WA2583 React JavaScript Programming

Web Age Solutions Inc.
USA: 1-877-517-6540
Canada: 1-877-812-8887
Web: http://www.webagesolutions.com

For customizations of this book or other sales inquiries, please contact us at:

USA: 1-877-517-6540, email: getinfousa@webagesolutions.com
Canada: 1-877-812-8887 toll free, email: getinfo@webagesolutions.com

Web Age Solutions Inc.
821A Bloor Street West
Toronto
Ontario, M6G 1M1

Table of Contents

Chapter 1 - React Overview

Objectives

In this module we will discuss:

- What is React?
- One way data flow
- JSX
- Virtual (Mock) DOM
- create-react-app

1.1 What is React?

- React (also referred to as React.js or ReactJS) is an open-source JavaScript library for creating data-driven interactive views for web pages and single-page applications
  - Main web site: https:///reactjs.org
  - Git repository: https://github.com/facebook/react
- Originally developed by Jordan Walke at Facebook and first released in March 2013; now maintained by Facebook, Instagram and a vibrant React development community
- Who uses it: Facebook, Instagram, Et al.

1.2 What's in a Name?

- The *React* name was chosen to indicate that its functionality is rooted in the reactive programming paradigm
- In reactive programming paradigm, changes in the underlying model (data) are automatically (by means of a library, runtime, etc.) reflected in the view
- Spreadsheets are a good example of applications written using reactive programming paradigm: a change in a cell is automatically propagated to all the cells that depend on it

1.3 React Component Model

- React builds web UIs from components created with a mix of JavaScript and HTML where HTML elements are meshed with JavaScript using JSX technology (more on it later ...)
- You can create components using regular JavaScript functions or ES6 (ECMAScript 2015) classes
- React components are composable which makes React UIs extendable
- React components can maintain their state
- React can run and render HTML on Node.js (the server-side rendering option)
  - Additional libraries are required

1.4 What React Is Not

- React is not another client-side JavaScript MVC framework
  - Some believe that React is the **V** in the MVC triad
    - For that reason, React cannot be compared with complete MV(*) frameworks, like Angular, Ember, and others
- React is not a template-based UI kit
  - **Note**: When used with JSX extensions (more on this later), one can see that React does use templating code (which is positioned more like a domain-specific language)

1.5 What You Will Not Find in React

- Distinct model (data layer)
- Two-way data binding (like in AngularJS)
  - React supports only a unidirectional data flow
- Built-in support for asynchronous communication (AJAX, Promises)
- If you need any of the above, you will need to borrow this functionality from 3rd party libraries, frameworks, and tool-kits

- Support for IE below version 8 and below is not available

1.6 Motivation for Creating React

- React designers felt that the traditional ways of updating DOM from JavaScript, or from more advanced libraries like AngularJS have a number of shortcomings:
  - Traditional JavaScript way of updating DOM is intrusive (and not elegant): behavior and presentation are mixed
  - Arguably, "unobtrusive" JavaScript models (as supported, for example, by jQuery) may become hard to maintain
  - Angular takes two passes, compile and link, to update the DOM
  - Basically, the consensus is that DOM operations are slow and something more efficient is needed

Notes:

Note: With the "unobtrusive" philosophy in JavaScript, you separate the view and the code concerns so that views (HTML & CSS) are not intermingled with functionality/behavior (Javascript). When Javascript is disabled in the browser, the page should have content to advise the clients of the need to enable JavaScript in their browsers.

1.7 A React JavaScript Example

- The following regular React JavaScript code will create a simple React Element carrying the **Hello, React!** message within an **h1**-heading tag and will attach it to the page element with the *where_to_render* ID (which is used by React as a target container):

```
<script>
  const myElem = React.createElement('h1', null, 'Hello, React!');
  ReactDOM.render(myElem, document.getElementById('where_to_render'));
</script>
```

- **Note**: *null* in the above code indicates that we are not passing any system properties (IE. CSS styles, event handling function references, etc.) to the *myElem* React element

1.8 One-Way Data Flow

- React components receive parameters as immutable properties of JSON-encoded objects
- Components cannot directly modify received properties
- Updates, if needed, are done through a callback function passed when a component is rendered with the *ReactDOM.render()* API method, which takes the following parameter list:

```
ReactDOM.render(
  component/element, DOM container,[function callback]
);
```

- This idea is captured in React's data processing model:
  - *Properties flow down; actions flow up*

1.9 JSX

- JSX is a JavaScript extension specification that allows insertion of HTML fragments into JavaScript
  - *https://facebook.github.io/jsx/*
- You need to reserve a JSX compilation step in your software delivery pipeline (CI workflows, etc.)
  - The compilation (JSX pre-processing) step parses and converts HTML fragments into React JavaScript code that can run in regular browsers
- React components can be created with and without JSX
- The React team recommends using JSX

1.10 A JSX Example

- This JSX fragment (defined in the **<script>** tag):

```
const reactElement =
    <h1>
      Hello, JSX!
    </h1>
```

- will be compiled (transpiled) into this JavaScript code:

```
"use strict";

const reactElement = React.createElement(
  "h1",
  null,
  "Hello, JSX!"
);
```

- The following code can then be used to render element:

```
ReactDOM.render(reactElement, document.getElementById('where_to_render'));
```

1.11 The Virtual (Mock) DOM

- One of the appealing value proposition of React is faster DOM API operations
- React achieves this by caching the web page's native DOM in an in-memory structure referred to as **Virtual (**or **Mock) DOM**
- Any differences between the native DOM object tree (used by the web browser to render the mark-up of the web page) and the virtual (mock) DOM structure are efficiently reconciled by React making the native DOM up-to-date when it is required
- In addition to performance gains, the virtual DOM structure helps with cross-browser compatibility concerns

1.12 Only Sub-components that Actually Change are Re-Rendered

- React programmers write their application code in a way that appears as if the entire page is rendered on each change while, in fact, only sub-components that actually change are re-rendered
- The React Dev team claims that by using some powerful heuristics they managed to turn a complex ($O(n^3)$) performance problem of direct DOM object tree manipulation into a more basic (O(n)) one.

For more details see: *https://facebook.github.io/react/docs/reconciliation.html*

And: *https://en.wikipedia.org/wiki/Big_O_notation*

1.13 create-react-app

- React application projects are made using create-react-app

```
npx create-react-app my-app
```

- The resulting project includes the following dependencies:

```
"react": "^17.0.2",
"react-dom": "^17.0.2",
"react-scripts": "4.0.3",
"web-vitals": "^1.0.1"
"@testing-library/jest-dom": "^5.11.4",
"@testing-library/react": "^11.1.0",
"@testing-library/user-event": "^12.1.10"
```

- The development server is started with an npm script:

```
npm start
```

1.14 Summary

- In this module, we examined React, a component-based web UI library. We looked at:
  - What is React?
  - One way data flow
  - JSX
  - Virtual (Mock) DOM
  - React libraries

Chapter 2 - Basic Components and JSX

Objectives

In this module we will discuss:

- What is JSX
- JSX Transpilation
- React Components and Elements
- Nesting of Elements
- JSX Code Comments
- Setting Styles and Classes
- Input Parameters
- Expressions
- Fragments

2.1 What is JSX?

- JSX is a JavaScript extension specification that allows insertion of XML (HTML) tags/fragments into JavaScript
  - This is an example of simple but valid JSX:

```
const header = <h3> This is JSX all right </h3>;
```

- You can view JSX as a DSL (Domain Specific Language), or as a templating mechanism
- JSX pre-processing (via a source-to-source compilation, a.k.a. transpilation) is required to parse and convert JSX fragments into React JavaScript code that can run in regular browsers
- The React team recommends using JSX over plain React JavaScript code
- More information on JSX can be found here: *https://reactjs.org/docs/introducing-jsx.html*

2.2 JSX Transpilation to React Code Example

- JSX gets transpiled (by any transpiler with this capability) into the functionally equivalent *React.createElement()* React JavaScript function
- For example, this JSX-enabled code:

```
function Hello() {
  return <div>Hello, JSX!</div>;
}
```

- will be transpiled into this React JavaScript code:

```
"use strict";
function Hello() {
        return React.createElement("div", null,"Hello,
JSX!" );
}
```

2.3 Running the Transpiled Code

- To be able to run the above code in your HTML page, you will need to do two things:
  - Provide the target HTML element (container) where you want the React element to be rendered:

    ```
    <div id='where_to_render' >
    ```

  - Have in place this React element rendering code:

    ```
    ReactDOM.render(<Hello />,
    document.getElementById('where_to_render'));
    ```

- When executed, the code will display "*Hello, JSX!*" message

Notes:

The React team recommends wrapping JSX in parentheses and split it over multiple lines for readability, e.g.:

```
const greeting = (
<div>
        Hello, JSX!
</div>
```

```
);
```

2.4 Babel

- To help bring JSX functionality into React apps, the React team recommends using Babel (*https://babeljs.io/*) to transpile JSX into standard JavaScript
- Babel has presets (collections of plug-ins / transformers) for converting JS6 (ECMAScript 2015) and JSX into JavaScript code runnable in commonly used browsers
- Babel gives you three main ways of usage:
  - As JavaScript library that you can import and use in your pages
    - See next slide for details ...
  - On-line REPL (*https://babeljs.io/repl*)
    - Usually used for small projects and quick prototyping purposes where you are OK with manual a copy & paste type of work
  - Babel CLI npm module that you can install locally and run on-premise on your Node.js server
- You can install and configure Babel in any of the supported ways listed here: (*https://babeljs.io/docs/setup/*)

2.5 Playing Around in CodePen

- CodePen (*http://codepen.io/*) is a web-based playground for various font-end web technologies, including React
- It contains a number of collections grouped by various libraries
- E.g. React's code playground :
  1. *http://codepen.io/k3no/post/getting-started-with-reactjs*

2.6 React Components

- React components are self-contained, re-usable blocks of code that are designed to create composable UI

- Components are based on JavaScript functions that accept input parameters (called *props*) and return an instance of React component

```
const MyComp = function(props) { return
<h3>{props.title}</h3> }
```

- A React component controls what appears in a specific area of the browser's screen.

2.7 Creating a Functional Component Example

- Simply create a JavaScript function that returns a valid JSX element:

```
function Hello() {
  return <div>Hello, JSX!</div>;
}
```

- Make sure the function name starts with a capital letter otherwise React will not recognize it as a component.
- To render the above component, you need to bootstrap it with this code:

```
ReactDOM.render(<Hello />, document.getElementById('where_to_render'));
```

- Note the way *ReactDOM.render()* references the *Hello* function using JSX tag syntax:

```
<Hello />
```

2.8 Component Names Must Be Capitalized

- If you refer to a component from a JSX tag, the first letter of the component name must be capitalized
  - *<MyComponent />*
- Lowercase names are treated as HTML tags, like or <h3>
- Always start component names with a capital letter

2.9 Components vs Elements

- React components are built from single elements
  - This is an element:

```
const myElement = <h1>The message</h1>;
```

- An element describes what you want to show on the screen
- The *ReactDOM.render()* function can render components as well as single elements

2.10 Elements Are Immutable

- React elements are immutable
- Once an element is rendered, you can not change its children or attributes
- The only way to update the target UI area is to re-render the element (or create a new one) with a *ReactDOM.render()* call

2.11 Nesting JSX Elements

- You can add JSX elements as the children to the parent JSX tag. This is useful for displaying nested components:

```
<Container>
  <ComponentA />
  <ComponentB />
</Container>
```

2.12 Example of JSX Nesting

- You can build a hierarchy of React components like so:

```
function SubComp1(props) {
  return <h4>{props.p1}</h4>;
}

function SubComp2(props) {
  return <h4>{props.p2}</h4>;
}

function MainCompositeComponent() {
```

```
  return (
    <div>
      <SubComp1 p1="First instance of SubComp1" />
      <SubComp1 p1="Second instance of SubComp1"  />
      <SubComp2 p2="First instance of SubComp2"  />
    </div>
  );
}

ReactDOM.render(
  <MainCompositeComponent />,
  document.getElementById('where_to_render')
);
```

2.13 Comments in JSX Code

If you need to comment out code in JSX templates you can do this:

- Original line of code:

```
<MyComponent />
```

- Commented line

```
{/* <MyComponent /> */}
```

2.14 Setting CSS Styles Using Classes

- Files containing CSS styles are imported into 'App.js' like this:

```
import './styles.css';
```

- Classes are added to JSX code using the className attribute:

```
render(){ return <div className=”main bordered”>...</div> }
```

- They can also be set using a variable like this:

```
let classNames = 'main';
classNames += ' bordered';
render(){ return <div className={classNames}>...</div> }
```

- Props or state can be used when creating the variable:

```
let classNames = 'main ';
if(this.props.isActive){ classNames += ' active'; }
```

```
if(this.state.isHidden}{ classNames += ' hidden'; }
render(){ return <div className={classNames}>...</div> }
```

Notes:

```
/* styles.css */
.main { background-color: lightgrey; }
.bordered { border: 1px solid black; }
.active{ background-color: lightblue; }
```

2.15 Setting CSS Styles Directly

- CSS Styles can be added directly to html using the 'style' attribute:

```
<h1 style=”margin-top:0px;” >...</h1>
```

- The 'style' attribute in JSX works differently:

```
render(){ return <h1 style={{marginTop:0}} >...</h1> }
```

- Multiple styles can also be used:

```
render(){ return (
       <h1 style={{marginTop: 0, marginBottom: 2}}
>...</h1>
```

- They can also be set using a variable like this:

```
let styles = {marginTop: 0, marginBottom: 2};
render(){ return <h1 style={styles}>...</h1>
```

- Note the differences:
  - Dashes cannot be used so use: `marginTop` instead of `margin-top`
  - Numbers are numbers so use: `0` instead of `0px`
  - Use a comma `','` in place of the semicolon `';'` between styles

2.16 JSX Escapes Values

- HTML content is sometimes retrieved from a server via an AJAX call
- But HTML code retrieved from a server is subject to XSS (Cross Site Scripting) attacks.
- JSX takes this into account and automatically escapes any dynamic content inserted via JSX expressions
- Take this variable which is populated via a network call

```
const html = "<span>Some Text</span>;
```

- You may want to insert it like this:

```
<p>initial text {html} Even more text...</p>
```

- But it will not have the desired effect. In this case it will display the text of the span tag inline with the rest of the data instead of considering it to be HTML.

```
initial text <span>Some Text</span> Even more text...
```

- You can get around this using the attribute dangerouslySetInnerHtml though it is probably better to retrieve data when making server calls rather than raw html.

2.17 Input Parameters

- Components can receive data through input parameters
- If you want to pass an object of a greeting in our previous examples, you can do it with the following code (the functional component syntax is used below):

```
function Hello(props) {
  return <div>Hello, {props.who}!</div>;

ReactDOM.render(<Hello who=”Dr.Who”/>,
document.getElementById('where_to_render'));
```

- Note that we use curly braces to inject the property value in the JSX-encode string: * Hello, {props.who}! ;*

2.18 Property Naming Convention

- React DOM uses the “camelCase” JavaScript property naming convention
- So, in JSX you need to refer to
  - *tabindex* as *tabIndex*, and
  - to CSS *font-name* as *fontName*

2.19 Properties Default to 'True'

- If you pass a boolean property without a value, it defaults to **true**
- So these two JSX expressions are equivalent:

```
<LoanApp approved />

<LoanApp approved={true} />
```

- Using the boolean properties without specifying its value is generally not a good programming practice, and it may cause problems with ES6

2.20 Creating Attributes with Spread Operator

- The *spread* operator (ellipsis) … allows you to pass an object as a parameter
- Each property of the passed so object will be mapped to a distinct property

```
const userObject = {firstName: 'Bob', lastName: 'Marley'};
return <Greeting {...userObject} />;
```

2.21 Expressions

- You can embed any JavaScript expression in JSX by wrapping it in curly braces, e.g.:

```
<Message msg={'DO NOT SHOUT'.toLowerCase() + "!" } />
```

- The passed value can be accessed inside the Message component as 'props.msg'
- The value of *props.msg* will be 'do not shout!'
- JavaScript functions can also be called inside expressions:

```
<Message msg={ getMessage().toLowerCase() + "!" } />
```

- In all cases the expression is evaluated before the data is passed.

2.22 Fragments

- The following code produces a transpile error:

```
const Comp1 = () => <li>One</li><li>Two</li>
```

SyntaxError: src/app.js: Adjacent JSX elements must be wrapped in an enclosing tag

- To fix this you might wrap the two elements with a div:

```
const Comp2 = () => <div><li>One</li><li>Two</li></div>;
```

- In cases where an extra div in the hierarchy might cause problems you can use React.Fragment to wrap the items instead:

```
const Comp3 = () =>
    <React.Fragment>
        <li>One</li>
        <li>Two</li>
    </ React.Fragment>
```

- The 'short syntax' for: **<React.Fragment></React.Fragment>**

  is: **<></>**

- When wrapped in this way the resulting component would consist of two elements without a parent. This makes sense when you are already providing a parent element in some other way, like this:

```
ReactDOM.render(<ul><Comp3 /><ul>,
document.getElementById('x'));
```

- For more information on Fragments see:

```
https://reactjs.org/docs/fragments.html
```

2.23 Summary

In this chapter we covered:

- What is JSX
- JSX Transpilation
- React Components and Elements
- Nesting of Elements
- JSX Code Comments
- Setting Styles and Classes
- Input Parameters
- Expressions
- Fragments

Chapter 3 - React Functional Component Concepts

Objectives

In this module we will discuss:

- Functional Components
- Displaying Lists
- Concept of State
- Props
- Immutability
- Virtual DOM
- Handling Events
- Component Lifecycle
- App Development Workflow

3.1 Functional Components

- This chapter includes React concepts that work with functional components.
- Many of these concepts will also apply to ES6 components (which we review in a later chapter)
- Example of a functional Component:

```
function About(props) {
  let styles = {marginTop: 0, marginBottom: 2};
  return (
    <div className="App-box bordered">
      <h1 className="App-header"
          style={styles}
      >About Component</h1>
      <div className="App-contents">
        <p>{props.msg}</p>
      </div>
```

```
            <div className='App-footer'>About Footer</div>
          </div>
        );
      }
```

3.2 Displaying Lists of Items

- Quite often you need to render elements of an array (list)
- The most common way is to use the JavaScript *map* function
- Example below uses ES6 's arrow function notation:

```
function List2Li(props) {
  const list = props.list;
  const listItems = list.map((x) => <li>{x}</li>);
  return ( <ul>{listItems}</ul> );
}

const list = [1, 2, 3, 4];
ReactDOM.render( <List2Li list={list} />, document.getElementById('where_to'));
```

- While the above code executes properly, you will see a warning in the JavaScript console to the effect that a *key* should be provided for list items (see the next slide for more details ...)

Notes:

The code above will print this bulleted list:

- 1
- 2
- 3
- 4

3.3 Keys in Lists

- React needs list items to be uniquely identifiable for the purposes of re-rendering them after a change
- It reserves the "key" property for this purpose

- The *key* property is part of the *props* object. It is only used internally by React
- The array index can be used if no other unique value is available for example when the array contains individual strings as opposed to objects with properties.
- Global uniqueness is not required. It is only required that keys be unique within the list being rendered

3.4 Example List With Key

- List code uses the JS array.map function:

```
const HelloKeyList = function(){
    const state = {pets: ["Cat","Dog","Turtle","Bird"] }
    const listHtml = state.pets.map(
            (pet, index) =>
            {return ( <li key={index}
>{index}-{pet}</li> )}
    );
    return ( <ul>{listHtml}</ul> );
}

ReactDOM.render(
    <div><h3>{title}</h3><HelloKeyList /></div>,
    document.getElementById('react-container')
);
```

3.5 State

- Data used by a react component is referred to as its "state"

```
state = { name: "jack" }
```

- State is immutable (cannot be changed)

```
state.name = "jill"; // NOT ALLOWED
```

- State Can be Replaced though

```
state = { name: "jill" } //whole object replaced
```

- Components must be re-rendered whenever state is replaced.

3.6 Types of State Data

- Data that makes up state can be of various types:
  - Domain data (books, customers, products, etc.)
  - App state (selected Item, progress, etc.)
  - UI State (displaying_modal, visible_form_page)

3.7 State Hierarchy

- State is hierarchical with some state data at the root level and other nested data

```
{
    customers: [ ... ],
    products: [ ... ],
    appState: {},
    ui: { form1: {}, form2:{} }
}
```

3.8 Lifting State Up

- Let parent hold state

```
state = { app_title: 'title-value', name: 'Jack' }
```

- Parent passes value down to child via attribute

```
<App><h1>{this.state.app_title}</h1>
    <SimpleText name={this.state.name} />
</App>
```

- Child displays attribute value

```
<SimpleText><h4>Name:</h1>
    <input type=text value={this.props.name} />
</SimpleText>
```

3.9 Props vs. State

- State is owned by a component
- Props:
  - Are passed into a component from its parent
  - Often represent data that is part of the parent's state
  - Sometimes represent functions of the parent component that, once passed down, can be called in the child

3.10 Immutability

The key to immutability is this: ***state is "replaced" not "changed"***

Take the object:

```
let joe = { name: "joe", phone:"5551212" }
```

If Joe's phone # changes, what do we do?

No (this changes the object):

```
joe.phone = "6661212";
```

Yes (this replaces the object):

```
newjoe = { name: "joe", phone:"6661212" }
joe = newjoe;
```

3.11 Immutability – Why?

- With immutability it is easier to determine when the object has changed.
- When state is treated as immutable checking if an object has been re-assigned is sufficient to determine that state has changed.

- Checking each property of the object is not required.

3.12 Virtual DOM and State

- What is the Virtual DOM?
  - A simplified version of the browser's DOM used programmatically by React to improve performance
- When is it used?
  - React maintains a virtual DOM based on an App's components.
  - When state changes React creates a new virtual DOM based on the new state
  - React diffs the new virtual DOM with the previous virtual DOM to determine changes
  - React applies the changes it found to the browser DOM

3.13 Setting state

- For React Functional Components
  - state is managed via one or more calls to the useState hook

```
const[ product setProduct] = useState( productToEdit );
const[ types setTypes] = useState( initialTypesList );
```

  - useState returns a variable representing the data as well as a setter function that can be used to update the data.
  - Calling the set function (setProduct or setTypes) also triggers a re-render of the component.
- Hooks like this one are covered in more detail in the Hooks chapter.

3.14 Handling Input Field Change Events

- Entering data into input fields updates their value (state). Components

must be re-rendered when state is updated.

- The input field:

```
<input type={'text'} onChange={handleChange}
  name={'name'} value={product.name} />
```

- onChange Handler

```
let handleChange = function(e){
  product[e.currentTarget.name]=e.currentTarget.value
  setProduct(product); // updates and triggers re-render
}
```

3.15 Passing Props to Components

- Pass a literal message:

```
<HelloMessage message="clean your room" />
```

- Pass a property:

```
<HelloMessage message={props.message} />
```

3.16 Passing Functions to Components

- In this example the parent passes the function 'doStuff' that needs to be executed to the child component.
- In Parent Component:

```
function App() {
  let doStuff=function(a){
    console.log('value: ' + a);
  }
  return (<div><ButtonTest handler={doStuff} /></div>);
}
```

- Child component:

```
let ButtonTest = (props)=>{
  return (
    <button onClick={()=> {props.handler('stuff')}} >ClickMe</button>
  );
}
```

3.17 Event Handling

- You can attach event handling functions to React elements
- Use *camelCase* for React event names
- React proxies the actual (native) event objects with its own "virtual" ones
- These events are referred to as synthetic events and are based on the W3C spec *https://www.w3.org/TR/DOM-Level-3-Events/* that defines UI Events which extend the DOM Event objects
- Synthetic events help make the event model work consistently across browsers

3.18 Event Handler Example

- This JSX below creates a button with an *onclick* handler:

```
function clickEventHandler(e) { … }

<button onClick={clickEventHandler}>Click!</button>
```

- This gets translated into the following React code:

```
React.createElement("button",{onClick:
clickEventHandler},"Click!");
```

  - Notice how the second parameter to "createElement" is no longer null. It is used instead to pass in a reference to the event handler function.

3.19 Event Binding - DOs

- Good – Use arrow functions, they autobind to "this"

```
<input type="button" value='Change'
onClick = {()=>this.handleChange()} />
```

- Good – Bind to "this" manually

```
<input type="button" value='Change'
onClick = { this.handleChange.bind(this) } />
```

3.20 Event Binding – Don'ts

- Bad - handleChange is called but has the wrong "this" context.

```
<input type="button" value='Change'
onClick={ this.handleChange } />
```

- Bad – handleChange is executed immediately, not bound. If handleChange modifies the component's state this can produce an infinite loop.

```
<input type="button" value='Change'
onClick={ this.handleChange() } />
```

3.21 Passing Parameters to Event Handlers

- Pass the event object:

```
<input type="button" value='Change'
  onClick = {(e)=>this.handleChange(e)} />
```

- Passing a parameter:

```
<input type="button" value='Change'
  onClick = {(e)=>this.handleChange(e, parm)} />
```

- The handler:

```
handleChange(event, parm) {
```

```
    console.log(event.target.value + ", " + parm );
  }
```

3.22 Component Life-cycle

- Components in React go through various stages as they are being displayed:
  - Initial state is set
  - JSX code is rendered
  - A shadow DOM is created and compared with the existing DOM for the component to determine how they differ
  - React renders the DOM for the component using the differences
  - Later, based on updated state or props, the cycle is repeated

3.23 Life-cycle 'Hooks' in Functional Components

- React 'Hooks' are used when life-cycle related code needs to be inserted into functional components.
- We will be reviewing various 'Hooks' in a later chapter.

3.24 App Development Workflow – 1/3

- **Start with:**
  - JSON data model
  - Mock ui design
- **Break up mock ui into components**
- **Define the component hierarchy**

- **Build a static version in react**
  - Ignore interactivity for now
  - Just build the components

3.25 App Development Workflow – 2/3

- **Define what constitutes "state"**
  - List all items of data in the app
  - Identify which items:
    - are static
    - change over the life of the app
    - depend on other items
  - Determine which items:
    - are state and at what level of the component hierarchy
    - are passed in to which components
  - Remember: which components hold state is likely to change as your application develops and this is OK

3.26 App Development Workflow – 3/3

- **Add inverse data flow:**
  - identify how data changes during app lifetime
    - User enters data or clicks a button
    - State is updated reactively from outside (push)
    - State is updated automatically based on time
  - Create functions at the proper level of hierarchy to affect state
  - Pass functions down the hierarchy to where they will be acted upon (to input component etc.)

3.27 Summary

In this chapter, we covered:

- Functional Components
- Displaying Lists
- Concept of State
- Props
- Immutability
- Virtual DOM
- Handling Events
- Component Lifecycle
- App Development Workflow

Chapter 4 - React Router v6.x

Objectives

Key objectives of this chapter

- Routing and Navigation
- Creating a react-router based project
- BrowserRouter Component
- Routes Component
- Route Component
- Route and Query Parameters
- Routing and Redux

4.1 Routing and Navigation

- Applications typically consist of multiple views
- Routing defines possible view transitions
  - Happens at design-time
  - Involves definition of routes
- Navigation executes specific view transitions
  - Navigational links are defined at design-time
  - Links are invoked by an end-user at run-time.

4.2 react-router

- The react-router and related packages allow developers to implement routing and navigation between React components.
- Several packages are available:
  - react-router - core package
  - react-router-dom - for use with web applications

- ◇ react-router-native - for use with react-native applications
- ◇ react-router-redux - for web apps that also use Redux
- ◇ react-router-config - for supports static routes for server-side rendering

- This chapter covers version **6.x** of **react-router-dom** which requires React version 16.8 or later.

4.3 Creating a react-router based project

Steps:

- Use the Create React App utility to create a basic React project.

```
npx create-react-app app-name
```

- Install react-router-dom v6.x

```
npm install react-router-dom@6 --save
```

- Edit 'index.js' and import BrowserRouter:

```
import { BrowserRouter } from 'react-router-dom'
```

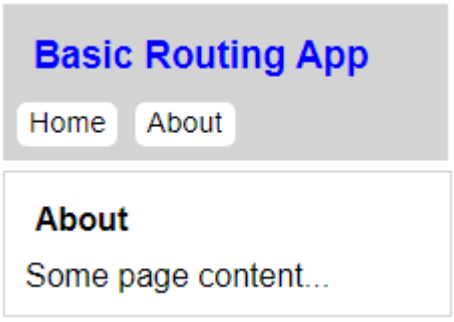
- In 'index.js' wrap the App component with BrowserRouter:

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

4.4 A Basic Routed Component

This sample app defines two routes

- One that shows the Home component
- One that shows the About component
- Code is on the next page



4.5 A Basic Routed Component

- Basic Routing Component Code:

```
const App = function(){
  return (
      <div>
        <h1>Basic Routing App</h1>
        <PageNav />
        <Routes>
          <Route exact path="/" element={ <Home /> } />
          <Route path="/about" element={ <About /> } />
        </Routes>
      </div>
  )
}
```

- Since the <BrowserRouter> component wraps around <App/> (in index.js) then it also wraps both the navigation and Routes sections.
- A common usage is to have <BrowserRouter> wrap an application's main <App/> component

Notes:

The complete listing:

```
import React from 'react';
import './App.css';
import Home from './home'
import About from './about'
import { Route, Routes, NavLink } from 'react-router-dom'

const PageNav = function () {
  return (<nav>
    <NavLink exact to="/">Home</NavLink> 
    <NavLink to="/about">About</NavLink><br />
  </nav>)
}

const App = function () {
  return (
    <div>
      <h3>Basic Routing App</h3>
      <PageNav />
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />
      </Routes>
    </div>
  )
}

export default App;
```

4.6 The Route component

- Is used to define routes
- All <Route /> elements must be wrapped by a <Routes /> element
- Main attributes:
  - **path** - the route is chosen when a navigation matches the path
  - **element** - when the path is matched the item defined here is displayed
- Example:

```
<Routes>
  <Route path="/" element={<Home />} />
  <Route path="/about" element={<About />} />
</Routes>
```

4.7 <Routes>

- All Route components with matching paths are displayed. This allows for more than one to be shown at the same time:
- If you need to have only one Route in a set of Routes to be displayed you need to wrap the Routes with the <Switch> component

```
<Router>
    <AppNav />
    <Routes>
      <Route exact path="/" element={<Home />}/>
      <Route path="/about" element={<About />}/>
    </Routes>
</Router>
```

- If a navigation path doesn't match an existing Route then:
  - The previously matched Route component is removed
  - No new component is displayed
- To fix this you can use the "*" path to create a default route that gets displayed when no other Route matches.

```
<Route path="*" element={<Error />} />
```

4.8 Redirecting with Navigate

- Sometimes you need to redirect from a path provided in a navigation to a different existing route
- You would use the <Navigate > component to do this:

```
<Routes>
  <Route path="/home" element={<Home />} />
  <Route path="/" element={<Navigate to="/home" />} />
  <Route path="/about" element={<About />} />
  <Route path="*" element={<Error />} />
</Routes>
```

- Here we are navigating from the root path "/" to the "/home" path
- Navigate needs to be imported before being used:

```
import { Route, Routes, Navigate, NavLink } from 'react-router-dom'
```

4.9 Navigating with <Link>

- Routes are activated when a user clicks on an anchor link <a>
- Anchor links for navigation are created by using the react-router-dom <Link> element

```
<BrowserRouter>
  ...
  <Link to="/">Home</Link>
  <Link to="/about">About</Link>
  ...
</BrowserRouter>
```

- <Link> elements must be nested inside <BrowserRouter> (wrapping BrowserRouter around the whole App takes care of this)
- The "to" attribute provides the path you want to navigate to
- The replace attribute is used when you want the new path location to replace the existing one in the browser's history instead of adding to it:

```
<Link to="/about" replace >About</Link>
```

4.10 Navigating with <NavLink>

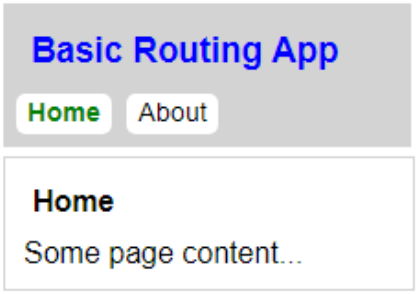
The currently selected link can be highlighted by using <NavLink> instead of <Link>

```
<NavLink to="/home" >Home</NavLink>
```

Then adding a CSS style for the "active":

```
.active{
    color: green;
    font-weight:bold;
}
```

A class name other than 'active' can be specified with the 'activeClassName' attribute



(the home link above is shown with the "active" styling)

4.11 Route Parameters

- Route Parameters can be used to pass data to a component during navigation.
- Route parameters are added to a Route's path like this:

```
<Route path="/details/:index" element={<ProductDetails />} />
```

- ":index" which appears in the above Route will hold whatever value is placed after '/details/' when navigating:
- For example for this link:

```
<Link to='/details/P001'>Details</Link>
```

- Index is equal to:

```
"P001"
```

4.12 Retrieving Route Parameters

- To make use of a Route Parameter that has been passed to it the destination component first needs to import the useParams hook:

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

- Then it must retrieve the parameter's value.

```
const ProductDetails = () => {
  const params = useParams();
  let index = params.index;
  let selectedProduct = getProduct(index);
  return (
    <div>{selectedProduct.name}, … </div>
  )
}
```

- Notice how the "index" route parameter value is retrieved using:

```
params.index
```

4.13 QueryString Parameters

- Query String parameters are another way to pass data to a component during navigation

```
<NavLink
  to='/sendmail?name=mark&email=mark@abc.com'
>Send Email</NavLink>
```

- This link will match the following Route, even with 'exact' specified:

```
<Route path='/sendmail' element={ <Mail />} />
```

- The parameter string can be retrieved via the useSearchParams hook:

```
props.location.search
(retrieves: "?name=mark&email=mark@abc.com" )
```

- To get individual parameters you can do this (or use a 3rd party lib):

```
function Mail() {
  const [searchParams, setSearchParams] = useSearchParams();
  return (<div className='page pageContent' >
    Name: {searchParams.get('name')}<br />
    Email: {searchParams.get('email')}
  </div>);
}
```

4.14 Using Router with Redux

- React-Router is compatible with Redux - both can be used together.
- In this case the Redux **<Provider>** wraps around the react-router **<BrowserRouter>** which then wraps the application's **Routes**

```
// in index.js
ReactDOM.render(
  <Provider store={store}>
    <BrowserRouter>
      <h2>React, Redux, Route App</h2>
      <nav>
        <Link to="/">Home</Link>
        <Link to="/about">About</Link>
      </nav>
      <Routes>
        <Route exact path="/" component={Home}/>
        <Route path="/about" component={About}/>
      </Routes>
    </BrowserRouter>
  </Provider>,
  document.getElementById('root')
);
```

- More sophisticated users may wish to add the *react-router-redux* package. This package incorporates the app's location URL into the Redux state thus allowing you to rewind navigational actions use Time Travel.

4.15 Summary

In this Chapter we Covered:

- Routing and Navigation
- Creating a react-router based project
- BrowserRouter Component
- Routes Component
- Route Component
- Route and Query Parameters
- Routing and Redux

Chapter 5 - State Management for React

Objectives
Key objectives of this chapter ■ React State Basics – State and Props ■ Managing State with Hooks ■ The Problem with Props and State ■ Redux State Library ■ Basic Rules of State Management ■ Types of State

5.1 React State Basics – Props and State

- Props
  - Stands for 'properties'.
  - Are data passed into components
  - Are like parameters passed into a function
  - Any type of data can be passed (string, number, function, object, etc.)
- State
  - 'state' is data managed within a component (directly or via Hooks)
  - Is like variables defined in a function.
  - Any type of data can be held (string, number, function, object, etc.)
- Both props and state are used to define how a component is rendered and what data it shows.

Notes

5.2 Props

- Here an object is passed as a property into the ToDo1 component:

```
let item = {'done': false, 'text': 'call home' };
<div>
  <ToDo1 item={item} ></ToDo1>
</div>
```

- Here is a ToDo1 component that uses the property:

```
function ToDo1(props) {
  return (
    <div className="box">
     ToDo-1: 
     <input type={'checkbox'} checked={props.item.done} />
       {props.item.text}
    </div>
  );
}
```

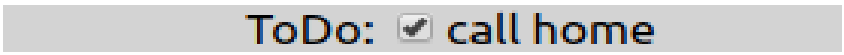
Notes

For class based components props are passed as parameters to the constructor.
Though if the constructor is simple like the one here you could leave it out as it will be created for you.

```
class ToDo1Class extends React.Component {
    constructor(props) {
        super(props);
    }
    render() {
        return (
            <div className="box">
                ToDo: 
                <input type={'checkbox'}
                    readOnly
                    checked={this.props.item.done} />
                {this.props.item.text}
            </div>
        )
    }
}
```

5.3 State in Class Based Components

- State is initialized in the constructor (for Class based components)
- Functional components are supposed to be stateless (but that's changing as we'll see later when we take a look at Hooks)



- In this component the state of the checkbox is held inside the component.

```
constructor(props){
    super(props);
    this.state = { done : false }
}
```

- Later, when the box is clicked on the state is updated using **setState**:

```
handleCBChange(event){
    const name = event.target.name;
    this.setState({ done : event.target.checked } );
}
```

- **setState** works fine to update the state inside the component but what if the data needs to be shared with another component?

Notes

If the data needs to be shared with another component then state should live outside the components. This is the type of thing that is done when using a state library like Redux.

5.4 Managing State with Hooks in Functional Components

- Functional components can't use constructors or member functions to manage state so instead they use Hooks.

```
function Counter() {
    const [count, setCount] = useState(0);
    return (
        <div className="box">{count} 
          <button onClick={() => setCount(count + 1)}>
                Click to Increment</button>
        </div>);
}
```

- The useState hook in the line “const [count, setCount] = useState(0);”:

- Create sa state variable named 'count'
- Creates a function 'setCount' that can be used to set the variable
- Sets the initial value of 'count' to 0

5.5 The Problem with Props and State

- With props and state we can:
  - Pass data from parent to child component for display
  - Manage state inside of a component
- But we do not have a good way to:
  - Share state data from child back to the parent
  - Share state data between side-by-side components
  - Easily manage data in one place and reuse it elsewhere
- For these reasons other methods of managing state have been created (for example: Redux)

Notes

5.6 Redux State Library

- Redux is an open-source library for managing application state
- When using Redux:
  - A **“Store”** is created at the application level
  - The Store is set as a property on the Provider component which then makes it available to the application

```
<Provider store={store}>
  <App />
```

```
</Provider>
```

- “**Reducer”** functions are created to manage state data in the store
- State is updated by passing “**Actions**” to the Reducer(s)
- Components are re-rendered each time the store is changed.

Notes

5.7 Redux Advantages

- Components can...
  - Access data without having to pass it in props
  - Update data without passing callback functions as props
- Centralizing data changes via actions and reducers...
  - Makes data manipulation code easier to test
  - Allows for 'time-travel' debugging
- Having state defined at the application level...
  - Makes it easier to share data between components. (for example: list and detail information for customers, products, and other data sets)
  - Simplifies data updates

Notes

5.8 Redux Disadvantages

- Requires a lot of coding to get started: Actions, Reducers, mapStateToProps, mapDispatchToProps, Connect, Provider, createStore, combineReducers, ...

- Is overly complex for small applications
- Is an overly complex way to store simple state like checkbox values, modal dialog flags, etc.
- Certain disadvantages can be mitigated by managing simple state with basic React methods and Hooks and complex state with Redux.

Notes

5.9 Basic Rules for State Management

- Use props to pass display-only data from parent to child component.
- Use props to pass functions from parent to child when actions on the child component need to update parent data
- Use local React state management (setState or Hooks) when data does not need to be shared outside the component
- Use a state management library like Redux for application-wide data and when data needs to be shared between adjacent components
- Next we will take a look at various types of data encountered in an application and discuss how to handle state management for each.

5.10 Types of State

- Lets take a look at the types of state in a typical application.
- Types of State
  - Data State
  - Communication State
  - Control State
  - Session State
  - Location State
- We will review these types over the next few pages

Notes

5.11 Data State

- Data State Includes:
  - business data your app manages
  - for invoicing app this includes the invoices
  - for a todo app this includes the todos
  - for a chat app this includes the conversation text
  - etc...
- Each piece of data has:
  - **Type**: For example an Invoice class
  - **Selector**: For example a REST URI string like “/invoice/{id}”
- To allow sharing of data between components and centralized update of data, data state is best managed at the application level using a state management library like Redux

Notes

5.12 Communication State

- Communication State Includes:
  - 'loading indicator/spinner' state
  - error messages
  - 'Promise active' state
  - the data 'type/selector' is state
  - 'expected change' from network calls

- implement an array of the above state for each Request
  - can have its own reducer
  - think of this as 'invisible state', its managed behind the scenes, not mapped to a specific screen
- Being global to the application communication state is best managed at the application level using a state management library like Redux

Notes

5.13 Control State

- Control State Includes
  - text entered in forms
  - button click choices
  - scroll direction and extent
  - selected list items
  - selected tree view nodes
- is specific to a given component/view
- does not need to be shared with other components
- is not stored in the URL or browser history (not route info)
- Control state which does not need to be shared between components can be managed using default React state management techniques (for example: setState or Hooks)

Notes

5.14 Session State

- Session State Includes
  - information about who is using the application
  - is only read when the component is mounted
  - is used to initialize the view
- May be a copy of control state (ie. expanded tree view node list)
- Examples include: ui state, userId, permissions, preferences, etc.
- When required by multiple components session state is best managed at the application level using a state management library like Redux
- Session state that is not customized and persisted can be managed using standard React state techniques.

Notes

5.15 Location State

- Basic location State Includes
  - Directions for accessing part of an application (URL...)
  - Address bar contents
  - Browser history contents
- Basic location State can be saved as a bookmark and used for navigation
- Extended location State Includes
  - List of accessible screens/ branches/ locations in the app
  - List of disabled screens/ branches/ locations in the app
- Change in location state can produce side effects
- Basic location state is typically managed by the Browser and the Router component.

- Extended location state must be accessible from any component and therefore should be managed at the application level.

Notes

5.16 Location State Side Effects

- Keep in mind that navigating to a new location can have side effects which include:
  - Mounting of container components which causes Control state to change
  - Triggering of HTTP requests which cause Communication state to change
    - Also Data state often changes when Comm requests return
  - Additional location changes when routes are guarded (for example – redirect)

Notes

5.17 Summary

- In this chapter we covered:
  - React State Basics – State and Props
  - Managing state with Hooks
  - The Problem with Props and State
  - Redux State Library

- Basic Rules of State Management
- Types of State

Chapter 6 - Building React Apps with Redux

Objectives

Key objectives of this chapter

- Actions
- Reducers
- Store
- Data Flow
- Immutability
- Copying Objects and Arrays
- Using Redux with React

6.1 Redux

- Redux is a state management library.
- It can be used in various types of applications including React.
- It allows you to request state changes from anywhere in an application.
- All state changes are made in a single location - the reducer.

6.2 Redux Terminology

- Actions - are objects that are used to request state changes
- Reducers - include code that implements changes based on actions.
- Store - is the location where application data (state) is kept
- Data Flow - data flows down the component hierarchy, events flow up

6.3 Redux Principles

- Redux is used to manage the Data Store
- The data store is immutable (can be replaced but not modified)
- Actions define changes to the store
- Reducers execute Actions and return a new store
- Updates to store trigger re-render of components

6.4 Redux: Actions

Actions:

- Describe user intent
- Can include payloads of information that send data from your application to your store.
- Are formatted as plain JavaScript objects
- Have at least a "type" property
- Include additional properties as required to implement an action.
- Are executed by pure functions called "reducers"

Example Action:

```
{type: 'ADD_BOOK', text: 'book title'}
```

6.5 Redux Action Types

- Types indicate the desired action

  type: 'ADD_BOOK'

  type: 'UPDATE_BOOK'

  type: 'DELETE_BOOK'

- Implementing type as a constant, promotes consistency and reduces errors:

```
const ADD_BOOK = 'ADD_BOOK';
{
  type: ADD_BOOK,
  text: 'book title...'
}
```

6.6 Action Creators

- Action Creators are functions that return Redux Action objects:

```
function deleteBook( id ){
    return { type: 'DEL_BOOK', id };
}
```

6.7 Dispatching Actions

- Actions are realized by dispatching them:

```
let action = deleteBook(5);
dispatch(action);
```

- Bound action creator's are functions that include Dispatching:

```
const boundDeleteBook = (id) =>
dispatch(deleteBook(id))
```

- Calling a bound action creator:

```
let id = 5;
boundDeleteBook(id);
```

6.8 Data Flow Basics

- Data flows down
  - down into components
  - most components receive data via props

- Actions flow up
  - up from components
  - typically implemented via event emitters

6.9 Redux Reducers

- Reducers are pure functions that:
  - Take:
    - The previous state
    - An Action
  - Produce a new state

- The term 'reducer' takes its name from the callback method of the Array's 'reduce' function which has a similar effect:

```
Array.prototype.reduce(reducer)
```

6.10 Redux Reducers

- Reducers should not:
  - mutate their arguments

- call API's or implement routing
- call non-pure functions

6.11 Pure Functions

- Pure functions, if called with the same arguments on different occasions must return the same results
- Pure functions should do nothing other than produce a new state. That means no side effects.
- Pure function example:

```
function add(a, b){ return a + b; }
```

- Non-Pure Function examples:

```
Date.now()
Math.random()
```

6.12 Reducer Example

```
function personApp(state, action) {
 switch (action.type) {
   case 'CHANGE_NAME':
     return Object.assign(
                  {}, state, { name:  action.name });

   case 'DEFAULT_NAME':
     return Object.assign(
                  {}, state, { name: 'Default' });

   default:
 return state
 }
}
```

6.13 Returning Default State

- When called for the first time and state is undefined the reducer should return an initial default for the state
- To do so we can update the previous example like this:

```
const defaultState = { name: "none" };

function personApp(state, action) {
  if(typeof state === 'undefined' ){
    state = defaultState;
  }
  switch (action.type) {
    ...
  }
}
```

6.14 Creating a Development Environment with create-react-app

- Redux requires a development setup that supports separation of code into multiple files and the loading of those files as needed. This is in addition to the ES6 and JSX transpilation requirements.
- **create-react-app** is a node.js based tool that allows you to create a complete development setup in just a few steps
- Steps:
  - npm install create-react-app
  - create-react-app my-app
  - cd my-app
  - npm start
  - open url: http://localhost:3000
- A development environment setup in this way includes

- ES6 & JSX Transpilation
- Automatic transpilation when files are changed
- The WebPack development server
- Automatic updating of browser after file changes

6.15 Using Redux with React

- Install Redux into your project

```
npm install --save redux
```

- Install React bindings

```
npm install --save react-redux
```

- React-Redux connects React components to the Redux store. When the store state changes it calls a function to re-render affected components.
- If issues occur during install you may need to use the –no-optional switch with the above commands.

6.16 Initializing the Store

The store maintains application data (state).

- Initialize Store:

```
let store = createStore(reducer)
```

- Store functions:

```
dispatch(action)
subscribe(listener)
getState()
replaceReducer(nextReducer)
```

6.17 Immutability

- State objects are immutable
- Don't modify objects directly
- When data changes:
  - Create a new object with the new data
  - Assign the new object to the existing state variable

    (thus replacing the old object)

6.18 Benefits of Immutable State

- It's easy to see WHERE in your code state changes occur because they are always the result of an action being executed by the reducer.
- Immutability makes it easy for the framework to determine WHEN state has changed.
- A side benefit is that changes can easily be replayed or rolled back.

6.19 Mutability of Standard types

- These JavaScript types are by nature immutable meaning that their values are replaced not modified:
  - Number,
  - String,
  - Boolean
- These JavaScript types are by nature mutable meaning that the values they hold can be directly modified:
  - Objects,
  - Arrays

6.20 Copying Objects in JavaScript

- Reducers in Redux need to create new copies of objects that include new state information.
- This is best done using the following ES6 function:

```
Object.assign(target_object, ...data_sources)
```

- Data sources are applied to the object in the order in which they are listed
- A typical use looks like this:

```
Object.assign({}, old_state, {name:'new_value'});
```

6.21 Copying Arrays in JavaScript

- Take the array:

```
todos = [{item:'buy milk'}]
```

- When adding an item you might be tempted to do this:

```
todos.push({item:'call mom'})
```

- This mutates the existing array and violates immutability
- Instead you should do this:

```
new_item = {item: 'call mom'};
todos = [...todos, new_item ];
```

- This uses the array spread operator (...array) to copy the original array elements into the new array.

6.22 One Store - Multiple Reducers

- One store is used to hold all application state but data is divided by usage

- Reducers are also divided by usage
- Each reducer handles a subset of actions
- Each subset handles changes to a subset of store data
- React always calls all the reducers in response to a dispatch call

6.23 Combining Reducers

- Individual reducers are combined like this:

```
import {combineReducers} from 'redux';
import todos from'./todoReducer';
import books from './bookReducer';

const rootReducer = combineReducers({
    todos: todos,
    books: books
});

export default rootReducer;
```

6.24 Components and Redux

Two Main Types of Components

- Container
  - Smart
  - Hold state
  - *Aware of Redux, Subscribe to Redux state*
  - *Focus on how things work*
  - *Dispatch Redux Actions*

- *Wrapped by React-Redux*

- Presentational
  - Do not hold their own state (not smart)
  - Implement child components
  - No dependencies / reusable
  - Gets state through props
  - *Unaware of Redux*
  - *Focus on how things look*
  - *Invoke Redux Actions through callbacks on props*

6.25 The React-Redux Package

React-Redux contains two parts:

- Provider:
  - Attaches app to the data store
- Connect:
  - Creates container components
  - Requires the following methods:
    - mapStateToProps
    - mapDispatchToProps

6.26 Wrapping App with Provider

- Provider:

- Wraps around your main App component
- Sets the store property
- Makes store available to all of App's components
- This is done once at the root of the project in the src\index.js file

- Example (in src\index.js):

```
<Provider store={this.props.store) >
    <App />
</Provider>
```

Notes

Full index.js file:

```
import React from 'react'
import { render } from 'react-dom'
import { Provider } from 'react-redux'
import { createStore } from 'redux'
import reducers from './reducers'
import App from './components/App'

let store = createStore( reducers );

render(
<Provider store={store}>
<App />
</Provider>,
document.getElementById('root')
);
```

6.27 mapStateToProps

- **mapStateToProps Example:**
  - Takes the state object as a parameter
  - Maps entire state or subset of state to props for use in a component

```
const mapStateToProps = (state) => {
  return {
    books: state.books,
    appState: state.appState
```

```
}
}
```

6.28 mapDispatchToProps

- **mapDispatchToProps()**
  - Used to map actions required by a component
  - Takes the store's "dispatch" method as a parameter.
  - Methods defined here can invoke actions that change state by calling dispatch

```
const mapDispatchToProps = (dispatch) => {
  return {
    handleClick: (event, item) => {
      dispatch( updateItem(item) );
    }
}
```

6.29 Using Mapped Properties and Methods

- Properties and Methods that are mapped in:

```
mapDispatchToProps()
mapStateToProps()
```

- Are passed as parameters to the JSX template:

```
const BookList =
  ({ books, appState, handleClick }) => {
    return ( {/* JSX template... */} );
  }
```

6.30 Wrapping Components with Connect

- Connect is added to container-component files.

- Here "MyApp" refers to the JSX component constant in the same file.
- Connect is 'wrapping' the component.
- Example:

```
export default connect(
    mapStateToProps,
    mapDispatchToProps
)(MyApp);
```

6.31 Configure Store

- A typical **configureStore.js** file looks like this:

```
import {createStore} from 'redux';
import rootReducer from '../reducers';

export default function configureStore(initialState)
{
  return createStore( rootReducer, initialState );
}
```

- CreateStore could also be called directly in the src\index.js file:

```
let store = createStore( reducers );
```

6.32 Programming Advice - MultiTab Console

- End to end development can require command lines for:
  - Database
  - REST server
  - Web Application
  - Etc.

- When working with several command line utilities you may find yourself running out of screen space.
- A multi-tabbed console is useful for consolidating and managing command prompts
- One example is ConEmu:

```
https://conemu.github.io
```

- Other options can be found by googling the terms:

```
"multitab command line"
```

6.33 Summary

In this chapter we covered:

- Actions
- Reducers
- Store
- Data Flow
- Immutability
- Copying Objects and Arrays
- Using Redux with React

Chapter 7 - Using React Hooks

Objectives

Key objectives of this chapter

- Overview of Hooks
- Hook Rules
- The useState Hook
- The useEffect Hook
- The useContext Hook
- Additional Hooks

7.1 Functional Component Shortcomings

- ES6 Class based components can do some things that functional components normally can't:
  - Class based components can handle their own state
  - Functional components typically get their state from passed-in props
  - Class based components can run code after render to:
    - Fetch data
    - Subscribe to updates
    - Modify the DOM
    - Execute log statements
  - Functional components are simple functions that are meant to simply return JSX code.

7.2 Hooks Overview

- Hooks add capabilities to functional components.
- Hooks are React methods that can be called in functional components to:

- Maintain state
- Run code after the component renders

- Hooks allow developers to use functional components in more situations instead of converting them to Class based components.
- The most common hooks include:
  - useState – used for state management
  - useEffect – used to run code in conjunction with life-cycle events
  - useContext - used to access a React Context

7.3 Hook Rules

- Before we get started lets cover the rules regarding the use of Hooks.
- Hooks are functions in React. When calling these functions the developer must abide by these rules:
  - Only call hooks at the Top Level of your function not inside loops, conditions or in nested functions.
  - Only Call Hooks inside functional component functions or inside custom Hooks.
- React provides an ESLint plugin that will enforce these rules

```
eslint-plugin-react-hooks
```

Notes

7.4 React Linter Example

- React includes a linter that checks syntax before transpilation and will let you know when you break Hook rules.
- The code below violates the two Hook Rules

```
function someRandomFunction(){
```

```
  const[message, setMessage]=useState('some message...');
  // ...
}

function App(props) {
  if(props.x){
    const[message, setMessage]=useState('some message...');
  }
  return ...
}
```

- The following error messages will be output at compile time:

```
Error01: React Hook "useState" is called in function
"someRandomFunction" that is neither a React function component nor a
custom React Hook function.

Error02: React Hook "useState" is called conditionally. React Hooks
must be called in the exact same order in every component render
react-hooks/rules-of-hooks
```

Notes

7.5 Functional Component Props

- Below is a basic functional component that accepts props passed in by its parent.
- The component will be used as an example for us to explore Hooks.

```
function Parent() {
  return (
    <div>
      <Message1 msg='initial message'></Message1>
    </div>
  );
}
```

```
function Message1(props) {
  return (
    <div className="box">
      <h1 className="header" >Message1 Component</h1>
      <div className="contents">{props.msg}</div>
    </div>
  );
}
```

7.6 The useState Hook

- The useState hook allows functional components to manage state.
- Needs to be imported before it can be used:

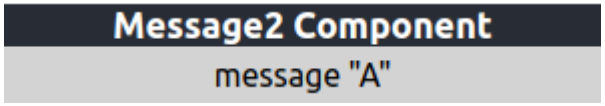
```
import React, { useState } from 'react';
```

- Syntax

```
const[message,setMessage] = useState('message text');
```

- The above code creates a variable and a method
  - The **message** variable holds the component's state
  - The **setMessage** function updates the state

7.7 Functional Component using the useState hook

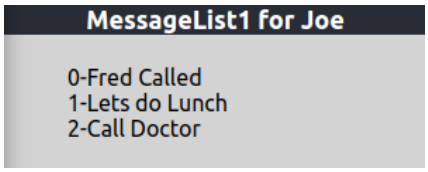


- A functional component that manages its own state using the useState hook:

```
function Parent() {
  return (<div><Message2></Message2></div>);
}

function Message2() {
  const[message,setMessage] = useState('message "A"');
  return (
    <div className="box">
      <h1 className="header" >Message2 Component</h1>
      <div className="contents">{message}</div>
    </div> );
}
```

7.8 useState with Multiple Variables

- useState can be called multiple times if needed to support different variables:



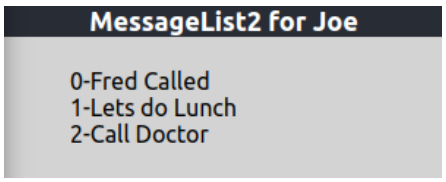
- Code:

```
const msgArray = ["Fred Called","Lets do Lunch","Call Dr"];

function MessageList1() {
  const [user, setUser] = useState('Joe');
  const [messages, setMessages] = useState(msgArray);
  const listHtml = messages.map( (msg, index) => {
    return (<li key={index} >{index}-{msg}</li>)
  } );
  return (
    <div className="box">
      <h1 className="header" >MessageList1 for {user}</h1>
      <div className="contents"><ul>{listHtml}</ul></div>
    </div>
```

```
  );
}
```

- See multiple useState() calls in bold

7.9 useState can also be used with Objects




```
const msgs = ["Fred Called", "Lets do Lunch", "Call Dr."];
const initialData = { 'user': 'Joe', 'messages': msgs };

function MessageList() {
  const [data, setData] = useState(initialData);
  const listHtml = data.messages.map( (msg, index) => {
    return (<li key={index} >{index}-{msg}</li>)
  } );
  return (
    <div className="box">
      <h1 className="header" >Messages for {data.user}</h1>
      <div className="contents"><ul>{listHtml}</ul></div>
    </div>
  );
}
```

- initialData here is an object
- Note: when calling setData() you need to provide the entire object.

7.10 The useEffect Hook

- The useEffect hook allows functional components to insert code into the component's life-cycle.
- UseEffect Needs to be imported before it can be used:

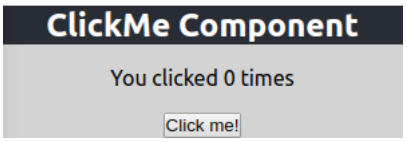
```
import React, { useState, useEffect } from 'react';
```

- Syntax

```
useEffect( myEffect );
```

- myEffect is a function
- React calls myEffect after flushing changes to the DOM
- myEffect is typically defined in-line and so has access to its props and state.
- MyEffect will be called after each render including the initial render
- useEffect can be called multiple times to register multiple effect functions

7.11 useEffect Hook Example



- Code:

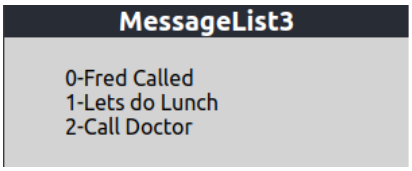
```
let message = 'na';

function ClickMe() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    message = `You clicked ${count} times`;
  });

  return (
    <div className="box">
      <h1 className="header" >ClickMe Component</h1>
      <p>{message}</p>
      <button onClick={()=>setCount(count + 1)}>Click me!
      </button>
```

```
    </div>
  );
}
```

- Note how the function passed to useEffect has access to both 'message' and 'count'.

Notes

7.12 Using useEffect Hook to Load Data



- Code:

```
function MessageList3() {
  const [msglist, setMsglist] = useState(['default msg']);
  useEffect(
    ()=>{
      getMessageList().then(
        (list)=>setMsglist(list)
      )
    }
  )
  ... // (code here the same as previous list examples)
}
```

- getMessageList() is a function that returns a Promise object that is waiting for a list of messages.
- The message 'default msg' will show briefly while the promise waits for the full message list to be delivered.

Notes

Full Code:

```
const messageList = ["Fred Called", "Lets do Lunch", "Call Doctor"];

function getMessageList() {
```

```
  let promise = new Promise(function (resolve, reject) {
    setTimeout(function () {
      resolve(messageList);
    }, 500);
  });
  return promise;
}

function MessageList3() {
  const [msglist, setMsglist] = useState(['default msg']);
  useEffect(
    ()=>{
      getMessageList().then(
        (list)=>setMsglist(list)
      )
    }
  )
  const listHtml = msglist.map( (msg, index) => {
      return (<li key={index} >{index}-{msg}</li>)
    } );
  return (
    <div className="box">
      <h1 className="header" >MessageList3 </h1>
      <div className="contents"><ul>{listHtml}</ul></div>
    </div>
  );
}
```

7.13 Restricting when useEffect is Called

- In the previous example the code that appears as the first parameter to useEffect() would be called after each render and if it changes any data that would trigger an additional render which could cause an infinite loop.
- Adding an empty array [ ] as the second useEffect() parameter allows you to avoid an infinite loop.

```
useEffect( ()=>{doSomething()}, [] )
```

- React checks the value of whatever is in the array with its value from the previous render and will not call your useEffect code if the value has not changed.
- Since the array is empty it never changes and so your code is only called once – after the first render.
- If you needed your code to be called whenever a specific state variable changed you would add that state variable inside the array.

7.14 The useContext Hook

- React Contexts provide state data to nested components without having to pass it down through the component hierarchy
- Contexts are created outside of components like this:

```
const AppContext = React.createContext({'articles':
articles_data });
```

- The Context is the accessed from inside a functional component:

```
 function Content(props){
    const ctx = useContext(AppContext);
    return (
        <ul>
        {ctx.articles.map( function(article){
            return (<Item article={article}
                          key={article.id}/>)
          }
        )}
      </ul>
    )
}
```

- The above code requires the following import statement

```
import React, { useContext } from 'react';
```

Notes:

Example that uses the useReducer hook (UseContextComponent.js):

```
import React, { useContext } from 'react';
import articles_data from './ArticlesData';

// create the context
const UserContext = React.createContext({'articles': articles_data });

export default function UseContextComponent() {
    // because we are using context
    // we no longer need to pass articles
    // into the Content component below
    return (
        <div className={'App'}>
            <h4>useContext Example</h4>
            <Content />
```

```
        </div>
    );
}

function Content(props){
    // use the context
    const ctx = useContext(UserContext);
    return (
        <ul>
        {ctx.articles.map( function(article){
            return (<Item article={article} key={article.id}/>)
          }
        )}
      </ul>
    )
}

function Item(props){
    return (
        <li>{props.article.title}</li>
    )
}
```

7.15 Additional Hooks

- The Hooks listed below are used less frequently and often to support specific edge cases. We will review them one at a time:
  - useReducer
  - useCallback
  - useMemo
  - useRef
  - useImperativeHandle
  - useLayoutEffect
  - useDebugValue (covered in the custom hooks chapter)
- The React Hooks API reference contains more information on these hooks:

  https://reactjs.org/docs/hooks-reference.html

7.16 The useReducer Hook

- This hook allows you to update state using a reducer and works in a way similar to the Redux state management library.

```
const[ state, dispatch] = useReducer(reducer,initialState )
```

- useReducer is typically used as an alternative to useState when managing state objects that contain multiple disparate properties
- The **'state'** variable provides access to the state data object
- **'dispatch'** is a function that replaces the current state object with a new one by an 'action' (and optionally some data) to the reducer

```
dispatch( action_object )
```

- An example action_object. The 'type' property is required.

```
let setSelectedAction = {'type': 'set-selected',
'selected':props.article.id }
```

- 'reducer' is a function that replaces the current state. It takes the current state and an action as parameters

```
function reducer(state, action){...}
```

Notes:

Example that uses the useReducer hook (UseReducerComponent.js):

```
import React, { useReducer } from 'react';
import articles_data from './ArticlesData';

/* reducer code */
const initialState = {'selected':-1, 'articles': articles_data };

function reducer(state, action) {
    console.log("in reducer");
    console.log(action);
    switch (action.type) {
      case 'set-selected':
        return {selected: action.selected, 'articles': initialState.articles };
      case 'unset-selected':
        return {selected: - 1, 'articles': initialState.articles };
      default:
```

```
        throw new Error();
    }
  }

/* components code */
export default function UseReducerComponent() {
    return (
        <div className={'App'}>
            <h4>useReducer Example</h4>
            <Content />
        </div>
    );
}

function Content(props){
    // employ useReducer to get state and dispatch
    const[ state, dispatch] = useReducer(reducer,initialState )
    return (
        <div>
        <ul>
        {state.articles.map( function(article){
            return (<Item article={article} key={article.id} dispatch={dispatch}
selected={state.selected}/>)
          }
        )}
      </ul>
      <button onClick={() => dispatch({type: 'unset-selected'})}>unselect</button>
      </div>
    )
}

function Item(props){
    return (
        <li
         onClick={()=>props.dispatch({'type': 'set-selected',
'selected':props.article.id})}
         className={(props.selected === props.article.id)? 'selected': ''}
        >{props.article.title}</li>
    )
}
```

7.17 An Example Reducer Function

- Here is an example of a reducer function that could be used with the useReducer hook:

```
/* reducer code */
const initialState = {'selected':-1,
                      'articles': articles_data };

function reducer(state, action) {
```

```
    switch (action.type) {
      case 'set-selected':
        return {'selected': action.selected,
                'articles': initialState.articles };
      case 'unset-selected':
        return {'selected': - 1,
                'articles': initialState.articles };
      default:
        throw new Error();
    }
  }
```

- Reducers are commonly implemented as shown here using a switch-case.

7.18 Calling and Using useReducer

- The following functional component uses useReducer to manage state:

```
function Content(props){
  const[state,dispatch] = useReducer(reducer,initialState);
  return (
        <div><ul>
        {state.articles.map( function(article){
            return (<Item article={article}
              key={article.id}
              dispatch={dispatch}
              selected={state.selected}/>)
          }
        )}
      </ul><button
        onClick={() => dispatch({type: 'unset-selected'})}
      >
      unselect</button>
      </div>
    )
}
```

7.19 The useMemo Hook

- The useMemo hook caches the results of expensive operations and returns the cached result when the operation is called with the same parameters as the last time it was called.

```
let result = useMemo(expensive_function(a,b), [a,b])
```

- The second parameter to useMemo is an array holding references to the parameters the function depends on.
- If the expression is called with the same parameters as the previous call then expensive_function is not called and the cached result is returned instead.
- An example of useMemo

Notes:

Example that uses the useMemo hook to bypass calling of an expensive displayCount function. (UseMemoComponent.js):

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

export default function UseMemoComponent() {

    const [count, setCount] = useState({'value':1});
    const [countstring, setCountstring] = useState("X");
    useEffect(()=>{console.log('UseMemoComponent was rendered: ' + count.value)})

    const displayCount = function(){
        console.log("displaycount called with: " + count.value);
        setTimeout(() => {
            setCountstring("X".repeat(count.value));
        }, 1000);
    }

    useMemo(displayCount,[count.value]);

    return (
        <div className={'App'}>
            <h4>useMemo Example</h4>
            <div className={'indent'}>
                <p>You clicked {count.value}, {countstring} times</p>
                <button onClick={() => setCount({'value': count.value + 1})}>
                    Click me
                </button>
                <button onClick={() => setCount({'value':1})}>1</button>
                <button onClick={() => setCount({'value':2})}>2</button>
```

```
                <button onClick={() => setCount({'value':3})}>3</button>
            </div>
        </div>
    );
}
```

7.20 useMemo Example

- Example of useMemo

```
const [count, setCount] = useState(0);

// define function that calculates/sets value
const calculateCount = function() {
  let newCount = ... // run the expensive operation
  setCount(newCount);
}

// run the calculation inside useMemo
useMemo(calculateCount, [count]);
```

- The above code is placed inside a functional component.

7.21 The useCallback Hook

- The useCallback hook can be used to avoid unnecessary renders of components that depend on functions.
- Functions are often defined inside of functional components like this:

```
const doSomething = () => { return someValue }
```

- Functions like this are re-created and assigned a new variable reference every time the component is rendered.
- Although the function hasn't really changed its reference variable has and that will force any sub-component that depends on the function to be re-rendered.

```
<SubComponent clickHandler={doSomething} />
```

- Technically the useCallback hook is caching the created function (as

opposed to useMemo which caches the results of calling a function)

Notes:

Example that uses the useCallback hook to cache the doSomething function and avoid re-rendering of the CommentComponent. (UseCallbackComponent.js):

```
import React, { useState, memo, useCallback, useEffect } from 'react';

export default function UseCallbackComponent() {

  const [count, setCount] = useState(10)
  const handleClick = () => setCount(count + 1)
  const someValue = "someValue"

  // This version causes CommentComponent to re-render on button click
  // const doSomething = () => { return someValue }

  // this version does not cause the Comment component to re-render
  const doSomething = useCallback(() => { return someValue }, [someValue])

  return (
    <div className={'App'}>
      <h4>useCallback Example</h4>
      <div className={'indent'}>
        <CountComponent count={count} handleClick={handleClick} />
        <CommentComponent doSomething={doSomething} />
      </div>
    </div>
  )
}

const CountComponent = ({ count, handleClick }) => {
  return (
    <div>
      <div style={{ paddingTop: "10px", paddingBottom: "10px" }}>
        The count value is {count}!
      </div>
      <button onClick={handleClick}>Increment Counter</button>
    </div>
  )
}

const CommentComponent = memo((props) => {
  useEffect(() => console.log('CommentComponent did re-render'))

  return (
    <div style={{ border: '1px solid black', marginTop: 5, width: 200 }}>
      <span>Comment:</span>
      <p>Here is a static comment</p>
    </div>
```

```
  )
})
```

7.22 useCallback Example

- This component uses useCallback to avoid unecessary rerendering of the CommentComponent:

```
export default function UseCallbackComponent() {
  const [count, setCount] = useState(10)
  const handleClick = () => setCount(count + 1)
  const someValue = "someValue"
  const doSomething = useCallback(
           () => { return someValue }, [someValue]
         )

  return (
    <div className={'App'}>
      <h4>useCallback Example</h4>
      <div className={'indent'}>
        <CountComponent count={count}
                        handleClick={handleClick} />
        <CommentComponent doSomething={doSomething} />
      </div>
    </div>
  )
}
```

- Note how CommentComponent depends on the doSomething function.

7.23 The useRef Hook

- useRef can be used to reference to a DOM element. The element can then be manipulated using the reference.

```
const inputfield = useRef(null);
```

- The reference is associated with a DOM element like this:

```
<input ref={inputfield} type={'text'} />
```

- The DOM element is available through the references' 'current' property:

```
let element = inputfield.current
```

- Then we can manipulate the DOM element, in this case an input field, from other code in our component:

```
const handleBtnClick = () => {
  inputfield.current.value = "new value";
};
```

- We can even use the reference from within a useEffect function

```
useEffect( ()=>{
    let element = inputfield.current;
    element.style.color = 'red';
    element.style.width = '230px';
  }
)
```

Notes:

Full example of a component that uses useRef (UseRefComponent):

```
import React, { useRef, useEffect } from 'react';

export default function UseRefComponent() {

    const inputfield = useRef(null);
    const textvalue = useRef('Initial value...');

    const handleBtnClick = () => {
        inputfield.current.value = "Some other content for the textarea ...";
        console.log(inputfield.current)
    };

    useEffect(
        () => {
            let element = inputfield.current;
            element.value = textvalue.current;
            element.style.color = 'red';
            element.style.width = '230px';
        }
    )
    return (
        <div className={'App'}>
            <h4>useRef Example</h4>
```

```
            <div className={'indent'}>
                <input ref={inputfield} type={'text'} />
                <br />
                <button onClick={handleBtnClick}>Replace text</button>
            </div>
        </div>
    );
};
```

7.24 Using useRef to Hold Values

- While useRef is often used to allow manipulation of DOM elements it can also be used simply to hold a value.

```
const textvalue = useRef('some text value');
```

- In this case the variable is not associated with a DOM element.
- The value is accessed via the references' 'current' property:

```
<p>{textvalue.current}</p>

textvalue.current = 'new value';
```

- The reference variable:
  - Can be referenced in code of the functional component
  - Can be modified
  - Keeps its value between renders
  - Does not trigger a new render if it is changed

7.25 The useImperativeHandle Hook

- This hook is used in a React component when exposing an internal element reference to a parent component.
- It allows developers to create methods on the reference that the parent component can then call. The code below adds a 'focus()' method to a child component's 'ref' property that calls focus() on the child's own internal inputRef.

```
useImperativeHandle(ref, () => ({
  focus: () => {inputRef.current.focus();}
}));
```

- The Parent creates its own ref variable and passes it to the child.

```
const customInputRef = useRef();

<CustomInput ref={customInputRef} />
```

- The created methods can be called to manipulate the child component's own internal reference.

```
customInputRef.current.focus();
```

- The above line of code in the parent component sets focus to an input field that is part of a custom input component on the form.

Notes:

This technique is used when the developer wants to control what can be done with the reference. If all you need is to have the internal reference exposed to the parent (along with all its unfiltered capabilities) you can do that with something called a forrwardRef.

```
const CustomButton = React.forwardRef((props, ref) => (
  <button ref={ref} className="FancyButton">
    {props.children}
  </button>
));

// You can now get the fully capable ref in a parent component like this:
const ref = React.createRef();
<CustomButton ref={ref}>Click me!</CustomButton>;
```

7.26 useImperativeHandle Hook Example

- The custom input component:

```
let CustomInput = function(props, ref) {
  const inputRef = useRef(null);
  useImperativeHandle(ref, () => ({
    focus: () => {inputRef.current.focus();}
  }));
  return <input ref={inputRef} />;
}
```

- The parent form component:

```
export default function FormComponent() {
  const customInputRef = useRef();

  const handleBtnClick = () => {
    customInputRef.current.focus();
  };

  return (
    <div className={'App'}>
      <h4>useImpHandle Example</h4>
      <div className={'indent'}>
        <CustomInput ref={customInputRef} />
        <br />
        <button onClick={handleBtnClick}>set focus</button>
      </div>
    </div>
  );
};
```

- Clicking the button on the parent form component sets focus to the referenced input field.

Notes:

Full code for the useImperativeHandle example (UseImpHandleComponent.js):

7.27 The useLayoutEffect Hook

- The useLayoutEffect hook works just like the useEffect hook except that its function is called sooner and synchronously.

```
useLayoutEffect( function_to_run, [deps] )
```

- useLayoutEffect works best and can reduces flickering with functions that contain a limited amount of code that modifies the DOM directly.
- useEffect tends to be used with functions that work behind the scenes, do not modify the DOM directly, and which may be asynchronous in nature like data retrievals.
- Technically useLayoutEffect is called after React completes its own DOM mutations which allows you to make modifications based on what React has already rendered.
- Care should be taken not to run lengthy calculations inside of the useLayoutEffect button because they could block other UI updates.
- As a rule of thumb try useEffect first and switch to useLayoutEffect only if you experience UI glitches.

Notes:

Full code of a component that uses the useLayoutEffect hook (UseLayoutEffectComponent.js):

```
import React, { useState, useEffect, useLayoutEffect, useRef } from 'react';
import '../App.css';

export default function UseLayoutEffectComponent() {
    const boxref = useRef(null);
    const[count, setCount] = useState(0);

    const handleClick = ()=>{
        setCount(count + 1);
    }

    // log to console when useEffect runs
    useEffect(()=>{ console.log(new Date().getTime().toString()
                    + ' - useEffect() called: ' + count);
    })

    // Make DOM changes inside useLayoutEffect
    useLayoutEffect(() => {
        console.log(new Date().getTime().toString() + ' - useLayoutEffect() called:
' + count);
```

```
        const value = 5 * count;
        boxref.current.style.marginLeft = value.toString() + 'px';
    }, [count]);

    return (<div className={'App'}>
        <h4>useLayoutEffect Example</h4>
        <div className={'grey'} style={{display:'inline-block', padding:10}}>
            <div id={'box1'} ref={boxref} className={'box left blue
bordered'}>box1</div>
        </div>
        <br/>
        <button onClick={handleClick} >Click</button>
         <span>{count}</span>
    </div>
    );
}
```

7.28 Summary

- In this chapter we covered:
  - Overview of Hooks
  - Hook Rules
  - The useState Hook
  - The useEffect Hook
  - The useContext Hook
  - Additional Hooks

Chapter 8 - Creating Custom React Hooks

Objectives

Key objectives of this chapter

- Custom Hooks
- Creating a Custom useMessage Hook
- Using the Custom useMessage Hook
- Creating a Custom useList Hook
- Using the Custom useList Hook
- The Built-In useDebugValue Hook

8.1 Custom Hooks

- Custom hooks are methods that:
  - Have names starting with “use”
  - Can call useState and useEffect
- Custom hooks can be used to:
  - Streamline code that might otherwise clutter up your component function
  - Create hooks that can be reused in multiple components

Notes

8.2 Custom Message Hook

- Below is a custom hook that maintains the state of a list of messages and returns an individual message based on an id:

```
function useMessage(id) {
    const [msgList] = useState(messageList);
    let message = 'default message';
```

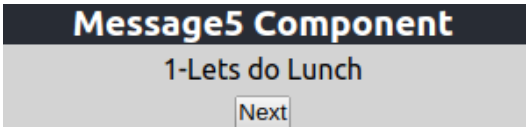
```
    if (msgList && id >= 0 && id < msgList.length) {
        message = msgList[id];
    }
    return message;
}
```

- Notice how it includes the 'useState' hook
- The above Hook can be used in a functional component like this:

```
let message = useMessage(count);
```

Notes

8.3 Using the Custom Message Hook



- The functional component below uses the useMessage custom hook

```
function MessageComponent() {
  const [count, setCount] = useState(0);
  let message = useMessage(count);
  return (
    <div className="box">
      <h1 className="header" >Message5 Component</h1>
      <div className="contents">{count}-{message}</div>
      <button onClick={()=>setCount(count+1)}>Next</button>
    </div> );
}
```

- The component code using the custom hook is easier to read and follow than it otherwise would have been

Notes:

The code in a version of the MessageComponent that **does not** use the custom hook is more complex.

```
function Message5b() {
  const [count, setCount] = useState(0);
  const[msgList,setMsgList] = useState(messageList);
  let message = '';
  if( msgList === undefined || count >= msgList.length ){
    message = 'default message';
  }else{
    message = msgList[count];
  }

  return (
    <div className="box">
      <h1 className="header" >Message5b Component</h1>
      <div className="contents">{count}-{message}</div>
      <button onClick={() => setCount(count + 1)}>Next</button>
    </div> );
}
```

8.4 A Custom useList Hook

- If you frequently display lists all the required code can be encapsulated into a custom hook.
- Here is an an example of such a hook:

```
function useList(initialState) {
  const [list, setList] = useState([]);

  useEffect( () => { setList(initialState);},
                     [initialState]
  )

  const list_component = function () {
    return (
      <ul>
        {list.map(
          (item, index) => {
            return <li key={index} >{item.name}</li>
          }
        )}
      </ul>
    )
  }

  return list_component;
```

```
}
```

- Lists passed to this hook must have items with a name property:

```
{ 'name': 'name text' }
```

- This hook returns a Component that displays the list passed into it.

8.5 Using the useList Custom Hook

- Using the useList custom hook simplifies our application component

```
export default function DisplayListsComponent() {
  const PetsListComponent = useList(petlist);
  const FriendsListComponent = useList(friendslist);

  return (
    <div className={'App'}>
      <h4>useList Custom Hook Example</h4>
      <PetsListComponent />
      <FriendsListComponent />
    </div>
  );
}
```

- Note how we're using the custom hook twice here. It is allowing us to display both the pets and the friends data.

Notes:

A complete listing of the custom hook and a component that uses it (UseListComponent.js):

```
import React, { useState, useEffect } from 'react';
import petlist from './PetsData.js';
import friendslist from './FriendsData.js';

function useList(initialState) {
  const [list, setList] = useState([]);

  useEffect(
    () => {
      setList(initialState);
    }, [initialState]
  )
```

```
  const list_component = function () {
    return (
      <ul>
        {list.map(
          (item, index) => {
            return <li key={index} >{item.name}</li>
          }
        )}
      </ul>
    )
  }

  return list_component;
}


export default function DisplayListsComponent() {
  const PetsListComponent = useList(petlist);
  const FriendsListComponent = useList(friendslist);

  return (
    <div className={'App'}>
      <h4>useList Custom Hook Example</h4>
      <PetsListComponent />
      <FriendsListComponent />
    </div>
  );
}

//PetsData.js
const pets = [
    {
        "id": 0,
        "type": "cat",
        "name": "Fluffy"
    },
    {
        "id": 1,
        "type": "dog",
        "name": "Spot"
    },
    {
        "id": 2,
        "type": "hamster",
        "name": "Tiny"
    },
]

export default pets


// FriendsData.js
const friends = [
```

```
    {
        "id": 0,
        "type": "best",
        "name": "Steve"
    },
    {
        "id": 1,
        "type": "good",
        "name": "Cindy"
    },
    {
        "id": 2,
        "type": "good",
        "name": "Darcy"
    },
]

export default friends
```

8.6 The built-in useDebugValue Hook

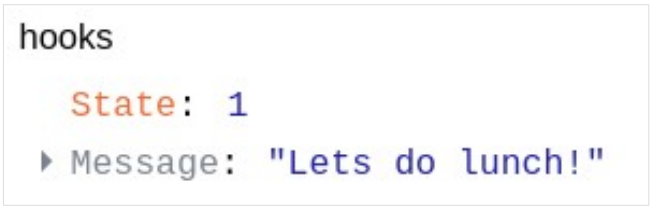
- The useDebugValue hook is used to provide extra data when viewing hooks in the React Developer Tools Chrome plugin.
- It allows you to display the current state of a custom hook.
- This is how you would use this hook in our custom message hook:

```
function useMessage(id) {
  const [msgList] = useState(messageList);
  let message = 'default message';
  if (msgList && id >= 0 && id < msgList.length) {
      message = msgList[id];
  }
  useDebugValue(message);
  return message;
}
```

- This will show the current message when viewing the custom hook in React Dev Tools

8.7 Viewing the Effect of the useDebugValue Hook

- React Developer Tools is a plugin for the Chrome browser that allows you to see behind the scenes of your React application.
- To see the hooks for a given component you would select the component in the tool's component hierarchy.
- When useDebugValue ***is used*** the hooks information looks like this:



- When useDebugValue ***is* not *used*** the hooks information looks like this:

- Notice the information missing after "Message:"

8.8 Summary

- In this chapter we covered:
  - Custom Hooks
  - Creating a Custom useMessage Hook
  - Using the Custom useMessage Hook
  - Creating a Custom useList Hook
  - Using the Custom useList Hook
  - The Built-In useDebugValue Hook

Chapter 9 - Unit Testing React with React Testing Library

Objectives

In this chapter we will discuss

- The React Testing Library
- Running Unit Tests
- Testing Asynchronous Code
- Building and Running Component Tests
- Snapshot Testing
- Query Functions
- Simulating Events
- Text Matching

9.1 React Testing Framework

- React Projects create with create-react-app come equipped with a testing framework from Facebook that's:
  - Invoked using “npm test”
  - JavaScript Based
  - Open Source BSD 3-clause license
  - Built on top of Jasmine
  - From Facebook
  - Works with other development frameworks as well (e.g. Angular)

9.2 Features

- Support for:
  - Component Testing
  - Mocks

- Asynchronous testing
- Spies, Stubs

- Test Runner Features
  - Snapshot testing
  - Code coverage
  - Interactive mode

9.3 Snapshot Testing

- Compare old and new user interfaces to detect changes
  - Manual: build user interface and compare visually
  - Automated: generate UI snapshot and compare against existing snapshots
- Snapshots (for React) are a serialized version of the React tree
  - Snapshots should be stored in configuration management (SCM)
- On subsequent runs test runner will render the UI and compare the serialized form (new snapshot) with the old snapshot
- The test will fail if the snapshots are different

9.4 Code Coverage

- The test runner has a built-in code coverage feature
- Add the `--coverage` flag to the npm test script

```
"test": "react-scripts test –coverage",
```

```
----------------|----------|----------|----------|----------|-------------------|
File            |  % Stmts | % Branch |  % Funcs |  % Lines | Uncovered Line #s |
----------------|----------|----------|----------|----------|-------------------|
All files       |    18.87 |        0 |    21.15 |    19.05 |                   |
 src            |    11.11 |      100 |    33.33 |    11.11 |                   |
  App.js        |      100 |      100 |      100 |      100 |                   |
  index-main.js |        0 |      100 |        0 |        0 |              9,14 |
  index-redux.js|        0 |      100 |        0 |        0 |    11,12,14,18,28 |
  index.js      |        0 |      100 |      100 |        0 |                 3 |
 src/examples   |    19.59 |        0 |    20.41 |    19.79 |                   |
  examples.js   |      100 |      100 |      100 |      100 |                   |
  hooks.js      |        4 |        0 |        0 |     4.05 |... 46,247,248,251 |
  state.js      |    71.43 |      100 |    69.23 |    71.43 |... ,80,90,103,104 |
----------------|----------|----------|----------|----------|-------------------|
Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        2.834s
Ran all test suites related to changed files.
```

9.5 Interactive Mode

One of the test runner's built-in features is its ability to watch for changes and selectively run (or not run tests)

- watch(default) – when a file changes it runs tests only on changed files
- You can also issue commands in interactive mode

```
 PASS  src/App.test.js
  ✓ renders without crashing (132ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        2.473s, estimated 3s
Ran all test suites related to changed files.

Watch Usage
 › Press a to run all tests.
 › Press f to run only failed tests.
 › Press q to quit watch mode.
 › Press p to filter by a filename regex pattern.
 › Press t to filter by a test name regex pattern.
 › Press Enter to trigger a test run.
```

Note

One non-trivial projects, test suites can become quite large. The ability to run tests only on changed files without constant reconfiguration makes it more likely that development teams will run tests frequently.

9.6 Projects created with *create-react-app*

- ***create-react-app*** is a command line utility for creating and maintaining React projects
- React projects created with create-react-app include the testing framework by default
- Installing create-react-app tool:

```
npx create-react-app simple-app
```

- Unit tests are placed in the same folder as the files they test:

```
mycomponent.test.js (tests mycomponent.js)
```

- Integration tests can be placed in a designated directory:

```
\tests\myintegration.test.js
```

- Unit tests are run with the following command:

```
npm test
```

9.7 Default App Component Test

- A simple test file(App.test.js') that tests the App component is generated along with each project.

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

test('renders learn react link', () => {
  render(<App />);
  const linkElement = screen.getByText(/learn react/i);
```

```
  expect(linkElement).toBeInTheDocument();
});
```

- Executing the test runner will run this test:

```
npm test
```

Note

This test is trivial, it just checks to see that the application renders properly.

9.8 Unit Tests

- Test the smallest 'unit' of an application
  - *function* or *method*
- Unit tests are independent (i.e. you can run one or several)
- Unit tests should be confined to the class being studied
  - *Mocks* represent other classes/services
- Unit tests should be applied before Integration tests

Note

Tests that cross unit boundaries (e.g. access a database, call a RESTful service) are integration tests. During an integration test where multiple *units* are involved it may be difficult to pinpoint precisely where failures occur. Validating the *units* before integration tests simplifies the process.

9.9 Anatomy of a Unit Test

- Natural language *like* syntax
- Filenames based on file being tested ***`*.spec.js`*** or ***`*.test.js`***
- Place tests in the source directory, `__tests__` or `__specs__` dirs
- Uses a `test()` or `it()` function with a callback to check the condition

- Test name should describe what's being testing:

  “isAvailable() should return false”

- How you would write a basic test:

```
it('isAvailable() returns false', () => {
  expect(isAvailable()).toBeFalsy();
});
```

Note

In JavaScript a value is *falsy* if it is *false, null, undefined, 0, NaN, '', ""*. A value is *truthy* if it is not *falsy*.

The above code uses an arrow function but regular functions also work:

```
it('isAvailable() returns false', function(){
  expect(isAvailable()).toBeFalsy();
});
```

9.10 Common Matchers

Matcher	Meaning
toBe	Strong equality (===), compares obj references
toEqual	Deep equality, compares object properties, (recursive)
toBeCloseTo	For floating point values allows for rounding
toBeTruthy, toBeFalsy	truthy/falsy
toBeNull	Checks for null values
toBeDefined, toBeUndefined	defined/undefined
toMatch	Tests a string against a regular expression
toContain	Array membership
toThrow	Tests for exceptions
not	Used to negate other matchers

	e.g. `expect(result).not.toBe(3);`

Note

This is only a partial list.

9.11 Combining Tests

- The **`describe()`** function combines tests
  - Uses a callback that calls **`test()`** or **`it()`** functions

```
describe( 'calculator', () => {
  it('should add',() => {
    expect(calc.add(3,2)).toBe(5);
  }
  test('should not divide by zero',() => {
    expect(calc.divide(3,0)).toThrow('divide by 0 error');
  }
});
```

Note

The `test` or `it` syntax may be used interchangeably.

9.12 Running Tests

- Run scripts inside your project

```
npm test
```

- "test" is defined in the package.json file:

```
"scripts": {
  "start": "react-scripts start",
  "build": "react-scripts build",
  "test": "react-scripts test",
  "eject": "react-scripts eject"
},
```

- Here test is defined to include code coverage statistics:

```
"scripts": {
  "start": "react-scripts start",
  "build": "react-scripts build",
  "test": "react-scripts test --coverage",
  "eject": "react-scripts eject"
},
```

9.13 Testing Promise based async code with 'done'

- Code that returns results asynchronously can be tested using the 'done' function as shown below.
- The 'done' function is always passed as a parameter into the testing function (but its only used when we need to test async code)

```
test('getResult returns with true', (done) => {
  getResult().then(
    (data)=>{
      expect(data).toBe(true);
      done();
    }
  )
});
```

- Here getResult() returns a Promise. We call 'then()' on the Promise and pass in a callback function. We call 'done' as the last line in the callback.
- 'done' lets the testing framework know when it can move on to the next test.

9.14 Setup and Teardown

- The testing framework will the following methods before and after tests.
- Add code to run before/after *all* tests to these methods:
  - `beforeAll(), afterAll()`
- Add code to run before/after *individual* tests to these methods:
  - `beforeEach(), afterEach()`
- Example:

```
beforeEach(
  ()=>{ console.log("in beforeEach");}
);
afterEach(
  ()=>{ console.log("in afterEach");}
);
```

- Note how the code above is placed inside of callback functions

9.15 react-testing-library

- The react-testing-library supports testing of components
- Installing the react-testing-library:

```
npm install --save @testing-library/react
```

- The react-testing-library makes use of jest-dom as well:

```
npm install --save @testing-library/jest-dom
```

- The following imports should be added at the top of you test file

```
import { render } from '@testing-library/react';
```

```
import '@testing-library/jest-dom/extend-expect';
```

- More information can be found here:

  https://testing-library.com/docs/react-testing-library/intro

9.16 A Simple Component Test

- Here is a simple component:

```
function Stuff() {
    return (
        <div className="box">
            <h3>Components</h3>
            <Stuff1></Stuff1>
        </div>
    );
}
```

- We will test it with this test:

```
test('renders welcome message', () => {
  const { getByText } = render(<Stuff />);
  expect( getByText('Components') ).toBeInTheDocument();
});
```

- The test renders the component to a simulated DOM and then checks if the text “React Examples” appears

9.17 A Simple Snapshot Test

- A snapshot is a serialized(JSON) version of the React tree
- This example renders a component `<Stuff/>` and compares it to a snapshot

```
test('matches snapshot', function(){
  const { asFragment } = render(<Stuff />);
  expect(asFragment()).toMatchSnapshot();
});
```

9.18 Running and Updating SnapShot Tests

- The first time this test is run a snapshot is saved to disk and you will see this message:

```
› 1 snapshot written from 1 test suite.
```

- The next time the test is run a new snapshot will be taken and compared against the one that was saved. If it matches then the test passes. If it does not match then you get this message:

```
› 1 snapshot failed from 1 test suite. Inspect your
code changes or press `u` to update them.
```

- Pressing 'u' will update the saved snapshot with the latest one.
- If the original snapshot was correct then you would adjust your Component code to fix the issue.

9.19 Building Component Tests

- Tests follow these steps:
  - Call render() to return utility functions

```
const { getByText } = render(<Stuff />);
```

  - Simulate events and user interactions (if needed)

```
fireEvent(node: HTMLElement, event: Event)
```

  - Add an assertion statement that tests the results

```
expect(getByText('Title Text')).toBeInTheDocument();
```

- We will go over details of these steps in the next few pages

9.20 Calling Render

- Before calling render you need to import the render function:

```
import { render } from '@testing-library/react'
```

- Render returns one or more properties that can be used to test the component:

```
const { getByText, container } = render(<Stuff />);
```

- You can pass parameters to components in render

```
render(<Stuff2 text='some text' />)
```

- Note that curly brackets are not needed around the text value 'text' (rather than {'text'} like in JSX)

Notes

Documentation for render can be found here:

https://testing-library.com/docs/native-testing-library/api#render

9.21 Render Properties

- Various properties can be returned from render:
  - Utility Properties:
    - baseElement – The base element of the rendered component
    - container – An instance of ReactTestRendererInstance,
    - debug – A method for printing the container (for example, to log)

        - rerender – function to rerender the component from within your test

    - Query Properties:
        - getByText – get the element that holds the given text
        - getAltText – get the element with a given alt attribute value
        - getByTitle – get the element with a given title attribute value

Notes

Documentation for queries can be found here:

https://testing-library.com/docs/dom-testing-library/api-queries#queries

Simulating Events

9.22 Simulating Events

- Before simulating events you will need to import 'fireEvent'

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

- fireEvent() can be called like this:

```
fireEvent.click( getByText('Click'), {button: 1});
```

- The second parameter above provides properties for the event object that gets passed when the event is triggered
- Various events can be simulated in this way:
    - `click, submit, keyUp, focus, blur, drag, drop, etc.`

Notes

Documentation for text matching in queries can be found here:

https://testing-library.com/docs/dom-testing-library/api-events

List of events that fireEvent can execute (fireEvent.event-name):

https://github.com/testing-library/dom-testing-library/blob/master/src/events.js

9.23 Testing Results

- Results can be tested with expectations (a type of assertion)
- Format for an expectation:

```
expect(htmlElement).matcher()
```

- The htmlElement can be obtained using a query returned by render()

```
let htmlElement = getByText('some text');
```

- An htmlElement matcher can be used to complete the test:

```
.toBeInTheDocument();
```

- The complete expression:

```
expect(getByText('text...').toBeInTheDocument();
```

Notes

Documentation for extended html Matchers can be found here:

https://github.com/testing-library/jest-dom#custom-matchers

9.24 Using Query Functions

- Query functions are returned by render()

```
const { getByText, queryByTitle } = render(<Stuff />);
```

- When executed they return HTML nodes, null or throw errors
- The results of calling query functions are tested using expections

```
expect(getByText('text...').toBeInTheDocument();
```

- Queries that start with "**get**" (getByText, getByTitle, etc):
  - Return the matching HTML node or array of nodes
  - or, ***Throw an error*** if no elements match
- Queries that start with "**query**" (queryByText, queryByTitle, etc):
  - Return the matching HTML node or array of nodes
  - or, ***return null*** if no elements match

Notes

Documentation covering queries can be found here:

https://testing-library.com/docs/dom-testing-library/api-queries

9.25 Text Matching

- Sometimes you need a query to match only a subset of the element text

```
getByText('substring', {exact:false})
```

- The above can retrieve the following element

```
<p>Find a substring in a larger string</p>
```

- The string passed in can also be a regex (Regular Expression)
- Other examples:

Query	***Matches***
`getByText(container, 'Hello World')`	`Full text 'Hello World'`
`getByText('World', {exact:false})`	`Any string with 'World' in it`
`getByText(/World/)`	`Any string with 'World' in it (regex)`
`getByText(/world/i)`	`Any string with 'World' or 'world' or 'WORLD', etc (regex)`

Notes

Documentation for text matching in queries can be found here:

https://testing-library.com/docs/dom-testing-library/api-queries#textmatch

9.26 Counter Component

- Counter Component

- Code:

```
export function Counter() {
    const [count, setCount] = useState(0);
    return (
      <div>Counter: <span>{count}</span> 
        <button onClick={()=>setCount(count + 1)}>
              Click</button>
      </div>);
}
```

- Clicking on the button increments the counter value. We will create a test to verify this functionality.

9.27 counter-test.js

- This test checks that clicking the button updates the counter:

```
test('clicking button updates counter', () => {
  const { getByText } = render(<Counter />);
  expect(getByText('0')).toBeInTheDocument();
  fireEvent.click(getByText('Click'),{button: 1});
  expect(getByText('1')).toBeInTheDocument();
});
```

- Explanation:
  - Line 01 Calls render() to get the 'getByText' query
  - Line 02 Checks for an element with the default value '0'
  - Line 03 Gets the button and clicks it
  - Line 04 Checks for an element with the updated value '1'

9.28 Summary

In this chapter we discussed

- The React Testing Library
- Running Unit Tests
- Testing Asynchronous Code
- Building and Running Component Tests
- Snapshot Testing
- Query Functions
- Simulating Events
- Text Matching

Chapter 10 - [OPTIONAL] Exception Handling in JavaScript

Objectives
Key objectives of this chapter ■ The **try … catch … finally** exception handling syntax in JavaScript ■ Nesting the try blocks ■ Throwing user-defined exceptions ■ Using the Error object

10.1 Exception Handling

- JavaScript handles exceptions using similar mechanisms found in most programming languages
- It uses the **try … catch** (and, optionally, **finally**) statement
- The **try** statement blocks a group of statements and establishes an exception interceptor which forwards exceptions to the **catch** bock should any exception be raised in those statements
- A **catch** clause contains statements for handling an exception if any is thrown in the *try* block; otherwise, the *catch* block is skipped
- The **try … catch** mechanism became full standard in ECMAScript 5.1

Notes:

For more information see:

http://www.ecma-international.org/ecma-262/5.1/#sec-12.14

10.2 Try Syntax

- The **try** statement consists of
  - A mandatory *try* block which groups one or more JavaScript statements which the developer considers as a critical code section that may throw an exception

- At least one *catch* clause or a *finally* clause
  - **try...catch** or **try...finally**
- Both *catch* and *finally* clauses
  - **try...catch...finally**

- **Note:** For the purpose of our discussion, we will view an exception as an action that sets a program's error state flag that is cleared when the catch block is entered

10.3 The Finally Block

- The **finally** block is executed after the *try* block and after the *catch* block, if an exception was raised
- The statements in the finally clause are always executed regardless of whether or not an exception was raised in the try block
- The *finally* block does not handle any errors that have not been caught in the preceding *catch* block
  - In other words, the *finally* block does not clear an error state flag, if it was set previously

10.4 The Nested Try Blocks

- The *try* blocks can be nested
- If an exception is raised in the inner *try* block and does not have a *catch* clause, the exception is forwarded to the enclosing *try* block and processed in its *catch* clause

```
try {
   // The enclosing try block
   try {
        // The inner try block
        // … some statements that may throw an error
    } finally{
        // Don't have a catch clause
   }
```

```
        } catch (e) {
          // Exceptions from the inner and enclosing try
blocks are handled here
      }
```

10.5 Exceptions Types in JavaScript

- JavaScript has a number of system (built-in) run-time exceptions that have not been universally accepted by all browsers:
  - **EvalError**
  - **ReferenceError**
  - **SyntaxError**
  - **TypeError**
  - **URIError**
- Developers can also define their own custom exceptions

10.6 The Throw Statement

- You can use the *throw* statement to raise a user-defined (custom) exception
  - It has simple syntax:
    - *throw* "Your error message";
    - throw <a number>, e.g. throw 12345;
- The throw statement can also be used to move (propagate) an exception outside of the catch block:

```
...
} catch (e) {
  // re-throwing exception:
  throw e;
}
```

10.7 Using the Error Object

- You can also create and throw user-defined exceptions using the generic **Error** object using the following syntax:
  - `throw new Error("Your error message");`
- There are browser vendor-specific Error object extensions (properties) that are not supported by all browsers:
  - **fileName**: Path to file that raised this error
  - **lineNumber**: Line number in file that raised this error
  - etc.
- You can check what properties of the *Error* object are supported in your browser by running this command:
  - `Error.prototype`

10.8 Summary

- JavaScript employs the **try … catch** (and, optionally, **finally**) statement to handle exceptions which is a similar mechanism found in most programming languages
- In addition to a number of system (built-in) run-time exceptions, developers can also define and throw their own exceptions using the throw statement and the *Error* object
- The *Error* object may have browser vendor-specific properties that are not supported by all browsers

Chapter 11 - [OPTIONAL] Web Storage, Web SQL, and IndexedDB

Objectives
Data Storage in HTML5 ■ Data Storage Options ■ LocalStorage ■ SessionStorage ■ IndexedDB

11.1 Data Storage

- Almost all applications need to store data
- Prior to HTML5 data was commonly stored at the browser in cookies
  - ◇ Cookies are hard to manage and not really suited for storage
    - Deleting them is awkward
    - "Leak" across windows/tabs
- The new HTML5 Storage APIs offer a clean interface to local storage in the browser

NOTES

Other pre-HTML5 storage strategies included:

- Flash storage
- IE UserData
- Google Gears
- Dojo Storage
- window.name(hack)

11.2 Data Storage Options

- Web Storage (AKA Local and Session storage)
  - Stores Key/Value pairs
  - Typically stores up to 5MB per application
  - Supported in all the latest browsers
  - Simple API, broad vendor support
  - Slow, No query language or schemas, no transactional support
- IndexedDB
  - Stores key / value pairs as well as JSON objects
  - Uses JavaScript to access the datastore through an object-based API
  - Supported by IE, Firefox, and Chrome
  - This is the approach that has been preferred and is progressing as an active specification with support from browsers increasing
- Web SQL Database
  - Originally meant to enable SQL database capabilities within the browser this specification is no longer being updated

Data Storage Options

The specification does not say how browsers should implement these storage mechanisms. The specifications do say that the storage could contain sensitive information so the browsers should protect this information in some way.

The specifications don't cover clearing this data although they do mention that this local data would be considered different than "cached data" and that browsers should provide users a way to clear all data that may be stored.

The 5MB limit for Web storage is just an arbitrary value suggested by the specification. Nothing in the specification requires browsers to have a limit or what it should be if they do.

11.3 Web Storage

- Stores simple string key/value pairs.

- If supported by the browser, there will be global variables '**localStorage**' and '**sessionStorage**'.
  - These objects support the exact same API
  - Data kept in sessionStorage cleared when the window or tab is closed.
  - Data kept in localStorage is persistent and services browser restarts. It could be cleared by the program or by the user.
- Both storage areas honor same origin policy. Data is stored in a domain name specific area. Only pages served from that domain can access the data.

11.4 Web Storage Programming Interface

Method	Description
`setItem(key, value)`	Saves 'value' with 'key' as reference
`getItem(key)`	Retrieves 'value' using its reference 'key'
`removeItem(key)`	Deletes 'value' from storage
`clear()`	Clears all stored values
`length`	Holds a count of items stored
`key(index)`	Retrieves a key from a list of available keys

- Browsers convert stored values to strings
- 'key' must be of string type
- Some browsers may prompt users for permission to store data
  - This is known as "white listing"
- The browser will throw a *QuotaExceededError* If storage is not allowed or if a site has reached the limit of what the browser will store.

Notes:

11.5 Web Storage Examples

- **Saving Data**

```
function btnStoreClick() {
    var key=document.getElementById("txtKey").value;
    var data=document.getElementById("txtData").value;
    localStorage.setItem(key, data);
}
```

- **Retrieving Data:**

```
function btnRetrieveClick() {
    var key=document.getElementById("txtKey").value;
    var data=localStorage.getItem(key);
    document.getElementById("txtData").value=data;
}
```

- **Listing All keys:**

```
function getKeys() {
    var keys=new Array();
    for (var i=0; i < localStorage.length; i++) {
        keys.push(localStorage.key(i));
    }
    return keys;
}
```

Notes:

You can also use a short-form “expando” (dot notation) syntax to access the storage objects:

```
sessionStorage.name='Bob';
var storedName=sessionStorage.name;
```

You should be careful not to set 'sessionStorage.key', this could replace the key() function!

11.6 Storing JavaScript Objects

- JavaScript objects should be converting to strings before storing them:

```
var user_obj = {name: 'Bob',email: 'bob@abc.com'}
```

```
var user_string = JSON.stringify(user_obj)
sessionStorage.setItem('bobData', user_string);
```

- After retrieving data items remember to convert them back to objects:

```
var user_string = sessionStorage.getItem('bobData')
var user_obj = JSON.parse( user_string );
```

11.7 IndexedDB

- IndexedDB is an:
  - object-based data store
  - supports in-order retrieval of records by index or key
- Dual mode operation:
  - Asynchronous API for the browser
  - Synchronous API for web workers

11.8 IndexedDB Terminology

- **Database**
  - Application domains (base url) each get an area where they can store multiple databases
  - A database is composed of one or more object stores, which hold the data stored in the database
- **Object Store**
  - An object store is the equivalent of a table in a SQL database
  - An object store consists of a list of records
  - Each record consists of key-value pairs
  - The records are sorted according to a key in ascending order
- **Index**
  - Records can be retrieved from an object store by key or index.

- Index values are mapped from properties of items in the store (e.g., customer.name).
- Index values are inserted automatically when items are inserted, updated or deleted from the store
- An object store can have multiple indexes

11.9 IndexedDB Terminology

- **Key**
  - A key, just like a primary key in a SQL database, must be unique
  - A key can automatically be generated or can correspond with a property of a JavaScript object that you specify using a key path
  - When creating an object store, you specify its key
- **Value**
  - A value normally corresponds with a custom JavaScript object
- e.g., Customer object store record
  - Key = 1
  - Value = {name: "Jenny", phone: "867-5309"}

11.10 IndexedDB Terminology

- **Request**
  - All read and write operations are performed within a transaction using Requests
  - Requests are made using asynchronous APIs which return an object similar to a promise object that can call an event handler when data is returned.
- **Transaction**
  - Transactions are associated with one of the following *modes:*
    - **readonly** - transaction can only read data. Used for queries.

        - **readwrite** - transaction can read, modify or delete data
        - **versionchange** - similar to a "readwrite" transaction, but can also create and remove object stores and indexes
    - A scope determining the object stores the transaction can interact with is also passed when a transaction is created.
    - Transactions commit automatically unless an error is thrown.
    - Multiple operations can return separate request objects that must be completed in order before a transaction can be finished

11.11 Getting indexedDB Objects

- The JavaScript object representing the indexedDB may be named differently in different browsers. Because of this your code should start with the following statements:

```
window.indexedDB = window.indexedDB || window.mozIndexedDB
  || window.webkitIndexedDB || window.msIndexedDB;

window.IDBTransaction = window.IDBTransaction  ||
  window.webkitIDBTransaction || window.msIDBTransaction;
```

- This will allow you to use a standard naming in the rest of your code

```
window.indexedDB          - the indexed db
window.IDBTransaction     - transaction
```

11.12 Opening a Database

- The first time you *open* a database, the database is created in the browser and a connection is established to it

```
var version = 1;
var request = window.indexedDB.open("ContactDB", version);
```

- A function attached to the onupgradeneeded event allows you to create object stores and databases

```
request.onupgradeneeded = function(event){ ... };
```

- If the open request fails, an *error* event handler is invoked. The error code can be obtained from the event object (e).

```
request.onerror = function(e){alert(e.target.errorCode);};
```

- For future open requests, a connection is just established to the database
- If the open request succeeds, a *success* event handler is invoked. A reference to the database can be obtained from the event object(e).

```
request.onsuccess = function(e){ db = e.target.result;}
```

11.13 Creating an Object Store

- The **onupgradeneeded** function will be called the first time you open a database. In it you can create your object store

```
request.onupgradeneeded = function(event) {
   db = event.currentTarget.result;

   // Create Contacts object store with autoincremented key
   var objectStore = db.createObjectStore("Contacts",
     { autoIncrement: true });

   // Create a non-unique index for a contact's name
   // property, so you can search for contacts based on the
   // contact's name
   objectStore.createIndex("ContactNameIndex", "name",
     { unique: false });
};
```

11.14 Inserting a Record

- The following code inserts a contact record into a Contacts object store:

```
var tx = db.transaction(["Contacts"], "readwrite");
var objectStore = tx.objectStore("Contacts");
var contact = {name: "Jenny", phone: "867-5309"};
var request = objectStore.add(contact);
request.onsuccess = function(event) {
   // get key for the inserted record
   var id = event.target.result;
   console.log(id);
};
```

- The code above, creates a transaction, gets a reference to the Contacts store, inserts a record and retrieves the record's autogenerated key

11.15 Retrieving a Record

- The following code retrieves a record using its key:

```
var tx = db.transaction(["Contacts"], "readwrite");
var objectStore = tx.objectStore("Contacts");
var key = 1;
var request = objectStore.get(key);
request.onsuccess = function(event) {
   var contact_object = event.target.result;
   console.log(contact_object.name);
};
```

- The code above, creates a transaction, gets a reference to the Contacts store and retrieves a record using its key
- More information on using indexedDB can be found here:

  https://developers.google.com/web/ilt/pwa/working-with-indexeddb

11.16 Summary

In this chapter we covered:

- Data Storage Options
- LocalStorage
- SessionStorage
- IndexedDB

Chapter 12 - [OPTIONAL] Asynchronous Programming with Promises

Objectives

Key objectives of this chapter

- The Problems with Callbacks
- Introduction to Promises
- Requirements for using promises
- Creating Promises manually
- Calling the Promise-based function
- Making APIs that support both callbacks and promises
- Using APIs that support both callbacks and promises
- Promisifying callbacks with Bluebird module
- Using Bluebird
- Error handling in Promise-based asynchronous functions

12.1 The Problems with Callbacks

- Asynchronous JavaScript, or JavaScript that uses callbacks, is hard to get right intuitively.
  - The code can end up with too many nested callbacks.
  - Due to nesting / indentation, the code looks like pyramid shaped

```
mymodule.getMainData(function (param) {
 // inline callback function ...

 getSomeData(param, function (someData) {
   // another inline callback function ...
   getMoreData(param, function(moreData) {
     // one more inline callback function ...
     getYetMoreData(param, function(yetMoreData) {
     // one more inline callback function ...
    });
```

```
  });
 });

 // etc ...
});
```

- Callbacks are the simplest possible mechanism for asynchronous code in JavaScript, but sacrifice the control flow.

12.2 Introduction to Promises

- Promises are a way to write asynchronous code that still appears though it is executing in a top-down way
- A promise is an abstraction for asynchronous programming.
- It's an object that provides for the return value or the exception thrown by a function that has to do some asynchronous processing
- Unlike callbacks, Promises allow more better control flow
- A promise object can be passed around and anyone with access to the promise can consume it regardless if the asynchronous operation has completed or not.

12.3 Requirements for Using Promises

- Callback-based function should be defined
- Create a wrapper function which returns a promise object (available in ES6)
- The promise object calls the callback-based function
- The promise object uses **resolve** and **reject** callback functions.
- **reject** is called when there's some error.
- **resolve** is called when data is available and there are no errors

12.4 Creating Promises Manually

- Define the callback-based function

```
function deposit(amount, callback) {
   let err = null;
   if(amount <= 0)
          err = "Invalid amount!";
   else
          balance += amount;
   callback(err, balance);
}
```

- Define a wrapper function which returns a promise object

```
function depositAsync(amount) {
  return new Promise(function(resolve, reject) {
    deposit(amount, function(err, data) {
      if(err)
        reject(err);
      else
        resolve(data);
      });
   });
}
```

12.5 Calling the Promise-based Function

- **then** method is called
- The method accepts two callback-functions as arguments
  - The first argument is utilized when data is processed successfully
  - The second argument is utilized when there's some error

```
let promise = depositAsync(500);

promise.then(
 function(data) {
    console.log("Balance is: \n" + data);
 },
 function(err) {
    console.log(err);
```

```
 }
);
```

- A better technique for executing the **then** method is to use then(…).catch(…)

```
promise.then(
 function(data) {
    console.log("Balance is: \n" + data);
 }
).catch(function(err) {
    console.log(err);
});
```

- then(…).catch(…) makes the syntax look more like synchronous calls
  - **try { } catch(...){ }**

12.6 Making APIs that support both callbacks and promises

```
function depositAsync(amount, callback) {
       if(callback)
              return deposit(amount, callback);
       else
         return new Promise(function(resolve, reject) {
              deposit(amount, function(err, data) {
                if(err)
                     return reject(err);
                else
                     resolve(data);
                });
          });
}
```

12.7 Using APIs that support both callbacks and promises

- If a callback is provided, it will be called with the standard callback style arguments.

```
depositAsync(500, function(err, data) {
       if (err)
```

```
            console.log(err);
      else
            console.log(data);
});
```

- If callback is not provided, it will return a promise object

```
depositAsync(500).then(
 function(data) {
    console.log("Sum is: \n" + data);
 }
).catch(function(err) {
    console.log(err);
});
```

12.8 Chaining then Method / Returning a Value or a Promise from then Method

- The **then** method can return a value or a promise object
- **then** method can be chained to obtain / resolve data from the previous **then** method
- In the example below, method1 and method2 returns promise objects. method3 returns a value.

```
let mainMethod = function() {
   let promise = new Promise(function(resolve, reject){
      resolve({data: 'mainMethod'});
   });
   return promise;
};

let method1 = function(data) {
      let promise = new Promise(function(resolve, reject){
            resolve({data: 'method 1 - completed'});
      console.log(data.data);
   });
   return promise;
};
```

```
let method2 = function(data) {
  let promise = new Promise(function(resolve, reject){
       resolve({data: 'method 2 - completed'});
       console.log(data.data);
  });
  return promise;
};

let method3 = function(data) {
  console.log(data.data);
};

mainMethod()
.then(method1)
.then(method2)
.then(method3)
.catch(function(err) {
    console.log(err);
 });

// Output
// method 1 - completed
// method 2 - completed
```

12.9 Promisifying Callbacks with Bluebird

- Implementing an asynchronous function which returns a promise object requires extra effort.
- Asynchronous functions can be generated automatically by using Bluebird module.
- Automatic generation of asynchronous functions is also called promisification.
- Other alternatives to Bluebird are Q, RSVP, ...

12.10 Using Bluebird

- Install Bluebird

```
npm install --save-dev bluebird
```

- Import Bluebird

```
let bluebird = require('bluebird');
```

- Promisify a module

```
// assuming you have let bank = require('bank');
let bank = bluebird.promisifyAll(bank);
```

- The above code will automatically generate depositAsync since there's a deposit function which makes use of a callback.
- Overall code will look like this

```
let bluebird = require('bluebird');
let bank = bluebird.promisifyAll(bank);

let promise = bank.depositAsync(500);

promise.then(
 function(data) {
 console.log("Sum is: \n" + data);
 })
.catch(function(err) {
 console.log(err);
});
```

12.11 Bluebird – List of Useful Functions

- promisify
  - Promisifies a single function

```
let readFile = Promise.promisify(require("fs").readFile);
```

- promisifyAll
  - Promsifies a whole library / module

```
let fs = Promise.promisifyAll(require("fs"));
```

- all

- Useful for when you want to wait for more than one promise to complete

```
let promises = [];
for (let i = 0; i < fileNames.length; ++i) {
    promises.push(fs.readFileAsync(fileNames[i]));
}
Promise.all(promises).then(function() {
    console.log("done");
});
```

- map
  - Simplifies push + all into a single function

```
Promise.map(fileNames, function(fileName) {
    // Promise.map awaits for returned promises as well.
    return fs.readFileAsync(fileName);
}).then(function() {
    console.log("done");
});
```

- some
  - Returns a promise that is fulfilled as soon as **count** promises are fulfilled in the array

```
Promise.some([
    ping("ns1.example.com"),
    ping("ns2.example.com"),
    ping("ns3.example.com"),
    ping("ns4.example.com")
], 2).then(function(first, second) {
    console.log(first, second);
});
```

- any
  - It is like **some** method, but with 1 as the **count** value.
- Complete API Reference
  - http://bluebirdjs.com/docs/api-reference.html

12.12 Benefit of using Bluebird over ES6 for Promisification

- promisify and promisifyAll make it very easy to promisify the callbacks.
  - Node v8 only supports util.promisify. This function isn't available in older version.
  - Bluebird supports both promisify and promisifyAll and works regardless of Node version.
- Bluebird is significantly faster than native ES6 promises in most environments
- Bluebird is more memory efficient than ES6 promises.
- Performing operations, such as iterating through an array, calling an async operation on each element is easier with Bluebird's utility functions.
- Bluebird works even when the browser doesn't support ES6.
  - This is nice if you're creating libraries that are intended to be run through 'webify' or 'webpack'
- Bluebird has a number of built-in warnings that alert you to issues that are probably wrong code. e.g. calling a function that creates a new promise inside a **then** method without returning that promise

12.13 Error Handling in Promise-based asynchronous functions

- As seen previously, error handling in done by utilizing:
  - **reject** parameter of the returned Promise object
  - The second parameter of **then** method
  - Best practice is to use the **catch** method

12.14 Summary

- Promises can be used to overcome the problems with callbacks
- Promises can be used to simplify asynchronous function calls
- Promises can be implemented manually or by using Bluebird.