REACT AND REDUX WITH TYPESCRIPT

CREDITS AND COPYRIGHT

Introduction to React

Written by John Paxton
pax@speedingplanet.com
for Speeding Planet
http://speedingplanet.com

FLIGHT CHECK

- Can you see my screen?
- Can you read my code?
- Can you hear me?
- Any distractions?

SCHEDULE

- Morning break
- Lunch break
- Afternoon break

CHAPTER 1: INTRODUCTION AND SETUP

CHAPTER PREVIEW

- Starting quickly
- Using create-react-app
- Creating a "Hello, world" component
- Testing our component

STARTING QUICKLY

- Our objective is to get into the guts of React quickly
- In the next chapter, we will talk about React itself
- For now, we want to get to writing code quickly!

STARTING QUICKLY, CONTINUED

- Open a command prompt or terminal window and enter these commands:

  `node --version` (Should report v8.x.x or better)

  `npm --version` (Should report v5.x.x or better)

  `npx --version` (Should report the same as **npm**)

- You should have node and npm on your path
- If either of these commands do not report back successfully, consult your setup instructions

CREATING A REACT APPLICATION

- Facebook (React's creators) provides a utility for creating React applications from scratch
- Called, conveniently, **create-react-app**
- It takes care of all of the work of setting up a server, watches, and deployments for you

CREATING A REACT APPLICATION, CONTINUED

- We can create a react application this way:
- Change directory to where you have your class files
- (Note: **NOT** the **ts-react-class** directory)

```
npx create-react-app first-react --use-npm
cd first-react
npm start
```

- Which will start a React application for you, opening it in your default browser

REGARDING NPX

- The npx command temporarily installs and uses the create-react-app library from the npm repository
- You could install it locally for a project with `npm install create-react-app`
- Or globally (if you were going to use it frequently) with `npm install --global create-react-app`
- Hit `Control-C` in the window where you started the application
- You can delete the directory, we will not be using it again

REACT AND TYPESCRIPT

- `create-react-app` assumes you are using ECMAScript to write your React application
- We want to create an application using TypeScript
- Try this:

```
npx create-react-app ts-first-react \
    --typescript --use-npm
cd ts-first-react
npm start
```

- You should see something that looks... mostly the same!
  - create-react-app has first-class support for TypeScript as of create-react-app 2.1.0

BACK TO STARTING QUICKLY

- Do the following, if you have not already:
- Open a command prompt or terminal window
  - On Windows, change directory to somewhere near the root of the drive you want to use
- Use Git to clone **https://github.com/speedingplanet/ts-react-class**
- Change directory to the newly created ts-react-class directory
- Run npm install

STARTING THE SERVER

- Windows: `start npm start`
  - Opens up the React app in a separate window, leaving the current command prompt available
- Macs (assuming bash shell): `npm start`
- The server will spin up, and your default browser should open soon at **http://localhost:3000** with a welcome page
- That's a React app!
- That was easy!
  - Although, to look at it, somewhat… underwhelming

BUILDING AN APPLICATION

- The "application" we currently have in ts-react-class does not really use much React
- In the following exercise, we will add the basic files and code needed to create a “Hello, world” React app
- Usually, exercise directions will be in the files for the project
- But in this case, we will follow the directions here in the slides

EXERCISE 1: OVERVIEW

- At its most basic, React allows you to design custom HTML tags
  - This is a *gross oversimplification*, but will do for the first exercise
- We need to do two things:
  - Write a custom tag
  - Hook it up to the browser's Document Object Model

EXERCISE 1: IDE SETUP

- Most IDEs will let you open a directory as a project
- Open the directory **ts-react-class** as a project
- Let's start by building a simple custom tag, called a component
- Set up the class files by entering the following command at a command prompt
  ```
  npx gulp start-exercise --src ex-01
  ```
- In the **src** folder in **ts-react-class**, edit the file **App.tsx**
  - Component names are capitalized
  - Files should contain one and only one component
  - The filename should be the same as the name of the component
- This will contain our new component `<App/>`

EXERCISE 1: DEFINING A COMPONENT

- The component can be written as a function
  - Later on, we will write components as TypeScript classes
  - You can mix and match, or define rules for when to use function-based components vs class-based components
  - Or you can just use one or the other
- Right now, the function takes no arguments
- Instead, we will have it return the HTML content of the component

EXERCISE 1: RETURNING HTML

- Our React components will return HTML in the form of JSX
  - It actually does not stand for anything!
  - It is a mix of JavaScript, TypeScript, XML, and HTML
  - We will look at the syntax in more detail later
- Two important points now:
  - Use `return (...)` to allow for statements which span lines
  - You must return one root element
    - Your root element can have as many sub-elements as you want
    - But there must be one root element

EXERCISE 1: THE REACT MODULE

- Our component is a function which is part of a TypeScript module
- The module must import the React module to work
- Specifically, it must import `React` from `react`
- Because of the interaction between TypeScript and React, we will use the import all construct:
  `import * as Foo from 'bar'`

EXERCISE 1: OUR NEW COMPONENT

```
import * as React from 'react';
export default function App() {
  return( <h1>Hello, world!</h1> )
}
```

EXERCISE 1: TYING THE COMPONENT TO THE DOM

- Next we need to tie the component to the DOM
- For the rest of the course, we will usually create and use components, not worrying about this step
- This is, then, a one-time step to connect the React application to the Document Object model of the browser
- The file that does this is usually called **index.tsx**
    - This is a convention, not a requirement
- Edit the **index.tsx** file in the **src** folder

EXERCISE 1: CREATING INDEX.TSX

- index.tsx is an ES2015 module (also a TypeScript module)
- index.tsx needs to import three modules:
  - ◦ as **React** from **react**
  - ◦ as **ReactDOM** from **react-dom**
  - **App** from **App**
- Importing React should be self-explanatory
- Importing ReactDOM provides the function that will add custom components to the DOM
- App is the custom component in question

EXERCISE 1: INDEX.TSX

```
import * as React from 'react';
import * as ReactDOM from 'react-dom';
import App from './App';

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

EXERCISE 1: CONNECTING TO THE DOM

- **ReactDOM** has the `render()` method which takes two arguments:
- The HTML or JSX to render (almost always JSX)
- A reference to an existing DOM node to render to
- If you look in **public/index.html**, you will find the following node: `<div id="root"></div>`
- This is the target node for `ReactDOM.render()`

EXERCISE 1: SUMMARY

- Create a component
- Create index.tsx
- Add a target element to **index.html**
  - This was already in place
- Tie the component to the target in index.tsx
- Examine code in a browser!

VIEWING SOLUTIONS

- If you want to see a solution, you can run
  `npx gulp show-solution --src ex-01`
- In a later exercise, we will use these commands to set up our environment
- For now, we will be building one exercise on another, so we will not use gulp (yet)

REACT AND TDD

- Throughout this course, we will place a special emphasis on testing
- We will not be using strict TDD
  - No test first, then write code
  - That is good for code, but not great for learning concepts!
- But we will write tests (failing and successful!) for the concepts we learn

TESTING REACT

- The Facebook team created the Jest framework to test React
- Jest incorporates a variety of testing features from various other frameworks
- And Jest is intended to be the official test framework for React
- We will use Jest to test all our React code
- Jest is included as part of any application created with create-react-app

EXERCISE 2: TESTING "HELLO, WORLD"

- Let's start with a basic test
- There are two parts of a Jest test:
- The test itself, created via the `test()` or `it()` method
- Expectations within the test, a combination of the `expect()` method and a matcher
    - You can have multiple expectations in one test
    - If you have more than 3-4 expectations, you should probably check to see if you should split the test into (several?) smaller tests

EXERCISE 2: CREATING A TEST

- Create a file in the **src** folder: **App.spec.tsx**
  - Jest automatically picks up files that contain the string **.test** or **.spec** in their names
  - Or any tests in the **tests** directory
  - This is configurable, though not in create-react-app
- Add the code on the next slide to **App.spec.tsx**

EXERCISE 2: A BASIC TEST

```
test( 'Adds 2 and 2 to equal 4',
  () => {  expect( 2 + 2 ).toBe( 4 );}
);
```

EXERCISE 2: RUNNING THE TEST

- Open a new command prompt or terminal window
- Change directory to the **ts-react-class** directory
- Run `npm test`
- You should see results indicating that the test was a success!
  - You may have to hit the 'a' key, to force a re-run of 'all' tests
- You can leave this window open, it will continue to watch your test(s) for changes, and re-run those tests when they change

EXERCISE 2: TESTING A COMPONENT

- That was a nice test but not, you know, realistic
- We would like to test our actual component
- We will write two more tests
- One to see if the component loaded correctly
- Another to see if the component has the right content

EXERCISE 2: DOES IT LOAD?

- How can we test a component to see if it loads?
- When we work with a component, we tie it to the DOM and then render it
- So we need a Document Object Model to work with, and a way to render a component to it
- Oh, and a component, too, but we have that in App

EXERCISE 2: RENDERING TO THE DOM

- Creating a DOM is actually simple: just use the DOM interface
- `document.createElement()` will return a reference to a DOM element
- How did we render to the “real” DOM in our application?
- We used `ReactDOM.render()`
- Which we can use again here in our test
- Finally, we will not have an expectation
- The fact that the component renders to the DOM without error is the test

EXERCISE 2: ADDING A TEST

- You can add the code from the next slide to `App.spec.tsx`
- Leave the test that we already created in the file
- Add the imports at the top
- And the new test either before or after the first test
  - Note that the new test uses the `it()` method instead of the `test()` method
  - They are interchangable (`it()` is an alias to `test()`)
- The `npm test` window should pick up the changes and re-run the test for you automatically

EXERCISE 2: TEST RENDERING

```
import * as React from 'react';
import * as ReactDOM from 'react-dom';
import App from './App';

// Leave our first test here

it('renders without crashing',
  () => {
    const div = document.createElement('div');
    ReactDOM.render(<App />, div);
  }
);
```

EXERCISE 2: TEST CONTENT

- Great, we have a basic test, and we can test that the App component actually renders to a DOM element
- But do we know if it renders useful information?
  - Given, our App component is hard-coded at the moment
- How can we test for the content of the element?
- We could use the DOM to retrieve the content of the created `<div>`
  - But this could get tricky, depending on rendering time, methods used to access the text of the DOM element, etc

EXERCISE 2: USING ENZYME

- Instead, we will add a test helping utility called Enzyme
- Enzyme, created and maintained by AirBnB, makes it easier to work with React components
- We will use Enzyme to shallowly render the App component
  - And then test its content

EXERCISE 2: SHALLOW RENDERING

- Enzyme exports a method, `shallow()`, which takes a component object as an argument
- It returns a `wrapper` around the component that lets us test it as if it were rendered
- In particular, we are interested in the `wrapper.text()` method, which pulls out a String representation of all of the text nodes in the rendered component

ENZYME CONFIGURATION

- Enzyme has become very popular over time, and is now used with several different frameworks
- With Enzyme version 3 and later, you will need to load Enzyme, and an adapter for your test target
  - In this case, an adapter for React v16
- Add the code below to the top of your **App.test.tsx** file

```
import {shallow} from 'enzyme';
import Enzyme from 'enzyme';
import Adapter from 'enzyme-adapter-react-16';

Enzyme.configure( { adapter: new Adapter() } );
```

EXERCISE 2: USING ENZYME

- Now your code should look like this:

```
import * as React from 'react';
import * as ReactDOM from 'react-dom';
import App from './App';
import {shallow} from 'enzyme';
import Enzyme from 'enzyme';
import Adapter from 'enzyme-adapter-react-16';

Enzyme.configure( { adapter: new Adapter() } );

// Tests 1 and 2 here

test('Contains "Hello"',
  () => {
    const wrapper = shallow(<App/>);
    expect(wrapper.text()).toMatch(/Hello/);
  }
);
```

EXERCISE 2: ANALYSIS

- We added an `import` line to bring the `shallow()` method in from enzyme
- We also added a test which created a wrapper around a shallowly rendered component
- The test looked at the text of the component to see if it included what we expected

EXERCISE 2: SUMMARY

- Testing React is very similar to writing React
- Either use React directly, or add in helper frameworks like Enzyme to test components
- Render the component you want to test, and then check its properties and state to see if it is acting the way you expect it to
- We will expand our testing capabilities as the course continues

REACT CHAPTER 2

REACT ARCHITECTURE

CHAPTER PREVIEW

- About React
- Why React?
- Passing data
- Our environment
  - Server
  - Transpiler
  - Watching files
- Where do we go from here?

ABOUT REACT

- React was created by Facebook and carries the open source BSD license
  - It is used extensively by Facebook, Instagram, Netflix and others
- React is “[a] declarative, efficient, and flexible JavaScript library for building user interfaces.”
  - According to the GitHub repository for React
- React is a View library, it does not provide nor is it opinionated about models or controllers
- It is wholly and solely concerned with rendering the UI

ABOUT REACT, CONTINUED

- Again, from the GitHub repo, React is:
- **Declarative**: React makes it painless to create interactive UIs. Design simple views for each state in your application, and React will efficiently update and render just the right components when your data changes. Declarative views make your code more predictable, simpler to understand, and easier to debug.
- **Component-Based**: Build encapsulated components that manage their own state, then compose them to make complex UIs. Since component logic is written in JavaScript instead of templates, you can easily pass rich data through your app and keep state out of the DOM.

ABOUT REACT, CONTINUED

- **Learn Once, Write Anywhere**: We don't make assumptions about the rest of your technology stack, so you can develop new features in React without rewriting existing code.

REACT INFO SHEET

- Home page: https://reactjs.org/
- Docs: https://reactjs.org/docs
- Github repo: https://github.com/facebook/react
- Version: 16.x (16.7.0 as of January of 2019)
- React devtools: https://github.com/facebook/react-devtools

WHY REACT

- What are the reasons for using React? Put another way, what does React do best?
- React is fast: perhaps one of the fastest UI renderers extant
- React is simple: React itself is not very complicated and does not aspire to be anything more than a view library
- React is extensible: Nonetheless, it is easy to extend React with other tools (like Redux) or to use it in a variety of different circumstances, situations, or environments
- React is component-based: React is aligned with the future of web application architecture in general (component-based vs MVC-style)

COMPONENT-BASED APPLICATIONS

- React promotes development of small, extensible, loosely coupled components as the building blocks of applications
- You do not truly build an “application” per se in React
    - You build the presentation of the application
- Use and re-use components to render various aspects of your application
- Do not worry about rendering or updating, as you can let React manage that part

COMPONENTS VS MVC

- There is not a “right” way to do client-side applications
- The first generation of JavaScript application frameworks (think AngularJS and Backbone) leaned heavily on the MVC pattern for inspiration
- In the context of the web, this has some shortcomings
  - Web application need to update their UI often
  - UI updates are costly and should be optimized
  - Clients have no persistent data connection
  - Clients must maintain their own state, but also reconcile that state with server state

COMPONENTS VS MVC, CONTINUED

- The second generation of client-side applications (React, Angular, Vue, others) relies on a component-based approach
- Rather than strict definitions of Model, View, and Controller, components represent parts of the View
- Components do not care about where they get the Model they display
- Components do not know about the concept of a Controller, either
- As application designers, this gives us flexibility in figuring out the M and C roles, while leaving React to manage the View

FUNCTIONAL-STYLE COMPONENTS

- In the first chapter, we created a component from a function
- Components can be created from functions
- They are limited in their capabilities by the limitations of a function
  - No methods
  - No constructor
  - Limited arguments
  - No lifecycle overrides
- But they are lightweight, easy to create, and easy to manage
- In the future, React may optimize functional components

CLASS-STYLE COMPONENTS

- Components can be created from TypeScript classes via inheritance
- Have the class extend `React.Component`
- Provide a `render()` method
  - In functional components, the function itself is the `render()` method
- Class style components have all of the benefits of classes
  - Extensible
  - Full lifecycle of events
  - Break out functionality into methods
  - Constructor
  - And more

CLASS VS FUNCTIONAL COMPONENTS

- So which should we use?
  - Use both, but at different times!
- The answer is, what does your component need?
- Some argue for using functional components when the component is stateless
  - And class-based components when the component is stateful
  - We will see a variant of this argument later in the course
- If you need features only accessible in class components, use those
  - Otherwise, prefer functional components
- In this course, we will use functional components unless we need the features of a class-based component

DEMO: CLASS-BASED COMPONENTS

- Here is the component from Exercise 1, re-rendered as a class-based component
- This has no implications for testing, so it should continue to run fine
  - You can find this in **exercises/ex-02/solution/src/AppClass.js**

```
import * as React from 'react';

export default class AppClass extends React.Component {
  render() {
    return (<h1>Hello, world!</h1> );
  }
}
```

PASSING DATA

- So far, we have built and tested a pretty basic component
- Let's expand the capabilities of the component by passing data to it
- Passing data to a component in React is surprisingly easy
- Inbound data is sent via attributes
  - `<MyComponent someInput="someValue"/>`
- In the component, you can access the data as a field called props
- Most components are written as `(props) => { … }`

USING DATA

- You can use passed data in your JSX as `{props.propertyName}`
- The props collection contains any and all information passed into this component as an attribute
- The props collection is **immutable**
  - Primitives cannot be changed at all
  - Object references can have their state changed, but not the reference itself
- Think of a component as a function: props are the input, HTML is the output

COMPONENT DATA

- What about component data?
  - Data that belongs to the component itself, but is not passed in
- Any data in the component can be accessed by its variable name
  - Declare a variable
  - `let x = 10;`
  - Use it in the JSX of the component
  - `<span>x is equal to {x}</span>`
- Later, we will use a special variable for this component data

CLASS-BASED DATA

- Accessing data in a class-based component is done differently
- If the class does not have a constructor, properties are available as `this.props` throughout the class
- If the class does have a constructor, it should have one argument, `props`, which is also passed to the `super()` constructor

TYPING PROPS

- The `props` property needs a type in TypeScript
- This is typically done by adding an interface which defines the properties of the props
- Here's a functional example

```
import * as React from 'react';

interface LocalProps {
  firstName: string;
  lastName: string;
}

const PersonDisplay = (props: LocalProps) => { ... }
```

TYPING PROPS, CONTINUED

- Things are a little different in class-based components
- Class-based components take advantage of generics, which allow you to vary types for both props and (as we will see later) state
- Here's an example of the same code as the last slide, using a class-based component

```
import * as React from 'react';

interface LocalProps {
  firstName: string;
  lastName: string;
}

export default class PersonDisplay<LocalProps> {
  constructor(props: LocalProps) {
    super();
  }
}
```

TYPING PROPS, CONTINUED

- The signature of a `React.Component` allows you to pass in the `props` type
  - Actually, it lets you pass in up to three types, for **props**, **state**, and **context**
  - We will see state later on in the course
- Despite passing in the props type on the class definition line, we still have to give `props` a type in the construction signature
  - A bit repetitive, but this is due to the fact that the TypeScript compiler will not know about the generic prop type (the one in <>) until after it has processed the constructor

EXERCISE 3: PASSING DATA

- We will update our application with custom headers and footers
- The `CustomHeader` and `CustomFooter` components are mostly concerned with CSS styling
- Though they take arguments to determine their content
- Run `npx gulp start-exercise --src ex-03`

OUR ENVIRONMENT

- We are using an application created by `create-react-app`, customized for TypeScript
- create-react-app makes it easy to prototype a React application without having to worry about many decisions related to setting up the environment
- It provides a web server, file watchers, TypeScript implementation, test framework, and code linting

IMPORTANT PARTS OF THE APPLICATION

- In this class, we do not worry to much about the application environment
- But in the world outside of class, you should think about these parts of your environment:
- Web server (Node JS + Express? Java + Tomcat? Windows + IIS?)
- Test framework (probably Jest, possibly with helper frameworks)
- ES2015 implementation (Most likely Babel)
- Task runner (Gulp, Grunt, or Maven, or Gradle, etc)
- Code packaging (Webpack, Browserify, possibly others)

WHERE DO WE GO FROM HERE?

- What is the plan?
- The conceit for the class is that we are building a consumer banking application
- Similar to what you would see when you log in to your bank to look at your checking account
- Most of the application will concentrate on the payees section

REST SERVER

- We also have a RESTful server available to us
- You can start it from **ts-react-class** by executing `npm run rest`
- The server has several RESTful endpoints
- tx (transactions)
- payees
- accounts
- people
- categories
- staticData
- It is implemented by **json-server**, an npm package

SUMMARY AND CONCLUSION

- React was created by and is maintained by Facebook
- With many outside contributions
- React is a client-side view library
- React implements component-based architecture, rather than strict MVC
- React itself is agnostic about other tools in the web development environment
- Possibly excepting Jest as the test framework
- As designers, we should be aware of the choices of other tools we will need in the environment, external to React

REACT CHAPTER 3

BASIC REACT COMPONENTS

CHAPTER PREVIEW

- Introduction to JSX
- Classes and style
- Snapshot testing
- Using child components and content
- Conditionally displaying data

INTRODUCTION TO JSX

- Whether using functions or classes to create components, we use JSX to generate HTML
- JSX is a mix of JavaScript, HTML, and XML
  - In our case, TypeScript, HTML, and XML
- It follows XML's rules about being well-formed
- One, single root element
- Tags must be `<balanced></balanced>` or `<standalone />`
- Tags must match their nesting `<i><b></b></i>` not `<i><b></i></b>`
- Tags are case-sensitive

JSX AND REACT

- A few more rules for working with React and JSX
- The first letter of the name of the component is capitalized
- Especially since it is often a class as well
- Attributes should be bound with quotation marks
- Single or double does not matter, but be consistent
- But attributes and other information can be substituted with { }
- `<MyComponent someAttr={someValue} />`
- Whenever you use JSX, React must be in scope (imported into the current TypeScript module)

JSX AND TYPESCRIPT

- We can add arbitrary TypeScript to our JSX by enclosing it in curly braces
- This is true for attribute values, code between elements, anywhere in our JSX we like
- But we still have to follow the XML rules and keep our code well-formed
- In practice, this means that our TypeScript is somewhere under the root element of our JSX
- The TypeScript of JSX is not a sub-set or a DSL, it is real TypeScript (well, JavaScript, once compiled)
- Write to whatever version of ECMAScript your deployment will support

JSX AND CONTEXT

- In JavaScript, context is very important for resolving variables
- JSX is no different
- The context for any variables in your JavaScript expressions in JSX is the context of the method (usually render) creating the JSX
- Variables do not cross scopes in JSX
- If your component is the child of another component, your JSX does not have access to the parent component
- Or any siblings, for that matter
- We will see soon how to pass information among components

JSX SYNTAX

- A few things are different in JSX syntax
- Comments out a block with `{/* <Block /> */}`
- Use a string literal in quotes `{ 'string literal' }`
- Values like `false`, `undefined`, `null`, and `true` are valid, but do not render out anything in the page
- Do not forget to return a single root element
- Do not forget that the return statement can wrap lines by using parentheses

JSX AND CHILD COMPONENTS

- JSX also allows you to render child components
- Given this structure:

```
<Parent>
<Child>
  Some text
</Child>
</Parent>
```

- The component `<Child>` will be in the `props.children` property of `<Parent/>`
- Likewise, the text Some text will be the props.children of `<Child/>`
- In a class-based component, refer to `this.props.children`

CLASSES AND STYLE

- CSS and React have some particular rules about interactions
- First, when specifying a CSS class or classes in JSX, use the `className` attribute instead of `class`
- `class` is a reserved word in TypeScript, preventing its use in JSX
- Otherwise a `className` is a property like any other, set it in TypeScript and assign it a literal or a variable
- `className=“foo bar baz”`
- `className={myClassList}`

STYLE ATTRIBUTES

- Style attributes must be JSX assignments
- `style=“color:blue”` is not valid in an element
- Use a JavaScript object literal to specify an in-line style
- `const myStyle = { color: 'blue' }…<li style={myStyle}>…</li>`
- Or specify. an object literal in-line
- `<li style={ { color: 'blue' } }>…</li>`

EXERCISE 4: USING JSX

- In this exercise, we will conditionally display data, as well as use `props.children` to render content
- Objectives:
- Use inline TypeScript in our JSX to conditionally display information
- Use the `props.children` feature to render out child elements of the parent component without knowing them in advance
- Style elements according to information in props
- Use `gulp` to load the class files for the exercise
- `npx gulp start-exercise --src ex-04`
- If you have problems with the exercise, ask your instructor for help

RESULTS TESTING

- Our last two exercises have focused on rendering out content to the DOM
- We would like to write some tests which can focus on the results of rendering a component
- Rather than test a small part of the component (which is still a useful test, of course), we would like to look at the component rendering tree as a whole
- Given a component, particularly one with child components, can we ensure that it is still rendering the way we expect after a change?

SNAPSHOT TESTING

- Testing the entire results of a rendered component is an ideal use case for snapshot testing
- The name comes from the practice of taking a snapshot of rendered content, and then making changes
- After the changes, a second snapshot is taken and then compared to the original snapshot
- The test passes if the two are the same
- Snapshot testing ensures that, despite changes, updates have not broken your component's UI functionality

CREATING SNAPSHOTS WITH JEST

- Snapshots are easy to create
- Import * as renderer from react-test-renderer
- Use `renderer` to create a snapshot and serialize it to JSON
- `const snap = renderer.create( <MyComponent/> ).toJSON()`
- Use the matcher `toMatchSnapshot()` in your expectation
- `expect(snap).toMatchSnapshot()`
- Snapshots are put into a folder called **__snapshots__** in the same folder as the test

SNAPSHOTS

- When you first run a test which uses renderer, it creates a snapshot file
  - It is in human-readable JSON
- Subsequent test runs are compared to this file
  - If the snapshots match, the test is successful
- If you need to regenerate snapshots, invoke jest like so
  - jest --updateSnapshot
  - This would work in an ejected app
  - While under `npm test`, you can enter `u` to update snapshots
- Commit your snapshots to your VCS along with other test code

EXERCISE 5: SNAPSHOT TESTING

- We will use snapshots to test the code from our last exercise
- Objectives:
  - Generate snapshots in our tests
  - Ensure that our tests pass when making minor updates to the code
- Use gulp to load the class files for the exercise
  - `npx gulp start-exercise --src ex-05`
- If you have problems with the exercise, ask your instructor for help

CONDITIONALLY DISPLAYING ELEMENTS

- JSX and JavaScript can be used to conditionally display entire elements
- Rather than just changing the content of elements
- Remember that elements are values in JSX, just like numbers, strings and so on
- Assign an element to a variable conditionally
- Then display the variable in the `return` statement of your component's rendering function
- Or, use a logical operator like `&&` or `?:` inline

EXERCISE 6: CONDITIONALLY DISPLAYING DATA

- This is the last of our exercises to experiment with our "Hello, world" home page
- In this exercise, we add conditional component display to our bag of tricks
- Objectives:
  - Depending on whether the user is logged in, display a ”Log In” button or a “Logged in as …” banner
- Use gulp to load the class files for the exercise
  - `npx gulp start-exercise --src ex-06`
- If you have problems with the exercise, ask your instructor for help

CONCLUSION

- So far, we have been working with simple React components, trying things out and experimenting
- We have used JSX to render content
- Sometimes conditionally!
- And also taken snapshots of that content for testing purposes
- In the next chapter, we will move over to a set of “real world” exercises

REACT CHAPTER 4

EVENT HANDLING AND STATE

CHAPTER PREVIEW

- Our real-world example: Payees
- Basic event handling
- Props and state
- Extracting components
- Inter-component communication
- Testing component state
- Using spies
- PropTypes

OUR REAL-WORLD EXAMPLE

- In the last chapter, we focused on a simple "Hello, world" example
- Now we are going to move to working with a more realistic topic: a consumer banking site
- We are in charge of implementing the Payees portion of the site
- Payees are associated with Transactions (tx) and Categories

THE PAYEE INTERFACE

- Found in **src/data/Payee.ts**
- HasId.ts is in the same folder

```
export default interface Payee extends HasId {
  id: string;
  payeeName: string;
  address?: string;
  city?: string;
  state?: string;
  zip?: string;
  categoryId: string;
  category?: Category;
  image?: string | null;
  motto?: string | null;
  active: boolean;
}
```

NOTES ABOUT PAYEES

- Payees are associated with categories through a category id
- Categories have a **categoryName** and a **categoryType**
- Payees do not know about their Transactions
  - But Transactions may know about Payees
- Payees may have an image, but there is, at the moment, no image folder where that image is held

TYPES: REACT FUNCTIONAL COMPONENTS

- Functional components can be just plain old functions
- But the **@types/react** package provides a utility type for simple functional components
- `React.SFC<P>`
  - **SFC** stands for Stateless Functional Component
  - The `<P>` is a generic placeholder for the Props of this component
- You can see how to use this type on the next slide

DEMO: REACT.SFC

```
import * as React from 'react';

interface GreeterProps {
  message: string;
}

const Greeter: React.SFC<GreeterProps> = (props: GreeterProps) => {
  // implementation ...
}
```

TESTING: ACCESSING PROPS

- We have talked about using **react-test-renderer** to create snapshots and **enzyme** to do more complex, feature-rich testing
- When using functional components, enzyme has some shortcomings
- Enzyme cannot provide access to the props of a stateless functional component when using `shallow()` to render that component
  - Using full rendering, via `mount()` will get you access to the component's props
- If we want to test the properties of a stateless functional component, we must use react-test-renderer

TESTING: ACCESSING PROPS, CONTINUED

- Using `react-test-renderer.create()` generates a `testRenderer` object
- `testRenderer` has several properties, detailed here https://reactjs.org/docs/test-renderer.html#testrenderer-instance
- The `root` property accesses the `testInstance`, the instance of the component under test
- The `testInstance` has access to a `props` property which contains a map of the properties and their values for this component instance

TESTING: ACCESSING PROPS, CONTINUED

- To test the properties of a simple functional component use code similar to this:

```
import * as renderer from 'react-test-renderer';

test('Getting at props', () => {
  const testRenderer = renderer.create(<SomeSFC key="value"/>);
  const testInstance = testRenderer.root;
  expect(testInstance.props.key).toBe('value');
});
```

EXERCISE 7: RENDERING A PAYEE

- Our first job is to create a `PayeeDetail` component
- This component will render the relevant portions of a Payee
- We will be using Bootstrap, a popular CSS library, to render the Payee as a panel
- Bootstrap's docs can be found at **getbootstrap.com**
- Specifically, we will use a Bootstrap Panel (from version 3) to render the Payee
- http://getbootstrap.com/components/#panels

EXERCISE 7: GETTING DATA FOR THE PAYEE

- Where can we get data from?
- Later on in the course, we will use the Fetch API to retrieve data from a remote REST server
- For now, we will include the data in our project directly
- Our `PayeeDetail` component can import **data/class-data.js**
- The class data module exports an object, `payeesDAO`, which provides access to Payees
- Ask for a Payee by id with 0

EXERCISE 7: RENDERING A PAYEE

- Now that we know how to access data, here are your tasks:
  - Build a `PayeeDetail` component
  - Get a Payee (payee #23 for instance)
  - Pass it into `PayeeDetail`
  - Render out its content
- Use gulp to load the class files for the exercise
  - `npx gulp start-exercise --src ex-07`
- If you have problems with the exercise, ask your instructor for help

TYPES: CLASS-BASED COMPONENTS

- Stateless functional components take advantage of a type that uses generics
- Class-based components do the same
- Extending React.Component can use generics as well
- `React.Component<Props, State>`
  - There's an optional third generic type which we will skip over at the moment
- The first generic type is the props type for the component
- The second is the state type
  - We will see more about state soon enough!

WHAT'S NEXT?

- The next feature on our to-do list for Payees is to allow users to page through a set of Payees
- What will we need to implement a simple paging toolbar?
- Next and previous buttons
- A set of data
- A way for the buttons to ask for the next or previous button in the set
- Let's dive in to React's version of event handling

EVENT HANDLING AND STATE

- So far our components have been static
- Charged with simply displaying the information passed to them
- In the real world, our components will need to react to user input, and manage information according to said input
- Reacting to user input is the province of event handling
- Manging that changing data comes under the heading of managing state
- We will look at event handling first

EVENT HANDLING

- To implement event handling with React, add code in two places
- First, in the component, add an event handling function
    - Class-based and functional components handle this differently
    - Look at the following slides for details
- Tie an event to the handler in the JSX
- Given an event handler, `clickHandler`, defined in your component code
- Tie it to the component by adding the following attribute`onClick={clickHandler}`
- Note that to bind an event handler, we use { } not quotation marks

FUNCTIONAL COMPONENT EVENT HANDLING

- In functional components, define the event handler as a sub-function of the component function
- In the JSX for the event handler, bind the function directly
- `<button onClick={clickHandler} />`
- It is somewhat of a convention that the event handler for a DOM event foo is fooHandler

CLASS-BASED EVENT HANDLING

- Class-based event handling is somewhat more complex
- First, the handling function should be a class-level method (i.e., a sibling of constructor)
- Beyond that, there are complications to how to bind the event to the handler
- "Complications" from one perspective, "flexibility" from other perspectives
- See the next slide for details

BINDING EVENT HANDLERS

- The typical form to use a class-based event handler is `<button onClick={this.handleClick} />`
- Unfortunately, the `this` in `this.handleClick` will not be bound correctly without some assistance
  - It will be bound to some other context, since this is relative to the **execution** context, not the context at binding time

BINDING EVENT HANDLERS, CONTINUED

- In the constructor, add this line:

```
this.handleClick = this.handleClick.bind( this );
```

- Yes, that looks weird
- This rewrites the `handleClick()` method of the component to always be bound to the current instance
- Yes, that should happen automatically (maybe in a later version of React)

OTHER WAYS TO BIND EVENT HANDLERS

- `<button onClick={ (event) => { … } } />`
  - No need to bind this handler in the constructor
  - But it creates a new handler each time the component is rendered
  - This may cause extra re-rendering
  - Never use this in a loop, for instance

THE PREFERRED WAY TO BIND EVENT HANDLERS

- `handleClick = () => { … }`
  - Use this at the class method level
  - This is experimental syntax and may not be finalized in later versions of JS
  - Create React App enables this by default
  - Only downside is that the syntax may go away in the future
  - Otherwise, it's the best, clearest, and easiest-to-use choice
  - No weird double-binding in the constructor
  - No bad side effects if used in a loop

TYPES: EVENT HANDLING

- React has several different event handling types
- The core event is `React.SyntheticEvent` which provides the typical properties
  - `currentTarget`, `preventDefault()`, `stopPropagation()` and so on
  - Note that `target` is **NOT** on `SyntheticEvent`
- `SyntheticEvent` uses generics to accept any type of HTML `Element` as the source of the event
  - There are many, many implementers of `Element`
  - Your IDE's intellisense is usually a help here

SYNTHETICEVENT SUB-INTERFACES

- You would not usually use `SyntheticEvent` directly
- Instead, use one of its many inheritors
  - `FocusEvent`
  - `ChangeEvent`
  - `MouseEvent`
  - and so on
- Event handling types help keep your event listeners type-safe

```
const buttonClickHandler = ( e: React.MouseEvent<HTMLButtonElement> ) => {
  // implementation....
}
```

EXERCISE 8: EVENT HANDLING

- The next three exercises have a common goal: Build a Payee browser where users can move through a set of Payees one at a time by clicking on Next and Previous buttons
- First, we will set up the buttons and attach event handlers
- Later on we will hook these up to data in the component
- Finally, we will use inter-component communication to choose which Payee to display

EXERCISE 8: EVENT HANDLING

- Objectives:
  - Add two buttons to the `PayeeDetail` component, **Next** and **Previous**
  - Attach event handlers to the respective buttons
  - For now, when the event handler triggers, you can simply log the event to the console
- Use gulp to load the class files for the exercise
  - `npx gulp start-exercise --src ex-08`
- If you have problems with the exercise, ask your instructor for help

PROPS AND PURE COMPONENTS

- The `props` collection contains all the data passed to this component
- It is immutable, and you should not attempt to change it
- Components which use only `props` are called **pure components**
- And, since they usually use the functional style, they are often known as pure functional components
- Pure components (like pure functions)
    - Do not modify their input
    - Have no side effects
    - Given the same input, always return the same output

WHY ARE PURE COMPONENTS USEFUL?

- Why do we care about pure components?
- Pure components are easier to test
  - Standard inputs, no mocking
  - No side effects to worry about
- Pure components are often easier to reason about
  - The inputs are known
  - The expected output is known
  - The work of the component is tightly focused
- Pure components may be optimized by JavaScript and React
  - Depending on the JS engine and future versions of React

COMPONENT-LEVEL STATE

- But what if my component needs to manage internal information?
  - Which is commonly called **state**
- React has a different data collection for malleable information within the component: state
- The state variable is not available to functional components, only to class-based components
  - In the constructor and throughout as `this.state`

WORKING WITH STATE

- The state variable should be initialized in your component's constructor
  - `this.state = { … }`
- Initialize state as an object literal
  - It can have nested objects as needed
- Access state as `this.state.variableName`
- Anywhere you modify state, use the `this.setState` function
- Pass it the modified state as an object literal
  - `this.setState( {name: 'John'} )`
- Do not modify state with direct assignments!

STATE AND SETSTATE

- Modifying state with `setState()` tells React that it may need to update this component
- `setState()'s` job is to determine whether there was a change which affects the DOM
- We will go into the lifecycle details later, but the short version is that `setState()` may trigger a re-run of the `render()` method on the class
- Because `setState()` may trigger DOM updates, it may act asynchronously, and it may choose to batch updates, if they are coming rapidly
- Never modify state without using `setState()`!

MODIFYING STATE

- Always use `this.setState()` to modify state
- Do not assign to `this.state` directly
- DO NOT DO THIS:
- `this.state.name = 'John'`
- Modifying `this.state` directly prevents React from testing the data to see if it needs to update the DOM
- Your changes will not be reflected in the UI
- Use `this.setState()` to change state!

STATE VS PROPS

- So what should be **state**, and what should be **props**?
- Guidelines for state:
  - Belongs to this component
  - Original, cannot be calculated from other available values (other state, props, etc.)
  - Changes over time
- Props can become state, it's unusual but not rare
- Think of doing so as making a local copy of data to modify

EXERCISE 9: WORKING WITH STATE

- Part 2 of our Payee pager exercises
- Objectives:
  - Load a list of Payees
  - Use state to track the currently displayed Payee
  - Use event handlers to browse through the set of Payees
- Use gulp to load the class files for the exercise
  - `npx gulp start-exercise --src ex-09`
- If you have problems with the exercise, ask your instructor for help

MULTIPLE COMPONENTS

- So far, we have kept our application to two components: `App` and `PayeeDetail`
- But our `App` may eventually have other topics
  - Transactions, Categories, etc
- And we will definitely have other components under Payees
- React is about building reusable, loosely coupled components
- So we should refactor our application to take that into account

PRESENTATIONAL AND CONTAINER COMPONENTS

- Adapted from https://medium.com/@dan_abramov/smart-and-dumb-components-7ca2f9a7c7d0
- When working with components, it is useful to divide them into two types: purely presentational, and containers
- Presentational components do not usually include data access or business logic
  - Their job is to display the content handed to them
  - Usually also pure functional components
- Container components may include some presentational aspects
  - But their job is more likely to manage child components, be they presentational or containers themselves

EXTRACTING COMPONENTS

- So, we are going to split out some responsibilities
- We will create a `Payees` component, which will be the container component for the Payees portion of our application
- The `Payees` component will contain the `PayeeDetail` component
- And `PayeeDetail` component will be a presentational component
- `Payees` will be responsible for holding on to the array of Payee objects

PARENT TO CHILD COMMUNICATION

- We have seen values passed into elements before
  - `<HelloWorld name={firstName}/>`
- We can do the same with our Payees to PayeeDetail communication
  - A parent component can easily pass information to a child component
  - In this case, `PayeeDetail` will be passed a selected Payee
  - If we want `PayeeDetail` to re-render if and when the selected Payee changes, what should we do?
  - Have the selected Payee be part of `Payees`' state!
  - When we change the selected Payee in `Payees` via `setState()`, we will trigger a re-render in `PayeeDetail`

CHILD TO PARENT COMMUNICATION?

- What if a child component wants to send a message to a parent component?
- Think of an edit form, purely presentational, which wants to inform a container parent about changes to its content
- For our case, think of the **Next** and **Previous** buttons
  - They do not belong to a `PayeeDetail` component
  - They do not belong to the `Payees` component
  - We will probably create a component for them
  - But how can we customize the behavior of the buttons?
  - How can we make our `BrowserButtons` component flexible?

INTER-COMPONENT COMMUNICATION

- The answer is custom events
- Create a custom event on a child component
- When a parent component uses a child component, the parent can pass in an event handler for the custom event
- When the child fires the custom event, the parent's event handler fires
    - Often, the child fires the custom event in response to a DOM event
    - Click on a button in the child, fire a custom event in the `onClick` handler
- The child component is sending a message to the parent anytime a custom event occurs

EXERCISE 10: EXTRACTING COMPONENTS

- Objectives:
  - Extract some code into the `PayeesContainer`, `PayeeDetail`, and `BrowserButtons` components
  - Add custom events to `BrowserButtons`
  - Refactor code to handle `BrowserButtons`' custom events
- Use `gulp` to load the class files for the exercise
  - `npx gulp start-exercise --src ex-10`
- If you have problems with the exercise, ask your instructor for help

TESTING EVENT HANDLING

- We have gone through several exercises to build our small-scale Payee browser
- And we do not have any testing to go with it!
- We should remedy this
- We could use basic testing and snapshot testing for some of our testing needs
- We will need to simulate user behavior, specifically clicking on a button, to have thorough testing

TESTING EVENTS

- The Enzyme test library allows you to simulate events on a component
- After wrapping a component, use the simulate method, passing in the name the the event to fire and any other relevant arguments
- `wrapped.simulate('click')`
- Enzyme actually finds the corresponding property (`onClick` in this case) and fires it
- This is not a proper DOM event and does not propagate
- Which is why it is called 'simulate"

CHECKING STATE AND PROPS

- Enzyme-wrapped class-based components have access to their respective state and props properties
- `wrapper.state().key`
  - Also `wrapper.state(key)`
- `wrapper.props()`
  - Returns a collection of the current props for this component
- `wrapper.prop(key)`
- Use these to test that the component is in an expected state after simulating an event
- Remember that you have to use **react-test-renderer** if you want to acces the props of a stateless functional component

SHALLOW AND FULL RENDERING

- When we first talked about enzyme, we used it to render a component shallowly
- At the time, we did not have parent-child component relationships, so shallow rendering made sense
- If we are to test our nested elements, we may have to fully render them with the mount method
- Enzyme makes `mount()` available to fully render an element to the Document Object model

FULL RENDERING REQUIREMENTS

- Fully rendering a component requires an actual working DOM
- You could run your tests in a browser, or
- You could use a simulated DOM like **jsdom**
- "jsdom is a pure-JavaScript implementation of many web standards, notably the WHATWG DOM and HTML Standards, for use with Node.js. In general, the goal of the project is to emulate enough of a subset of a web browser to be useful for testing and scraping real-world web applications."
    - From the jsdom GitHub page

SPYING ON CHANGES

- Firing an event may have concrete changes
- But it may also have subtle changes
- We want to be able to see what code ran when an event is fired
- The Jest library enables spying on methods
- We can use Jest to spy on an event handler to ensure that it was called
- Rather than looking for the effects of an event handler having been called

WORKING WITH JEST

- Jest allows you to create spy methods
- `const spy = jest.spyOn(targetObject, nameOfMethod)`
- The spy object then tracks calls to the named method on `targetObject`
- Check to see if the spy has been invoked: `expect(spy.mock.calls.length).toBe(1)`
- Spies will pass calls through to the method they are spying on

SPYON GENERICS

- `jest.spyOn(targetObject, nameOfMethod)` may not be able to determine that `targetObject` has the property specified in `nameOfMethod`
- Pass `spyOn()` two generic arguments
  - First, the component represented by `targetObject`
  - Second, keyof that component, so `spyOn()` gets a list of appropriate properties on which to spy

```
jest.spyOn<SomeComponent, keyof SomeComponent>(wrapper.instance(), 'someMethod');
```

PROBLEMS WITH SPIES

- Let's say we have a container component, which wraps around a few presentational components
- The container component has a property, fooHandler, which is passed into one of its children as an event handler
- We want to ensure that this fooHandler is being invoked by the child component
- We might write some code similar to the next slide

PROBLEMS WITH SPIES, CONTINUED

```
const container: any = wrapper.instance();
const spy = jest.spyOn<SomeContainer, keyof SomeContainer>
              (container, 'fooHandler');

// This passes
expect( spy.mock.calls.length ).toBe( 0 );
wrapper.update();

// Fires an event handler, so fooHandler should have been invoked!
wrapper.find( 'button' ).simulate( 'click' );

// Fails with "expected (1) but received (0)
expect( spy.mock.calls.length ).toBe( 1 );
```

SPIES AND COMPONENT LIFECYCLE

- What happened?
- If `fooHandler` was a property on `SomeContainer`, it had **already been assigned** to the child component, before we spied on it
  - Put another way, by the time we set up the spy, `SomeContainer.render()` has already been called
- We were, in actuality, spying on a different method!
- Thankfully, this is easy to solve

SPYING THE RIGHT WAY

```
const container: SomeContainer = wrapper.instance();
const spy = jest.spyOn<SomeContainer, keyof SomeContainer>
              (container, 'fooHandler');

// Calling mount() on the wrapper calls the entire lifecycle of the
// component again, including render()
wrapper.mount();

// Still passes
expect( spy.mock.calls.length ).toBe( 0 );
wrapper.update();

wrapper.find( 'button' ).simulate( 'click' );

// And now this passes as well
expect( spy.mock.calls.length ).toBe( 1 );
```

MOCKING FUNCTIONS

- Sometimes you want a function that will serve as a placeholder, an indication that a function has been provided, and should be tracked
- Other times, a component might require an event handler as a prop
  - You are not going to write a new event handler for the test
  - Instead, you want to provide a no-op function which you can check up on
- This is where Jest mocks come in
- `const mockObject = jest.fn()`

MOCKING FUNCTIONS, CONTINUED

- The `jest.fn()` method provides a function which is much like a spy, but is not specifically attached to any object
- Handy for event handlers, processors, etc.
- Has the same Mock API as spies, so you can track the behavior of the mock function and test it in a similar manner

TYPING MOCKS

- `jest.fn()` has some features that make it easier to work with in TypeScript
- It can take an argument of an implementation, which helps if you are mocking an event handler that requires a certain function signature
- `jest.fn((name: string) => 1)`

EXERCISE 11: TESTING COMPONENTS

- Objectives
  - Test our `PayeesComponent`, `PayeeDetail`, and `BrowserButtons` components
  - Use the `state()` and `props()` methods to test that the components have the proper state and props
  - Use spies to ensure that event handling functions were called correctly
- Use gulp to load the class files for the exercise
  - `npx gulp start-exercise --src ex-11`
- If you have problems with the exercise, ask your instructor for help

REACT CHAPTER 5

LISTS

CHAPTER PREVIEW

- Lists of data
- Sorting a list

LISTS OF DATA

- In previous exercises, we built a view which could page through a list one item at a time
- Realistically, we will also want to be able to render multiple items in a list
  - Think of a table, or just a list of search results
- React does not come with any API for rendering lists of data
- Instead, it leverages existing TypeScript functionality

RENDERING A LIST

- A list is more commonly called an array
- JavaScript has multiple tools for iterating over arrays
  - `forEach()`, `some()`, `every()`, `map()`, `filter()`, `find()`, etc.
- As a thought experiment, which tool would be useful here?
- We need to iterate over the elements of the list
- And render them as React components
- This is a sort of transformation of one list into another
  - A list of data into a list of components
- The `map()` method will be useful here

USING MAP TO RENDER A LIST

```
import * as React from 'react';

const PersonList: React.SFC<{}> = () => {
  const people: Person[] = [ /* Assume a list of Person objects here */ ];

  return (
    <ul>
    {
      people.map( (person: Person) => (
        <li key={person.id}>{person.firstName} {person.lastName}</li>
      ))
    }
    </ul>
  );
}
```

LISTS AND KEYS

- Rendering (or re-rendering) a list can be an expensive operation
  - There are many elements
  - It is difficult to tell whether elements have changes
  - Many changes to the DOM could be required
- React asks for listed elements to add a `key` property
- The `key` property should have a value unique within this list of elements
  - Different lists can have the same key values
  - Keys should be unique within their lists
- React uses the `key` property to compare items in the list to see if they need re-rendering

EXERCISE 13: LISTS AND KEYS

- Objectives
  - Create a component, `PayeeList`, which renders a list of Payees into a table
  - This will include a component, `PayeeRow`, as well
- Note that some code has been added to `PayeesComponent` to manage whether the UI displays `PayeeList` or `PayeeDetail`
- Also note that when the starter code loads, there will be an error (which will be fixed over the course of the exercise)
- Use gulp to load the class files for the exercise
  - `npx gulp start-exercise --src ex-13`
- If you have problems with the exercise, ask your instructor for help

SORTING A LIST

- What are the pieces of the puzzle if we want to sort the list?
- Arrays can be sorted with their native `sort()` method
- Though that method is used for arrays of primitives, not arrays of objects
- Lodash, the JavaScript utility library includes a `sortBy()` function
  - `sortBy(someArray, sortingFunction)`
  - `sortBy(someArray, [propOne, propTwo])`
- The `sortBy()` utility expects to sort arrays of objects
- It also returns a new array, rather than sorting the array in place
- We would also need an event handler to trigger sorting
- Probably when we click on the header for a column in the list

EXERCISE 14: SORTING A LIST

- Objectives
  - Clicking on a column header should sort our list of Payees
  - Clicking on the same column header again should reverse the sort (from ascending to descending and vice versa)
  - When browsing through `PayeeDetail`, the browse order should be affected by the current sort
- Use `gulp` to load the class files for the exercise
  - `npx gulp start-exercise --src ex-14`
- If you have problems with the exercise, ask your instructor for help

WHAT'S NEXT?

- We would like to do three things with our Payees application
  - Search
  - Add a Payee
  - Edit a Payee
- To get these features, we need to acknowledge that React is coming up against some limitations
  - All our state is stored in Payees, which will not be practical when we try to add or edit a Payee
  - We will be adding three new “views” but have been implementing view management in React, which is not ideal

ADDING TO THE TOOLKIT

- To solve these issues, we will expand our toolkit beyond React
- For state management, we will use the popular Redux library
- For view management, we will add routing to our application with the React Router
- The next few chapters will cover Redux and later React Router in detail
- Then, towards the end of the course, we will return React to work with forms, which we need for our Search, Add, and Edit views

CHAPTER 6: REDUX

INTRODUCTION TO REDUX

CHAPTER PREVIEW

- Why separate state management?
- Why Redux?
- About Redux
- Introducing Redux
- Pieces of the puzzle
  - Actions
  - Reducers
  - The store
- Testing Redux

WHY SEPARATE STATE MANAGEMENT?

- Managing state in React can be complex
  - Particularly as the complexity of an application increases
- Individual components can manage their own state, and some of their children's state
- But components are intended to be UI and view-oriented
- Offloading state management to another tool frees components to focus on their core job
- Additionally, the state manager can be tested independently of the view, which is a significant simplification and benefit

WHY REDUX?

- There are multiple state management solutions for React
- Redux is probably the most popular (at the moment)
- Which means more support and examples on the web
- Redux has extensive documentation
- Redux is very fast
- Redux is east to test
- While Redux is not bound to React, bindings to React exist and are well-documented

ABOUT REDUX

- Website http://redux.js.org
- Videos by Dan Abramov:
  - Getting Started with Redux: https://egghead.io/series/getting-started-with-redux
  - Building React Applications with Idiomatic Redux: https://egghead.io/courses/building-react-applications-with-idiomatic-redux
  - Both video courses are free!
- Github: https://github.com/reduxjs/redux

INTRODUCING REDUX

- Redux is based on a simple principle: There should be a single source of truth for an application
- This source of truth is represented by a store of data
- Any part of the application can query the store for updated data
- Any part of the application can interact with the store to change data as needed
- Interaction points between Redux and the view are well-defined and clear

INTRODUCING REDUX, CONTINUED

- Redux is influenced by two libraries
  - Flux for state management
  - Elm for simplicity
- Redux exists as a state tree roughly parallel to React's component tree
- The overlap is not exact
- Think of the human body: React is like the nervous system, with many branches
- Redux is more like a skeleton: the sturdy support structure

THE PIECES OF THE PUZZLE

- Work with Redux is comprised of three types of objects
- The **store**, which stores the current state
- **Reducers**, which manipulate state
- **Actions**, which define what state manipulations are available
- Actions are **dispatched** to reducers which update the store
- This interaction can sometimes feel rigid and overly-specced
- But remember that the goal here is to have a **single source of truth** whose interactions are very clearly defined

PIECES OF THE PUZZLE: STATE

- At the core of Redux's interactions is your application's state
- Redux maintains the state of your application, which is a JavaScript object with various key-value pairs, extending deeply as needed
- This state tree is held by the store
- The interactions with the state tree are defined by actions and executed by reducers

ACTIONS

- Actions define an interaction with a part of the state tree
  - Maybe a branch of the tree, maybe a leaf of the tree, it is up to the interaction
- Actions have a type and other arguments
  - The type is usually a string constant like 'ADD_TRANSACTION'
  - The other arguments are the information needed to complete the action
  - In this case, presumably, another transaction
  - Or the fields from which a transaction could be built
- Actions are not required to have other arguments
  - Think of a 'CLEAR_SORT' action on a list, for instance
  - But often there is at least one other argument

ACTION GENERATORS

- Action generators are functions which return an object
- The object is, obviously, an action

```
const addTransaction = (tx) => {
  return {
    type: 'ADD_TRANSACTION',
    tx: tx
  }
}
```

DEMO: ACTIONS

- We will be looking at the same file throughout this chapter
- We will be able to see the interactions of components, a store, reducers, and actions
- To see the file, run `npx gulp start-exercise --src ex-15`
- The file is under **src/demos** folder as **ReduxCounter.js**
- For actions, look for the section labeled with the comment `// Actions`

EXERCISE 15: PRELUDE

- Over the next few exercises, we will build a simple example of a Redux-enabled component
- This component would be a part of something we would work with in the real world of React and Redux
- It will serve as a prelude to how React and Redux would work together
- It should help us understand better what's going on with Redux and how it works

EXERCISE 15: ACTIONS

- Objectives:
  - We want to take a Payee and swap it from active to inactive or vice-versa
  - First, we will write an action which will define this interaction
- You should have already used gulp to load the class files for the exercise
  - `npx gulp start-exercise --src ex-15`
- If you have problems with the exercise, ask your instructor for help

REDUCERS

- Reducers manipulate state by processing actions
- Reducers are themselves functions which take two arguments:
  - The current state (or a default)
  - The action to perform on that state
- Reducers return updated state, depending on the action processed
  - Or the previous state, if no action was processed
- Reducers reduce a state and an action to a new state
  - and then return that new state
- Reducers are often named after the state element they work with

REDUCER FUNCTION

```
let reducer = (state = {}, action) => {
  switch (action.type) {
    case 'DO_SOMETHING':
      // Update state in one way
    case 'DO_OTHER_THING'
      // Update state in another way
    default:
      return state; // Do nothing
  }
}
```

STATE MANIPULATION

- Reducers never directly manipulate state
- In fact, Redux state trees should be seen as immutable
- Much like with React, instead of changing state, replace state with new objects
- Reducers should take the current state, copy it, mutate the copy, and then replace the original with the copy
- Behind the scenes, this makes for rapid comparisons between previous and current states
- Which allows React to quickly determine what it needs to re-render

DEMO: REDUCER

- Continue looking at ReduxCounter.ts in the demos folder under src
- The area marked with a comment `// Reducer` is where we can find the reducer defined

EXERCISE 16: REDUCERS

- Continuing to work with our rudimentary React-Redux application
- Objectives
  - Add a reducer called `payee`
  - It should check to see which `action.type` it receives
  - If it is processing `TOGGLE_PAYEE_ACTIVE` it should modify the state accordingly
- Use gulp to load the class files for the exercise
  - `npx gulp start-exercise --src ex-16`
- If you have problems with the exercise, ask your instructor for help

STORE

- A store is a collection of reducers
- Stores are initialized with state from reducers
  - Which provide default state
- Subsequent interactions with reducers manipulate that state
  - As said, never directly, always through replacements
- This is the first time we actually use a Redux function: 'createStore()'
- Create a store by passing a reducer (or set of reducers) to 'createStore()', which will return a store

CALLING A REDUCER ON A STORE

- Actions are passed to reducers on a store by a **dispatcher**
- The store has a `dispatch()` function which takes an **action** as an argument
- Or a function which returns an action
- Internally, the dispatcher figures out which reducer to call based on the `action.type`
- The reducer is called with state (if it exists, default otherwise) and the action object
- The state returned from the reducer is stored internally by the store

ACCESSING STORE DATA

- To see data in the store, there are two options
- At any time, you can call `store.getState()` to see the entire state tree of the store
- To find out about store updates, you can invoke `store.subscribe()`
- The `subscribe()` method takes a function as an argument, which will be invoked on any change
- In the listener function, invoke `store.getState()` to get the current state of the store
- This is a low-level tool we will not use in later chapters
- Though it helps us understand what's going on right now

DEMO: REDUX STORE

- Continue to look at **ReduxCounter.ts** under the **demos** folder
- Pay attention to the areas labeled
  - `// Creating the storeand`
  - `// Working with the storeand`
  - `{/* Dispatching actions */}`

EXERCISE 17: STORES AND TYING IT TOGETHER

- Now we will add a store, which will tie together the functionality of our Redux-enabled component
- Objectives
  - Create a store
  - Initialize the state of the component with the state of the store
  - Subscribe to the store with a listener which updates component state
  - Add a button which dispatches an action to the store
- Use gulp to load the class files for the exercise
  - gulp `start-exercise --src ex-17`
- If you have problems with the exercise, ask your instructor for help

TESTING REDUX

- At the moment, our actions and reducers are locked away in our component file
- We want to test these (of course) but it would be difficult
- Typically, actions and reducers are broken out into separate files which export their respective functions
- This allows for the actions and reducers to be tested independently from the components they work with

REDUX TESTS

- Actual tests are rather uncomplicated
- For action functions, test that the function, given the right input, returns a proper-looking action object
- For reducer functions, passed a state and an action, they should return the correct state
- Store tests are usually redundant if you have tests of actions and reducers

EXERCISE 18: WRITING TESTS

- We have moved the reducer and action into their own separate files, while keeping the original component functioning
- We have added test files under the **tests** folder for the reducer and action as well
  - The files are empty
- Objectives
  - Write tests for the reducer
  - Write tests for the action
- Use gulp to load the class files for the exercise
  - `npx gulp start-exercise --src ex-18`
- If you have problems with the exercise, ask your instructor for help

WHERE DO WE GO FROM HERE?

- In this chapter, we worked on a simple example for the sake of understanding concepts
- There are many aspects of Redux we used in this chapter we would not use in the real world
  - Getting state from `store.getState()`
  - Using `store.subscribe()` directly
  - And so on
- In the next chapter, we will tie React and Redux together "properly" using well-established patterns

CHAPTER 7

REDUX AND REACT

CHAPTER PREVIEW

- Redux and React
- Tools for Redux and React
- Tying state and props together
- Tying events and dispatches together
- Connecting a store to a component
- Testing?

REDUX AND REACT

- Redux and React have a lot in common
  - Trees of representations
  - Information flows downward in the tree
  - Or from root to branch to leaf, depending on your perspective
  - Changes trigger re-evaluation
- It is not difficult to envision tying Redux and React together with a small library which would make it easier to manage one from the other
- Which is, in fact, what the `react-redux` module does

TOOLS FOR REACT AND REDUX

- The react-redux module imports two special tools for working with React under Redux
- The `Provider` component makes it easy to make the store available to every branch of the tree
- The `connect()` function wires together a component and a store with details on how to hook up state to props and dispatch actions based on events
- The module does not really provide much else, for other features we would have to add functionality from Redux or an external library

USING THE CONNECT FUNCTION

- The connect function connects a React component to a Redux store
- It does so through two sets of mappings:
  - state to props, known as `mapStateToProps`
  - dispatch to props, called `mapDispatchToProps`
- Conceptually, `connect()` is plugging the component into Redux at two connect points: props and events
- When the store is updated, it can update your component's props
- When custom events are called, they can dispatch actions to the store, updating the state

CONNECT'S SIGNATURE

```
import { connect } from 'react-redux';

const ComponentToBeWrapped = () => (
  // Component implementation
);

const mapStateToProps = {};
const mapDispatchToProps = {};

const wrappedComponent = connect(
  mapStateToProps,
  mapDispatchToProps
)(ComponentToBeWrapped)
```

TYING STATE AND PROPS TOGETHER

- The first argument to connect is `mapStateToProps`
- `mapStateToProps = (state, [ownProps]) => ( { state to prop map } )`
- `mapStateToProps` is a function
- Arguments: store state and, optionally, props passed to the wrapped component
- Returns: An object mapping of store state to props in the wrapped component
- Think of `mapStateToProps` as the subscriber to store updates

TYING EVENTS AND PROPS TOGETHER

- The second argument to connect is `mapDispatchToProps`
- `mapDispatchToProps` can be either a function or an object
- If it is a function…
  - Arguments: The dispatch method from the store
  - Returns: an object mapping of props (usually event handlers) to dispatchers (usually action creators)
- If it is an object, it is what the functional version returns
- That is, an object mapping of props to dispatchers

LOOKING AT CONNECT

- Here's a basic component

```
const CustomizedButton = ({buttonText, onClickCucstom}) => (
  <button onClick={onClickCustom}>{buttonText}</button>
)
```

- The CustomizedButton component takes two parameters
  - `buttonText` which will be the label for the button
  - `onClickCustom' which is the custom event handler for the button
- Let's plug this into Redux

THE REDUX SIDE

- On the Redux side, this button might be used in a form
- Depending on the state of the form, the button might have different text
  - Or maybe it's internationalized and language-aware, who knows
- And we want the button to dispatch an action

```
// action generator

export const saveForm = (formData: any) => ({
  type: 'SAVE_FORM',
  formData
});
```

THE REDUX SIDE, CONTINUED

- The reducer that provides the button text
  - Perhaps a bit of license with the button labels

```
// reducer
const viewLabels = (state = {buttonText: ''}, action) => {
  case 'SHOW_ADD_FORM':
    return {buttonText: 'Add new thing'};
  case 'SHOW_EDIT_FORM':
    return {buttonText: 'Edit existing thing'};
};
```

WHAT SHOULD HAPPEN?

- When this particular `CustomizedButton` is rendered, it should use the label from the `viewLabels` reducer/state
- When this CustomizedButton is clicked on, it should dispatch a `saveForm()` action.

PLUGGING IN THE COMPONENT

```
import { connect } from 'react-redux';

const CustomizedButton = ({buttonText, onClickCucstom}) => (
  <button onClick={onClickCustom}>{buttonText}</button>
)

// mapStateToProps gets free access to the complete Redux state
const mapStateToProps = (state) => {
  return {
    buttonText: state.viewLabels.buttonText
  }
};

// mapDispatchToProps gets access to the store's dispatch() function
const mapDispatchToProps = (dispatch) => {
  return {
    // Assume we captured the form data somehow
    onClickCustom: () => dispatch(saveForm( formData ))
  }
};

// WHOA
export default connect(mapStateToProps, mapDispatchToProps)(CustomizedButton);
```

WHAT JUST HAPPENED?!

- What just happened in that last line?
- The `connect()` function returns a function, customized by `mapStateToProps` and `mapDispatchToProps`
- This is actually a utility function, **unattached** to any component
- But if you pass a component into this utility function, it will return a wrapped version of the component
- The wrapped version of the component is plugged into Redux
  - It is auto-subscribed to updates from the `store`
  - Its event handlers dispatch actions to the `store` as well
- Cool!

CONTAINERS AND CONNECTIONS

- The Redux docs point out accurately that wrapping a presentational component in a `connect()` call essentially generates a container component
- The connected component which results is **smart** by virtue of connect plugging the store into the wrapped **dumb** component
- The container component is a thin layer which interacts between the presentational component and the Redux store

APPLICATION-LEVEL REDUX

- We want to connect the store to many components in our application, or at least be able to
- How can we make a store available to multiple levels of an application?
- The react-redux bindings provide a utility component for this: `Provider`

PROVIDER

- The `Provider` component should be the root component for your entire application
- Wrap it around `App`, for instance
    - `Provider` as typed in `@types/react-redux` in fact **requires** you to wrap the root node of your application.
- Create the store and then pass it as a prop to Provider
- Provider takes advantage of a React global called context
- We will not be going over context here

EXERCISE 19: REDUX AND PAYEEBROWSER

- We are going to rebuild PayeeBrowser (including the PayeeDetail and BrowserButtons components) as a component connected to a Redux store
- Objectives:
  - Create a store to manage `PayeeBrowser`'s state
  - Create action(s) and reducer(s) to interact with the store
  - Connect the store to `PayeeBrowser` using 'connect()'
- Use gulp to load the class files for the exercise
  - `npx gulp start-exercise --src ex-19`
- If you have problems with the exercise, ask your instructor for help

REDUX-REACT ARCHITECTURE

- Container components can exist without being run through `connect()`
- Though usually a parent container will then be connected
- Container components can also interact with dispatching directly
- A container component that has been wrapped with `connect` has access to the `store.dispatch()` method on `props` as `props.dispatch()`

EXERCISE 20: REDUX AND PAYEES

- Now we will re-implement `PayeesContainer`, taking full advantage of Redux
- Objectives:
  - Create a store to manage Payees state
  - Create action(s) and reducer(s) to interact with the store
  - Connect the store to Payees using `connect`
- Use gulp to load the class files for the exercise
- `npx gulp start-exercise --src ex-20`
- If you have problems with the exercise, ask your instructor for help

TESTING?

- Should we write tests for the application?
- On the one hand, nothing in the view has changed, so our React-oriented tests should still run fine
- On the other hand, we have changed out state management to Redux, so anything that interacted with state will need to change
- We also have moved our actions and reducers into separate files, for ease of testing
- While the content we are testing has changed, the methodology has not

EXERCISE 21: TESTING OUR COMPONENTS

- Objectives
  - Go over the React-oriented tests for our components, see if any need changing
  - Write Redux-oriented tests for our actions and our reducers
- Use gulp to load the class files for the exercise
- `npx gulp start-exercise --src ex-21`
- If you have problems with the exercise, ask your instructor for help

CHAPTER 8

MIDDLEWARE AND THUNKS

CHAPTER PREVIEW

- Middleware in general
- The Redux Logger
- The Redux devtools
- Asynchronous middleware
- Asynchronous actions with thunks

MIDDLEWARE IN GENERAL

- Redux allows for registering middleware on the store
- Middleware has a chance to look at any dispatched actions before Redux does
- Middleware can also look at any information coming back out of the store (in the event of a state update) before it goes out
- And, of course, middleware could do both!
- Register middleware with the `applyMiddleware()` function from the `redux` module

THE REDUX LOGGER

- Let's see some middleware in action

```
import { createStore, applyMiddleware } from 'redux';
import { createLogger } from 'redux-logger';

// Later on
const store = createStore( reducer, applyMiddleware( createLogger() ) );
```

REDUX LOGGER EFFECTS

- Try adding the code above to any of the exercises that use Redux
- You will see that the Redux logger sends logging information to the console on every state update
- Every time an action is dispatched, the Redux logger logs the action, the previous state, and the new state
- Quite nifty, really, and handy for debugging

THE REDUX DEVTOOLS

- If you would like a greater set of tools for finding out what's going on with Redux, you can add the Redux devtools as middleware
- The Redux devtools are actually a mini React application that let you see everything you have done in Redux
  - They include tools for examining and modifying the current state, recording actions, traveling through time (well, the time of your Redux application)

ADDING THE REDUX DEVTOOLS

- First, you should install the extension appropriate to your browser for the devtools
  - You can actually use the devtools without an extension, but they will take up a chunk of your page, which we don't want

ADDING THE REDUX DEVTOOLS, CONTINUED

```
import { composeWithDevTools } from 'redux-devtools-extension';

// other code

// Used alone
const store = createStore( reducer, composeWithDevTools() );

// Used with other middleware
const store = createStore( reducer, composeWithDevTools(
  applyMiddleware( /* other middleware */ )
) );
```

- This assumes you are replacing the logger with the devtools
- `composeWithDevTools` is a convenience function provided by `redux-devtools-extension`

USING THE DEVTOOLS

- In your browser, open the Developer Tools
- Click on the **Redux** tab
- Enjoy the wonderfulness of the Redux devtools!

ASYNCHRONOUS REDUX

- Using Redux asynchronously introduces a new set of complications
- Chief among these is how to deal with an asynchronous process
- An async request goes through two of three possible phases
  - First the request is sent
  - Then, either the request completes successfully
  - Or it fails
- Use Redux, these are three different states:
  - Loading
  - Loading complete & success
  - Loading complete & failure

ASYNCHRONOUS ACTIONS

- There are several different patterns for invoking actions asynchronously
- We will cover a low-level, simple-but-effective pattern, using a concept called a thunk
- The thunk will allow us to tell Redux to run a series of branching actions
    - Which matches the series of request -> response or error in general HTTP requests

WHAT'S A THUNK?

- A thunk is a subroutine created to assist a call to another subroutine
- In functional programming, thunks are used to pass work or behavior from one function to another
- Because functions are first-class citizens in JavaScript, thunks are a logical way to pass executable code from one part of our application to another
- When we write a thunk, it will be a function which returns a function to be executed later

REDUX AND THUNKS

- Redux does not handle a thunk natively
  - Remember that Redux stores expect action objects, not functions
- Adding the redux-thunk package as middleware allows Redux to handle thunks
  - Technically, the middleware handles the thunk and passes proper action objects to Redux
- Register the thunk middleware with your Redux store
- Invoke action generators which return functions the same way you would action generators which create objects

SETTING UP FOR ASYNC

- Import `thunk` from `redux-thunk`
- Import `applyMiddleware` from `redux`
- When creating the store, use middleware
  - `createStore(reducers, applyMiddleware(thunk))`
- And you are done!
- From now on, your action creators can return functions which will run code
  - The returned functions should, in turn, dispatch actions
  - They can still return just objects, too

REGISTERING THUNKS

```
import {createStore, applyMiddleware} from 'redux';
import thunkMiddleware from 'redux-thunk';

const store = createStore(
  reducer,
  applyMiddleware( thunkMiddleware )
);
```

THUNK ACTIONS

- At their most basic, thunk actions are functions, instead of typical action objects
- It is easier to write an action generator that returns the appropriate thunk
- A basic example:

```
const thunkGenerator = (args: () => void) => {
  return function () { /* whatever */ };

  // Or

  return () => ( /* whatever */ );
}
```

ASYNCHRONOUS ACTIONS

- Independent of their implementation, asynchronous actions are complicated
  - Becuase they are, you know, **asynchronous**
- State can change several times during an asynchronous request
  - Request start
  - Successful response
  - Error response
  - Parsing data
  - ...possibly more
- Our code will have to deal with these possibilities

ASYNCHRONOUS ACTIONS

- Here are basic actions for handling success

```
interface FetchStatus {
  type: string
  isLoading: boolean;
  payload?: any;
}

const sendRequest = (): FetchStatus => ({
  type: 'REQUEST_SENT',
  isLoading: true
});

const fetchSuccess = (payload: any): FetchStatus => ({
    type: 'REQUEST_SUCCESS',
    isLoading: false,
    payload
});

const fetchFailure = (payload: any): FetchStatus => ({
    type: 'REQUEST_SUCCESS',
    isLoading: false,
    payload
});
```

DISPATCHING ACTIONS

- When should we dispatch these actions?
- Inside another action...?
- It sounds weird, but yes, we want to have a higher-level action which manages the fetching process
- This is also where our thunk comes from

RUNNING A FETCH

- A sketch of a fetch, if you will

```
const fetchData = (dispatch) => {
  dispatch(sendRequest());

  // The fetch API is the replacement for Ajax in modern browsers
  fetch('http://localhost:8001')
    .then( (response: Response) => {
      if (response.ok) {
        return response.json()
      } else {
        return Promise.reject('Could not find data');
      }
    })
    .then( results => dispatch( fetchSuccess(results) ) )
    .catch( err => dispatch( fetchFailure(err) ) )
}
```

FETCH EXPLANATION

- The function starts by dispatching notice that the request has begun
- The fetch API makes a request to a URL
- The request will return a reponse or an error
- If we get a response, we check that it's ok
    - If so, parse the json of the response (which returns another promise)
    - If not, return a rejected promise

FETCH EXPLANATION CONTINUED

- In our second-level promise, we have the actual, parsed results we were looking for
- Dispatch these results to `fetchSuccess()`
- Use the `catch()` function on a Promise to catch any errors thrown by the promise
- Dispatch errors to `fetchFailure()`

EXERCISE 22: ASYNCHRONOUS THUNKS

- Objectives
  - Move store creation to its own file
  - Add thunk middleware to the store
  - Add action creators for downloading the payees portion of the state asynchronously
  - Add code to the reducer to handle this asynchronously-obtained state
- Use gulp to load the class files for the exercise
  - `npx gulp start-exercise --src ex-22`
- If you have problems with the exercise, ask your instructor for help

CHAPTER 9

THE REACT ROUTER

CHAPTER PREVIEW

- What is routing?
- What does routing do for me?
- The React Router library
- Basic Routing configuration
- Accessing parameters

WHAT IS ROUTING?

- Routing allows tying a view to a URL
- Enter a particular URL, and a particular view (or set of views) appears
- Each URL is also an entry into the browser's history stack

WHAT DOES ROUTING DO FOR ME?

- Routing allows us to tie components to URLs
- Changing the URL in our browser will change the view
  - Without losing state
- This has all the advantages of working with URLs
  - Added to browser history
  - Forward and back buttons work
  - Less fear of accidentally losing state by switching URLs
  - Bookmarkable and sendable

ROUTER IMPLEMENTATIONS

- There are several implementations of routing which are compatible with React
- The most popular, and the one we will be looking at, is `react-router`
- The React Router is a third party router which uses an adaptable, flexible, component-based architecture
- The React Router library is a third-party tool and is not supported by the React team
- We are interested in the web-based router, found here: https://reacttraining.com/react-router/web/
    - Also called `react-router-dom` when imported

BASIC ROUTING CONFIGURATION

- Basic routing configuration requires two components, the Router and the Route
- Typically, an application has only one Router component, either wrapping the App component, or as the first child of the App component
- An application will have several Route components, matching various URLs

THE ROUTER COMPONENT

- The Router component is actually the BrowserRouter component
  - React Router has a MemoryRouter and a few others
- The React Router style is to import BrowserRouter as Router
- Make the Router component either the parent or the child of the App component
  - The Router component can occur several levels below the App component if you want, but since we will be using the Route components throughout our application it makes sense to have it fairly high up in the component tree
- You can add a configurable history object, or rely on the one the BrowserRouter creates automatically

THE ROUTE COMPONENT

- Somewhere under the Router (possibly by several layers), you will start using the Route component
- The Route component takes, as a minimum, two attributes
  - `path`: What URL path should activate this route
  - `component`, `render`, or `children`

THE ROUTE COMPONENT, CONTINUED

- The second property passed to `Route` determines what will be loaded for this URL:
- `component`: A reference to a component
- `render`: A function which returns the value to be rendered when this route is activated
- `children`: A function which returns the value to be rendered regardless of whether this route is activated
- The `component` and `render` properties are more commonly used

DEMO: BASIC ROUTING

- Use gulp to load the code for the upcoming exercise:
  - `npx gulp start-exercise --src ex-23`
- This will add the `Router` component to `App` and add two new routes, `demos` and `payees`.

ACTIVATING A ROUTE

- There are two ways to activate a route: programmatic and declarative
- Programmatic activation occurs in JavaScript code
- Declarative activation occurs in the HTML
- Let's look at the easier one, declarative activation, first

THE LINK COMPONENT

- The Link component generates an anchor tag that will point to the URL specified in the component's to attribute
- The Link component is used for declarative activation of a view / route
- It is simple and straightforward
- Check out **src/Navbar.js** to see the `Link` component in action

EXERCISE 23: BASIC ROUTING

- Objectives
  - Note configuration of the Router in our application
  - Add Route configuration for "top-level" routing between Payees, Categories, and other areas of the application
- You should have already used gulp to load the class files
  - `npx gulp start-exercise --src ex-23`
- If you have problems with the exercise, ask your instructor for help

CONCLUSION