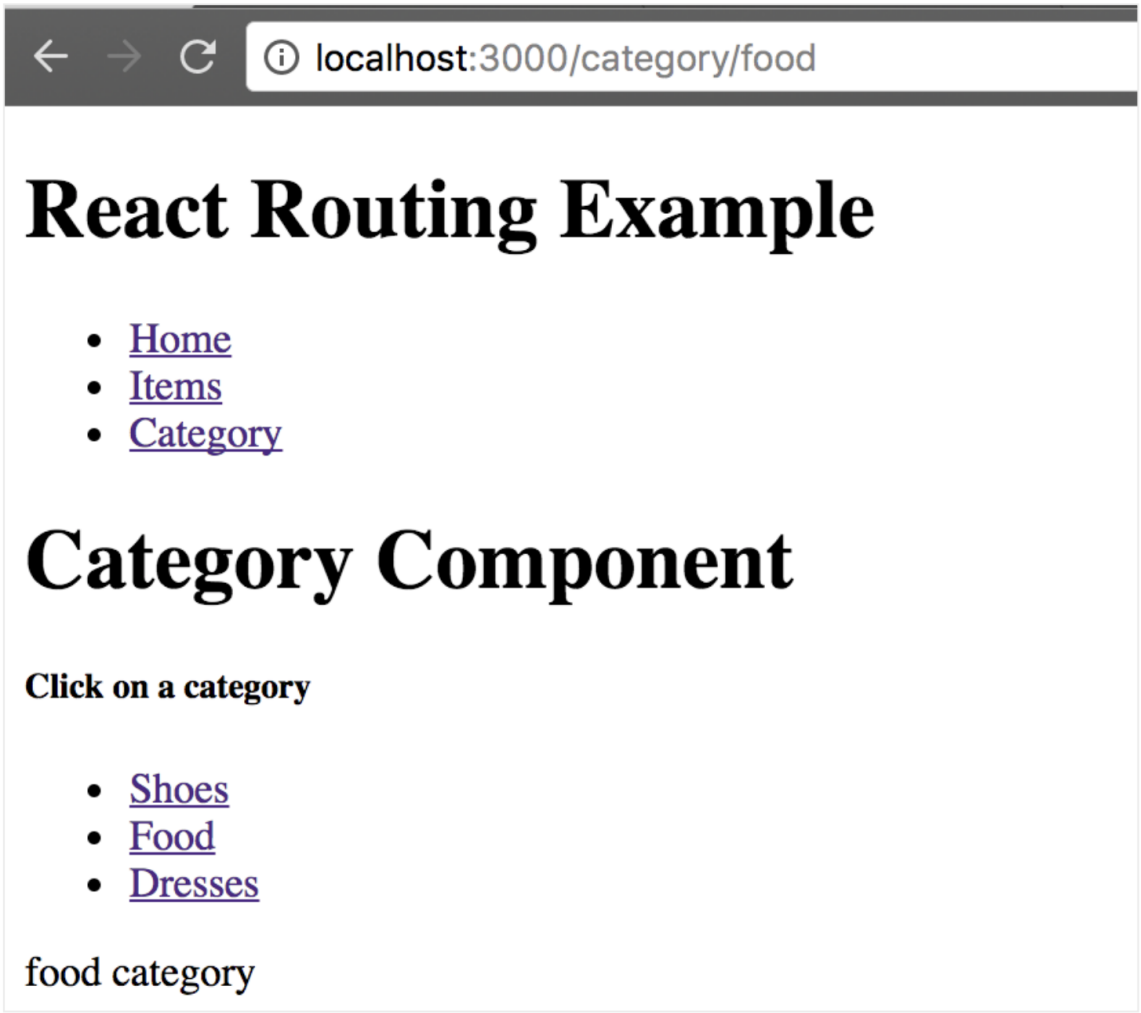
Passing the value to the `path` prop in this way saves us from having to hardcode all the different category routes. Also, notice the use of template literals to construct the right path.

A pathname like `category/shoes` creates a param object like the one below

```
{
 categoryName: “shoes”
}
```

The `render` prop in this route example runs an inline render which displays the `categoryName` param from the `match` object contained within the props.

That should fix the issue of an unchanging UI and now, clicking on one of the categories should trigger an update of both the URL and the UI like so



Protected Routes

The rationale of having is a protected route is that when a user tries to access part of the application without logging in, they are redirected to the login page to sign into the application.

For this redirect to work as intended, the **react-router** package provides a `<Redirect/>` component to serve this purpose. This component has a to prop which is passed to it in form of an object containing the `pathname` and state as shown below.

```
<Redirect
  to={{pathname: '/login', state: {from:props.location}}}
/>
```

Here, the *Redirect* component replaces the current *location* in the stack with the `pathname` provided in the object (`/login`) and then stores the *location* that the user was attempting to visit, in the `state` property. The value in `state` can be accessed from within the *Login* component using `this.props.location.state`.

For example, if a user attempts to navigate to `/admin`, a protected route, without logging in first, they will be redirected to the login page. Following a successful sign in, they will be redirected to `/admin`, the route they intended to visit in the first place.

Custom Routes

In order to achieve the concept of protected routes, we need to understand first how to create custom routes.

Custom routes are a fancy way of saying nesting a route inside a component. This is typically done when there is a need to decide whether a component should be rendered, or not.

In the case of a protected route, a given route should only be accessed when a user is logged in, otherwise, the user should be directed to the login page.

Let's explore custom routes more in the next example. Clone the project here and run `npm install && npm start` to set up.

A private route is also grouped with all other routes as shown below.

```
<Switch><Route exact path="/" component={Home}/>
 <Route path="/items" component={Items}/>
 <Route path="/category" component={Category}/>
 <Route path="/login" component={Login}/>}/>
 <PrivateRoute
   path="/admin"
   component={Admin}
   isAuthenticated={fakeAuth.isAuthenticated}
  />
</Switch>
```

The private route has the `path`, `component` and `isAuthenticated` props. Let's take a closer look at the private (custom) route.

```
import React from 'react';
import { Route, Redirect } from "react-router-dom";

const PrivateRoute = ({component: Component, isAuthenticated, ...rest}) => (
  <Route {...rest} render={props => (
    isAuthenticated
      ?
      (<Component {...props}/>)
      :
      (<Redirect to={{pathname: '/login', state: {from: props.location}}}/>)
  )}/>
);
```

We destructure the props within the argument list and rename `component` to `Component`. We use the *Route* component by passing it the `···rest` and `render` props. Within the `render` prop, we write logic that determines whether to render a component and which one to render if the user is signed in. Otherwise, the user is redirected to the login page.

The *Login* component contains a dummy authentication method which signs the user in when they click the **Login** button within its `render` method. See below the code snippet from the *Login* component.

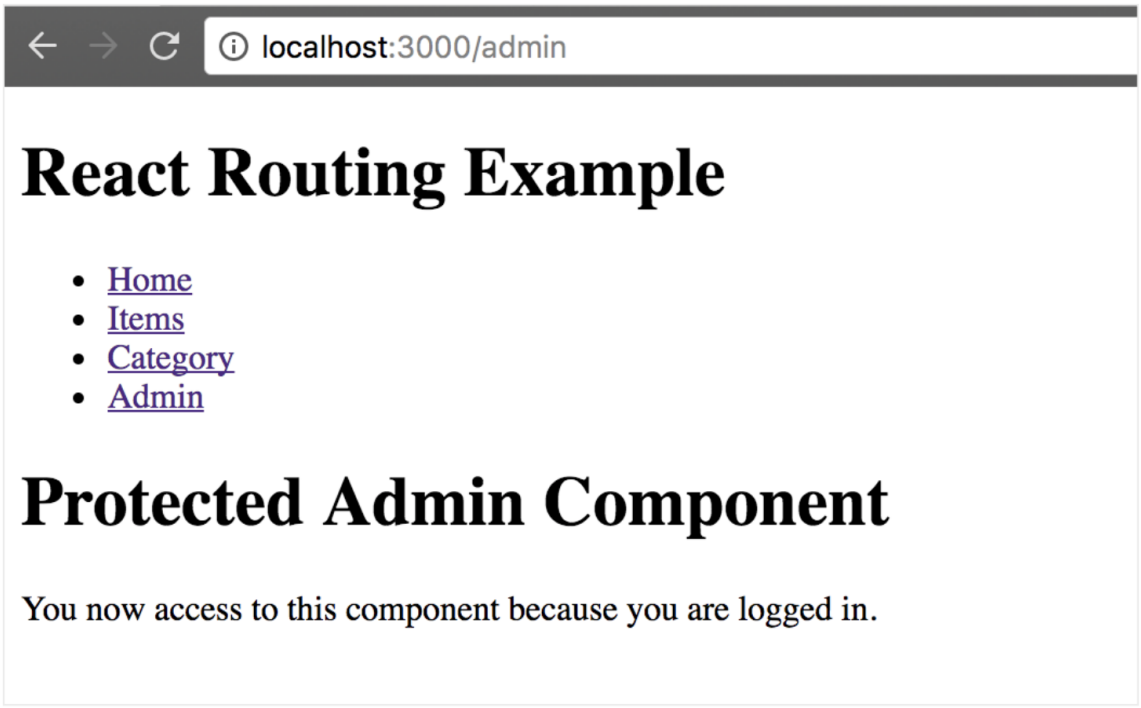
```
import React from 'react';
import {Redirect} from 'react-router-dom';

class Login extends React.Component {
  state = {
    redirectToReferrer: false
  };

  login = () => {
    fakeAuth.authenticate(() => {
      this.setState({
        redirectToReferrer: true
      })
    })
  };

  render() {
    const { from } = this.props.location.state || {from: {pathname: '/'}};
    const { redirectToReferrer } = this.state;

    if (redirectToReferrer) {
      return (
        <Redirect to={from}/>
      )
    }

    return (
      <div>
        <p> You must log in to view the content at {from.pathname} </p>
        <button onClick={this.login}> Log in </button>
      </div>
    )
  }
}

/* A fake authentication function */
export const fakeAuth = {
  isAuthenticated: false,
  authenticate(cb) {
    this.isAuthenticated = true;
    setTimeout(cb, 100)
  },
};

export default Login
```

The `redirectToReferrer` state property is set to `true` when the user is signed in. This triggers a redirect to the route they had intended to visit, or to the '/' path in case they navigated directly to the login route.

Run `npm start` if you do not already have the project running and navigate to `localhost:3000`. You should see this

Clicking on the **Admin** link when not signed in redirects you to the `/login` page, showing the **Login** button.

After clicking the **Login** button, you are redirected to the protected admin page as shown below



You’re now fully equipped to build a complete React application. You are well on your way to gaining the ability to write complex React applications. We’re excited to see what you’ll build.

Index

R

S

T

U

V

About the Authors

Edmond Atto

Edmond Atto is a Senior Software Engineer at Andela and Co-founder of Arvana - a startup that launched a mobile application providing a digitised physical addressing system for Uganda, and The Andela Way - a publication that features freemium content for people looking to get started with software development. He has written standalone articles for several publications and/or websites i.e. The Creative Cafe, The Andela Way and Digest Africa. He is part of the 2019-20 cohort of the Obama Foundation Scholars at Columbia University in New York. Away from work, he spends time dreaming up new travel destinations, reading books and watching car shows.

John Kagga

John Kagga is a Senior Software Engineer at Andela. For two years before joining Andela, he was the Co-founder and lead developer at Arvana, a startup that launched a mobile application providing a digitised physical addressing system for Uganda. He has written several articles for The Andela Way; a publication with over 5000 unique daily visitors. He is passionate about growing technology communities; he pioneered a Women in Tech Leadership program powered by Andela. Away from work, he spends time playing football, tennis, and dancing. He is currently working on a web platform for Educators, at Intentional Futures - a Seattle based company.

Colophon

Credit goes to John Paul Serumba who designed the cover page.

The word *Vumbula* used in the title of this book is from a local dialect in Uganda, to be precise *Luganda*. Vumbula literally translates to *discover*. So this book means **Discover React**.

The text font used is Noto Serif and the code font is M+ 1mn.