Contents

Chapter 1

Introduction

ReactJS Fundamentals

What is React?

- A component library written by developers from Facebook
- A way of building applications that focuses on components
- A collection of other APIs, some by Facebook, some by others, that circle around the core API

React is a View/Controller technology

- React Components are made centrally from a `render` method, it's the only required function
- Components can have internal state, and can share read-only properties with children
- Components have a one-way data flow (downward) and event flow (upward)
- React does not force you into a specific model technology (you can use plain old JavaScript objects and functions)

Key React Concepts

- Components maintain a "Virtual DOM" which is a memory-based model of the contents of the DOM. The differences between the Virtual DOM and the real DOM are computed and applied as diffs.

- Components have a lifecycle and can react to initialization, changes in state, modified properties, and can control when and what to update.
- React makes view development simpler with JSX

React History

(courtesy Wikipedia)

- Originally authored by Jordan Walke - software engineer at Facebook
- Began by running FB news feed in 2011, Instagram in 2012
- Released in March 2013
- Open Sourced using a 3-clause BSD license and a Facebook "patent troll protection addendum" - at JSConf US 2013

Constellation of APIs

- `Redux` - a state management API, following (loosely) the Flux pattern
- `React-Router` - a standard SPA router package
- `Flow` - a JavaScript static typing system
- `GraphQL` and `Relay` - Declarative object data stores and data fetch engine
- Developer tools and Chrome Plugins
- Much more...

React -vs- Angular 1 -vs- Angular 2

- Angular 1 is a large, stable, slower framework to write SPA applications
- Angular 2 is a large, unstable, fast framework that will eventually let you write applications
- React is a tiny, stable, fast core with a constallation of other APIs that lets you opt in and write applications

React and JavaScript versions

- React works fine in ES5
- But the React team embraces ES6
- All new development uses ES6 classes
- The React Appplication Starter tool installs Babel and other tools to make this easy to deal with
- *some* people are interested in TypeScript with React but it would be good to look at Flow as well

Course Goals

- We'll take you through the core React API
- You'll learn how to
- Develop Components
- Work with state
- Work with properties
- Deal with events, forms
- Integrate Redux as a local data store
- Configure the React Router
- Use the development tools

But first we have to learn a little ES6...

A Semi-Gentle Introduction to ES.2015 and ES.2016

React Fundamentals Training

Chapter 2

Time marches on

- . . . and so does the TC-39 committee

ES2015 is

- ES.Next, ES6, etc. . . - All of the recent changes.

ES2016 is

- ES7, the newest updates

So Why Do We Care?

- React is supporting and encouraging use of modern JS
- Certain features, like modules, destructuring and the spread operator are vital to keeping code complexity down
- Once you begin to code in ES2015+, ES5 seems quaint

Major Features to be Aware Of

- Arrow Functions - handle the `this` conundrum

```
let sayHello = (name) => { console.log(`Hello, ${name}`); };
```

- I snuck in backtick strings - which can do substitutions and multiple lines

```
let template = `
<p>
One, two, three
</p>`;
```

The `let` and `const` keywords

- `let` introduces true block scope, and `const` makes the variables read-only

```
const msg = 'ABC';
for (let i = 0; i < 100; i++) {
  console.log(`${msg} - ${i}`);
}
```

Modules - defining

```
// in abc.js
let x = 234;
let doSomething = function() ...

export x, doSomething;
```

Modules, using

```
// in def.js
import {x, doSomething} from './abc';
doSomething(x);
```

What was the { } about?

- Destructuring - pulling multiple variables out of a module at once
- If a module `foo` exports `a`, `b`, and `c`:

```
// get them as three vars
import {a, b, c} from './foo';

// get an object with members a, b, and c
import * as foo from 'foo';
// foo.a, foo.b, foo.c are now defined
```

Flattening using ...

- Say you have to pass elements into a method one-by-one, but they are packed as an array:

```
// this is an ES2015/ES6 features
let x = String.concat(...FLAGS);
```

- Say you have to pass the fields of one variable into another

```
// this is an ES2016 / ES7 feature

let fields = {
   ...otherObjectProps,
   a: 123,
   b: 444
};
```

Classes and Constructors (featuring Modules)

```
export class Customer extends Printable {
  constructor(name) {
    this.name = name;
  }
}
```

- We exported the class constructor `Customer` from this module / file

What are ES Classes Anyway?

- They act like constructor functions
- They translate to constructor functions in ES5
- But they can provide encapsulation and useful semantics

```
export class Customer {
   constructor(name) { this.name = name }

   sendMessage(message) {
     console.log(`${message}, ${this.name}`);
   }
}
```

- Static methods are also supported

Getting ES2015 and ES2016 Now

- Use `babel` in your builds
- Can be run from Gulp, Grunt, WebPack, and many other build systems
- Can translate your code to ES5
- Combine this with shims for various features
- Most newer browsers can run your apps without forcing you to code ES5 directly
- We will use the `React Project Creator` to build our projects while supporting ES2015/2016 through Babel transpilation.

Questions?

Chapter 3

ReactJS and its Ecosystem

ReactJS Fundamentals

Topics

- React and its Ecosystem
- Flux pattern, Redux and other implementations
- Middleware
- Flow - types in JavaScript
- React-router
- React-redux, react-redux-router, etc...
- `npm` and build ecosystem
- `developer tooling` with ReactJS Chrome Plugin, Redux tools

React Ecosystem - TODO - Diagram

[React in center, spinning out are Flow, Redux, React Router and middleware (attached to it), React Native, DevTools

Persistence - The Flux pattern

- Flux is an architectural pattern for application state management

- Data is kept in a ‘store’
- Actions operate on the store
- Callbacks notify the subscribers of the store that the state has changed
- This architecture emerged from Facebook in 2014

Redux - a functional implementation of Flux

- Redux implements the Flux design requirements
- The stores keep immutable data (the structures cannot change, they must be replaced when the state changes)
- Actions make requests against the store, handing it an action key and a payload
- The store runs its *reducer* function to process the request - a pure function
- The reducer either replaces the state or does nothing
- Actions can be executed synchronously or asynchronously
- Redux has a robust middleware API for plugging in various feature

Redux Middleware

- Provides a number of key features
- Can be installed as needed by developers
- Sample middleware - redux-logger, redux-thunk, redux-saga
- Redux developer tools - redux-dev-tools, redux-immutable, etc. . .
- See - https://github.com/xgrommx/awesome-redux#react—a-javascript-library-for-building-user-interfaces

Flow - Datatypes in JavaScript

- Facebook Flow allows you to define and statically check types

```
//@flow
function add(customer) { customers.push(customer); }

// will throw an error - 12 doesn't have a `push` method
add(12);
```

- Use `@flow` to typecheck a file, others not checked

Flow with Type Annotations

```
//@flow
function add(customer: Customer) { ... }

// throws an error, more specifically that 12 is not a customer
add(12);
```

Types Supported

- primitives - number, string, boolean
- Arrays and Maps
- Object types

And more, see https://flowtype.org/docs/syntax.html

React-Router

- Provides view routing in a heirarchical manner
- Routes are delivered to components
- Components can nest other routes
- Complex URL patterns are available

Integrating Libraries

- Can be a challenge as nobody wants to own the chore of documenting approaches
- We will show you how to

- Install Redux and integrate it with React
- Install the React Router and integrate it with React
- Interact with APIs such as Ajax calls
- Discuss issues and extra APIs for additional integration between Router and Redux

React Native

- An implementation of React that creates native web applications
- Uses native iOS, Android developer kits
- Implements React component APIs in native APIs
- Allows for debugging of React Native applications in iOS, Android simulators

create-react-app

- A NPM module and tool that creates a simple React project
- Allows for additional dependencies (Redux, etc) via `npm install --save projectname`
- Execute your project with `npm start`
- Same configuration for everyone allows for good support questions
- Many, many other starters exist

Tools

- React Dev Tools - a browser plugin for Chrome that shows React components, events, data
- Redux Dev Tools - state inspector for Redux, includes time travel debugging

Chapter 4

Introducing Components

ReactJS Fundamentals

Topics

- The Component API
- Displaying a Component
- JavaScript
- JSX
- Building with Components
- (Synthetic)Event Handling
- Stateful -vs- Stateless Components
- Presentational and Container Components
- Context and Global State

Introducing Components

- Extend `React.Component`
- Must implement `render()`
  - written with JavaScript or JSX
- Optional Lifecycle Methods
  - Mounting
  - Updating
  - Unmounting

A Minimal Component (JavaScript)

```
class Square extends Component {
  render() {
    return React.createElement('button',
         null, // no attributes
        'X');
  }
}

export default Square;
```

A Minimal Component (JavaScript)

- `createElement` parameters:
  - custom component (Class) or standard HTML tag (string)
  - properties and/or events (key/value pairs)
    - standard HTML, React built-ins, or custom properties
  - any child elements
- Factories can be used to create standard DOM elements

```
React.DOM.div(null, 'X');
```

A Component is a Single Element

- A component can only include 1 top-level `React.createElement` call
- One or more additional React Elements and/or standard HTML elements can be passed in to top-level React Element as child elements

```
React.DOM.ul(null, React.DOM.li(null,'Tic'),
                   React.DOM.li(null ,'Tac'),
                   React.DOM.li(null, 'Toe'));
```

Mounting a Component (JavaScript)

- `ReactDom.render` parameters:

  - a *root* React Element
  - a DOM container for the *root* Element

in `index.js`

```
ReactDOM.render(
  React.createElement(Square),
  document.getElementById('root')
);
```

in `index.html`

```
<div id="root"></div> <!-- Doesn't have to be named 'root' -->
```

Specifying Properties (JavaScript)

- Composite attributes
  - Use camelCase
- Boolean attributes
  - specify `true` or `false`
- JavaScript reserved words
  - Use `className`, not `class`
  - Use `htmlFor`, not `for`

```
React.createElement("button",
  {disabled: true, className: "square"},
  'X')
```

Specifying Inline Styles (JavaScript)

- Use camelCase, not dashes
- Use an object to specify styles

```
React.createElement("button",
  {style: {fontSize: "20px", color: "blue"}, disabled: true},
  'X')
```

`Non-DOM` Attributes

- `ref`
  - Used to access a DOM node or React component class instance
  - Most commonly used for setting focus or obtaining a `ref` to pass to a 3rd party library
- `key`
  - When child components are represented by Developer assigns a unique `key` to each item in a collection, exclusively for use by React
  - React uses `key` to determine whether an item's DOM node can be reused or destroyed

`Non-DOM Attributes` Cont'd

- `dangerouslySetInnerHTML`
  - Helps protect against XSS attacks
  - Provided by React as a safer alternative to setting `innerHTML` directly
    - React does not allow `innerHTML` to be set directly

Mounting Children with Iterators

- A Tic-React-Toe Board with 9 Numbered Squares

```
createSquare = (id) => React.DOM.button(null, id + 1)

render() {
   return React.DOM.div({style: {width: "75px"}, key: 'a'},
      React.DOM.hr({key: 'b'}),
      [...Array(9).keys()].map((id) => this.createSquare(id)),
      React.DOM.hr({key: 'c', style:{borderColor:'red'}}))
}
```

- With JSX only dynamically generated children need keys; with JavaScript, all children need keys

The Case for JSX

- Most React developers use it
- Facilitates working with designers
- Easier to picture a layout by quickly scanning it

```
render() {
    let ids = [...Array(9).keys()];
    return (
    <div style={{width: "75px"}}>
      <hr/>
        {ids.map((index) => <button key={index + 1}>
                                {index+1}
                              </button>)}
      <hr style={{borderColor:'red'}}/>
  </div>
 )}
```

JSX Basics

- HTML-like
- philosophy: keep markup and display logic together
- attribute values can be strings
- attributes can be JavaScript expressions
- wrap JavaScript expressions in curly braces
- if/else is not supported within JavaScript expressions
- use the ternary operator or switch statement

```
<button>{false ? 'O' : 'X'}</button>
```

JSX Basics

- JSX generates JavaScript, not HTML
- it generates calls to `createElement`
- Use JavaScript comments in curly braces

```
{/* This will not show up in the browser */}

<!-- This will show up -->{// but not this }
```

```
{/* or
     this
/*}
```

Differences from JavaScript

- Inline style syntax

```
<div style={{'color: red'}}/>
```

- Boolean properties
  - The following are equivalent

```
<div disabled />
<div disabled=true>
```

Building with Components

- Structure of `Tic-React-Toe` Component Demo App
  - Gameboard
    - Square(s)
    - Game Piece(s) ————————-
    - Theme Picker
    - Theme Button(s)

Configuring a Component with `props`

- Setting `props` is similar to setting HTML attributes
- This square is setting `prop` values on it's GamePiece, based on `props` passed to it from the Gameboard

```
<div className="square" onClick={this.onClickHandler}>
  <GamePiece mark={this.props.mark}
             markColor={this.props.markColor}
             squareId={this.props.squareId}/>
</div>
```

Passing Props

- It's often best to specify `props` explicitly, but React supports a `bulk` passing mechanism, courtesy of the spread operator

```
<div className="square" onClick={this.onClickHandler}>
  <GamePiece {...props}/>
</div>
```

Validating Props

- A PropTypes provides development mode guarantees against missing props and wrong types.
- You can specify types or object shapes or write a custom validator

```
Square.propTypes =
          { turnHandler: React.PropTypes.func.isRequired,
            markColor: React.PropTypes.string };
```

Passing Child Elements

- Use `props.children` to access elements placed between the start and end tags of a custom component, such as `Gameboard` and `Leaderboard` below

```
<Gameframe gameName="Tic-React-Toe">
    <Gameboard/>
    <Leaderboard/>
</GameFrame>
```

Accessing Child Elements

- Positioning `Gameboard` and `Leaderboard` under game name and between 2 lines:

```
class GameFrame extends Component {
  render() {
      <hr style={{borderColor:'green'}} />
      <div style={{textAlign: "center"}}>
      {this.props.gameName}
       </div>
```

```
      {this.props.children}
      <hr style={{borderColor:'green'}}/>
  }
}
```

Iterating Through Child Elements

- Displaying child elements in a table
- the `children` API also include `forEach` and `only`

```
render() {
  return (
<div>
  <table>
    <tr>
      {this.props.children.map(function(child) {
         return <td>{child}</td>
       })}
    </tr>
  </table>
</div>
   )
}
```

Accessing `Real` DOM Elements

- Assign a variable assignment function to `ref`
- React will call it when the element is instantiated
- In most cases, the component should be accessible in `componentDidMount`

```
export class FormPanel extends React.Component {

 setBtn(c) {
   this._btn = c;
 }
 render() {
   <button  ref={(c) => this.btn = c}
            onClick={this.resetHandler}>
            Reset
   </button>
 }
```

Stateful Applications

- Initialize state in the component constructor

```
this.state = {
  moves: {},
};
```

- Use `setState` or `replaceState` to update state
- React *schedules* an update setState is called; the actual state change happens asynchronously

Tracking State

- Because `setState` only queues the update, React supports an optional callback parameter – a function that it calls after the update actually fires
- Even the Tic-React-Toe demo app needs to be aware of whether it is evaluating current or pending changes. It needs to know if it should count 2 squares plus the latest move, or 3 squares, to determine if the latest move ended the game

Chapter 5

What to Store

- Do not store anything that can be derived
- In Tic-React-Toe, only a set of key/value pairs with the key representing the turn number and the value representing the marker (X or O)
  - everything else can be derived from that

Chapter 6

Stateless Components

- Stateless components are similar to pure functions in that they always behave exactly the same way, given the same input (`props`)
- These components can be expressed as functions instead of classes.
- The render method becomes the function body
- Behind the scenes React still generates reactElements for these types of components. They are not really functions, but this option

Chapter 7

Global State with Context

- The experimental context feature enables components anywhere along a component hierarchy to access data, without needing for it to be passed down
- Redux uses this feature heavily
- The React community recommends only using this feature through a 3rd party library

ReactJS State Management with Redux

ReactJS Fundamentals

What is Redux?

A container for application state that * aggregates all non-transient state into a single root object * provides a mechanism to change the state * provides a mechanism to be notified when state changes

Why is it used?

React applications with distributed state can become complex to manage * State is distributed throughout the application * Hard to reason about * data changes * data relationships * No easy way to add tools such as logging and debugging

How to install

Install the npm dependency:

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

Import and create the store

```
import {createStore} from 'redux';
let store = createStore(...);
```

Reducers

- Reducers are functions that operate on the current stored state
- They are given the currently stored state and an *action* (command object)
- The action *must* contain a *type* property
- The reducer will introspect the *type* property and either return the current state or create a new state based on other information from the action

A sample Reducer

```
function calculatorReducer(state = 0, action) {
  switch (action.type) {
    case 'ADD':
      return state + action.value;
    case 'SUBTRACT':
      return state - action.value;
    default:
      return state;
  }
}
```

Calling the reducer with dispatch

```
import { dispatch } from 'redux';

dispatch({ type: 'ADD', value: 10});
```

- Dispatch synchronously calls your reducer

Subscribing to state changes

The brute force way:

```
store.subscribe(() => {
  let updatedState= store.getState();
});
```

- We'll see a better way soon
- But this can be tested easily

Redux State management rules

- Never mutate state (only return new objects)
- Reducers are idempotent (same inputs = same store values)
- Reducers should not contain business logic

Actions

- Actions *request* changes to the store
- They can contain validation, business logic
- They can mutate data structures
- They `dispatch` the action to the store with assembled data

Introduction to middleware

Middleware sits between action requests and reducer responses

Can perform tasks such as

- Logging
- Exception handling
- Security/policy enforcement

Chapter 8

Logging middleware

Install with

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

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

let logger = createLogger();

// middleware at the end
let store = createStore(reducerFn,
                applyMiddleware(logger));
```

Interactive Demo - Plain Redux

Redux with React

Redux was built to use within a React application * React handles the UI concerns * Redux handles state * The (`react-redux`) library provides React bindings for Redux * Redux state and actions become props in a React application * React automatically refreshes the component as data changes in the store

What to Store

- Do not store anything that can be derived
- Functions that do computations or derive values based on state changes are referred to as *selectors*

- The `reselect` library generates memoizable `selectors` that only recalculate when their arguments change

Where to create the store

- Top-level component that bootstraps your application

```
import { Provider } from 'redux';

let store = createStore(reducer);

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

Action Creators in a React Application

```
export const takeTurn = (squareIndex) => {
  return {
    type: 'TAKE_TURN',
    squareIndex
  }
}

export const reset = () => {
  return {
    type: 'RESET'
  }
}
```

Dispatching Actions

- Just as parent components pass event handlers to child components, Redux makes dispatching methods available in the form of `props`
- Use the Redux library's `mapDispatchToProps` to declare names for the dispatching actions

Mapping Actions to prop Names

```
function mapDispatchToProps(dispatch){
  return {
    takeTurn: (squareIndex) => {
      dispatch(takeTurn(squareIndex))
    },
    reset: () => {
      dispatch(reset())
    }
  }
}
```

Dispatch Mapping Shortcuts

- To give action dispatching methods the same name as the corresponding actions, use `bindActionCreators`

```
function mapDispatchToProps(dispatch) {
 return {
    actions: bindActionCreators(Actions, dispatch)
  }
}
```

Subscribing to the Redux Store

- Redux's `connect` encapsulates the subscription logic
- Pass `connect` handles to `mapDispatchToProps` and `mapStateToProps`
- Components that subscribe to the store are referred to as *connected* components

Subscribing to the Redux Store

- `connect` is a higher-order function; to connect a component to Redux, pass it to the return value of `connect`

```
export default connect(
  mapStateToProps,
  mapDispatchToProps
)(GameboardContainer)
```

Using Multiple Reducers

- Use `combineReducers`
- The store will be name-spaced
- use `state.gameReducer` & `state.themeReducer`

```
const themedGameReducer = combineReducers({
  gameReducer, themeReducer
});

export default themedGameReducer;
```

Which Components Should Be *Connected*

- Commonly, the highest level component is the only *Connected* component
- Connecting some mid-level components can be a good way to reduce the number of `props` that have to be passed down through multiple levels

TODO mapStateToProps

Top-level only with props downward

Connect containers, props to dumb stateless components

Challenges with tangled connections and event firings

Computing derived values - in mapStateToProps (great place for it)

Challenges with re-rendering due to connect methods

Forms and Redux (redux)

Chapter 9

ReactJS and Ajax with Axios

ReactJS Fundamentals

Topics

- React and AJAX
- Survey of AJAX APIs

React and Ajax

- As we've seen, React doesn't proscribe anything external to itself
- There are tons of APIs out there
- Good [survey here][http://andrewhfarmer.com/ajax-libraries/]

Easy option - jQuery

- But it's large
- You could do a custom build with just Ajax, but that's ugly
- So not necessarily the default choice

XMLHttpRequest

- Are you a browser warrior?
- Do you have time to write your wrapper for this?
- Would you like to waste a few sprints?

Axios

- An AJAX library that is promise-based
- Similar in features to jQuery Ajax, other common ones
- Smaller community of committers, relatively new

fetch()

- The new standard AJAX API that will be native to all browers, currently in Chrome and FireFox
- Easy to work with
- Supports `cors` exclusion to fetch from other sites
- But needs polyfill to work with other browsers

superagent

- Works on all browsers and Node
- Has lots of plugins
- Nice DSL for providing query parameters, setting headers, etc.

What to use?

- Up to you
- Have `jquery` expertise on staff and don't mind size? Use that.
- Try `axios` or `superagent` for more sophisticated projects
- Consider `fetch()` but realize you may have browser-specific issues
- We'll use axios for this chapter

Axios in an ES6 project

```
$ npm install --save axios
```

```
import * as axios from 'axios';

...
```

```
axios.get('http://my.url.com')
  .then(function(response) {
      // response.data - the payload
  });
```

- Uses typical promise processes - with `response` that contains data, header, etc...

Using in component

```
export class MyComponent extends React.Component {
  componentDidMount() {
     axios.get('/api/v1/customer/1')
      .then((response) => {
         this.setState({ customer: response.data });
      });
  }

  render() {
    return (<div>First: {this.state.customer.firstName}</div>);
  }
}
```

Proper error handling

- Axios calls should inlcude `catch` at the end
- Helps to deal with failed AJAX calls
- A bit different than typical promises and their error methods

```
axios.get('xxxx')
  .then((results) => { ... })
  .catch((error) => { // report and handle error });
```

Config methods

- You can pass configuration settings to calls as the second (or third) parameter

```
axios.get('xxxx', { timeout: 10000, withCredentials: true })
.then....
```

Setting axios default config

```
import * as axios from 'axios';
axios.defaults.config['timeout'] = 10000;
```

AJAX and update strategies

- Make call in proper method
- An event your user triggers like `onClick`
- Loading a component with `componentDidMount`
- In a timed event
- Upon promise resolve you can
- Trigger state change with `setState`
- Call method in props to tell parent component to change state
- Determine not to update the UI at all
- Up to you

Using AJAX with Redux

- Todo, figure out best practices here
- How to square with Redux - when do you fetch ajax and put in redux etc?
- Do the fetch in the Action creator etc...
- How to initialize, etc...

Recap

TODO

Async Techniques in React with Redux

React Fundamentals Training

Topics

- Synchronous actions recap
- Can we just use a Promise?
- The need for async actions
- Two primary techniques
- The `thunk` middleware API
- Writing Sagas

Synchronous Actions...

1. Your event calls the action creator, passing it the necessary parameters
2. The action creator does the work to prepare the request to Redux
3. Once created, you pass the action to Redux
4. Redux processes the action request via its store reducer(s)
5. Redux notifies your store subscribers
6. Control is returned to you

This is good, but...

- What if you have async work to do (Ajax, timers, websocket message processes)?
- You could always create a promise and handle it manually, handing off the request when it is done
- But this is a one-off approach
- Redux is prepared for these kinds of requests

Dealing with Promises

- Redux has a middleware for promise resolution, named `redux-promise`
- Install it, return a promise from your action, and it waits to handle until the promise resolves
- TODO - review syntax and show example

Introducing the `thunk` middleware

- Want more control? Use Thunk.

In a Thunk Action:

1. You call your action creator, which takes whatever parameters you need to satisfy your request
2. Your action creator returns a method accepting a single parameter (`dispatch`)
3. In that method, you execute your logic using promises or other async methods, and when you are finished, you call the dispatch method, passing it your payload

Installing the Thunk middleware

```
import {createStore, applyMiddleware} from 'redux';
import thunk from 'redux-thunk';
import videoReducer from '../reducers/video-reducer';

let initializeStore = (initialState) => {
    const store = createStore(
        videoReducer,
        initialState,
        applyMiddleware(thunk)
    );

    return store;

};
```

Thunk Action example:

```
let loadVideos = () => {
    function loadVideosAction(data) {
        return {
            type: LOAD_VIDEOS,
            items: data
        };
    }
    return function(dispatch) {
        return axios.get('http://vimeo.com/api/v2/chariotsolutions/videos.json')
            .then((response) => {
                dispatch(loadVideosAction(response.data));
            });
    };
};
```

Calling the Thunk action

- Call it just like a synchronous action

```
store.dispatch(loadVideos());
```

- Redux will execute the code asynchronously

Coordinating multiple requests

Consider this case for a Weather application:

1. Call an action to get current lat/long from the browser
2. Use the lat/long to call a web service for weather reports

How can we do this?

1. Do both calls in the action creator as a set of chained promises
2. Do each call separately and coordinate in React by creating the weather component conditionally
3. Or use a Saga to provide two actions in a long running story

The Redux Saga library

- Sagas have been around a long time as a concept
- Redux Sagas are simply an implementation of this concept
- Redux Sagas run alongside the store
- Sagas can watch for behavior such as specific Action types
- Sagas can then execute asynchronous and synchronous calls
- Each time the saga may `yield` its effects to the store

Installing Saga middleware

```
const sagaMiddleware = createSagaMiddleware();

let store = createStore(
  combineReducers({
    location: locationReducer,
    forecast: forecastReducer
  }),
  applyMiddleware(
    //all middlewares
    sagaMiddleware,
    ...
  )
);
```

Starting a Saga with `run`

- Sagas need to be started to execute
- Sagas can be started anytime
- Long-running ones (that act as daemons) can be started after initializing the store

  ```
  sagaMiddleware.run(getLocationAndWeather);
  ```

- Now we're ready to define the saga itself

A Location and Weather Saga

Here is the Saga function itself (`getLocationAndWeather`)

```
export function *getLocationAndWeather() {
    while (true) {
        yield take(GET_LOCATION);
        const location = yield call(getLocation);
        yield put({type: SET_LOCATION, ...location});
        const forecast = yield call(getForecast, location);
        yield put({type: SET_FORECAST, report: forecast});
    }
}
```

A closer look - what is the * monicker?

```
export function *getLocationAndWeather()
```

- This is a generator function begins with `*`
- This is an ES2015 feature
- It states that the function may yield answers multiple times

This Saga is a Daemon

```
while(true) {
   yield take(GET_LOCATION);
   ...
```

- It runs forever (not in a tight loop)
- It will pause until a reducer recieves a `GET_LOCATION` action type
- This is a signal for us to begin the process

The Call to our Browser Location API

```
   const location = yield call(getLocation);
```

- Now we need to return a `Promise` from `getLocation`
- The saga sleeps until it gets a resolution from the promise
- We can then send the location to our store...

What Does the `getLocation` Call Look Like?

```
function getLocation() {
    return new Promise((resolve, reject) => {
        let watch = navigator.geolocation.watchPosition(
          function (geoposition) {
            navigator.geolocation.clearWatch(watch);
            resolve({
                coords: geoposition.coords,
                gatheredOn: new Date(geoposition.timestamp)
            });
          });
    });
}
```

- So a regular promise-based method

Sending our results back to the store

```
  yield put({ type: SET_LOCATION, ...location});
```

- We just dispatched an action request to the store
- this runs after the generator yielded to call the `getLocation` method and got an answer from the promise

We destructured the returned location so each property is set in the store separately (not needed, but interesting)

The Next Step - Fetch our Weather Based on our Location

```
        const forecast = yield call(getForecast, location);
        yield put({type: SET_FORECAST, report: forecast});
}
```

- We cheated and jumped ahead...
- but we know everything we need to do here...

The getForecast function

```
function getForecast(location) {
    return new Promise((resolve, reject) => {
      ...
      // my API key
      let request = new Request(`/forecast/apikeyhere/` +
          `${location.coords.latitude},${location.coords.longitude}`);

      fetch(request, init)
      .then((data) => {
        return data.json();
      })
      .then((jsonData) => {
        // THIS IS WHERE WE END IT ALL!
        resolve(jsonData);
      });
    });
}
```

The getLocationAndWeather Saga again

```
export function *getLocationAndWeather() {
    while (true) {
        yield take(GET_LOCATION);
        const location = yield call(getLocation);
        yield put({type: SET_LOCATION, ...location});
        const forecast = yield call(getForecast, location);
        yield put({type: SET_FORECAST, report: forecast});
    }
}
```

Chapter 10

React and Forms

ReactJS Fundamentals

TODO

Chapter 11

Routing with react-router

ReactJS Fundamentals

Views and Single Page Applications

- SPAs switch views on the fly within their page
- This speeds up UI
- Provides different use case displays
- Typically done by watching the hash at the end of an URL

```
http://localhost:8080/myApp  (consider this '#')
http://localhost:8080/myApp#/main
http://localhost:8080/myApp#/messages
```

None of the # changes cause a page reload

Watching the Hash

- Add an event listener to the window

```
window.addEventListener('hashchange', () => {

  console.log('hash changed to',
    window.location.hash.substr(1));

});
```

You Could Write Your Own Routing

- Just watch this in a component
- React to change by switching view content with a render

Why not do this? * Doesn't deal with * History (go back, forward) * Parameters (inline in URL, query string) * Complex matching rules

Enter the React Router

- Provides comprehensive routing services
- Several ways to generate URIs
- Deals with history using the `history` API
- Handles path parameters
- Handles query string parameters
- Uses a `<Router>` wrapper component to install within a component

Configuring the Router

- Create root component (your main 'view')
- Provide links to other routed components
- Create router wrapper component
- Render root router wrapper component
- Create other routed-to components

Later... * Integrate with other middleware (Redux, especially)

Router HelloWorld - Root Component

```
import { Link } from 'react-router'

class MainComponent extends React.Component {
  render() {
    return (
      <div>
      <Link to="/hello">Hello!</a> |
```

```
      <Link to="/world">World!</a>
      { this.props.children }
      </div>
    );
  }
}
```

- `/hello` and `/world` will be prepended with `#`
- `{this.props.children}` renders inner content

The Router Installation

```
import { Router, Route,
         IndexRoute, hashHistory} from 'react-router';

class MyRouterRoot extends React.Component {
  render() {
    <Router history={hashHistory}>
      <Route path="/" component={MainComponent} >
        <IndexRoute component={Hello} />
        <Route path="hello" component={Hello} />
        <Route path="world" component={World} />
      </Route>
    </Router>
  } ...
```

- `IndexRoute` is the default route
- the `history` parameter selects the URI strategy

Factor Out your Routes to Another File

- Extract your route instructions
- Makes your root component easier to view

```
...
import myRoutes from './my-routes';

class MyRouterRoot extends React.Component {
  render() {
    <Router history={hashHistory} routes={myRoutes} />
  } ...
```

The Routes File Looks Like This

```
export default (
    <Route path="/" component={MainComponent} >
        <IndexRoute component={Hello} />
        <Route path="hello" component={Hello} />
        <Route path="world" component={World} />
    </Route>);
```

Our Routed Components

```
// Hello.js
let Hello = (params) => {
  return <div>Hello!!</div>;
};

export default Hello;

// World.js
let World = (params) => {
  return <div>World!!</div>;
};

export default World;
```

Routing Instructions

Inline Parameters

- Variables such as an ID in an URI path (`/customer/424`)

Route Instruction: use `:name` as a placeholder:

```
<Route path="customer/:id" component={CustomerDetails} />
```

Acquiring route data - use `props`

```
...
componentDidMount() {
  let id = this.props.routeParams.id;
  ...
}
```

Wildcards in Routing

Wildcard	Effect
`*`	Match all characters (non greedy) up to next /, ? or #
`**`	Match all characters (greedy) up to next /, ? or #
`()`	Match, but optional
`:param`	match param up to next /, ? or # (segment)

Matching is done from top to bottom in order of definition

Matching Examples

Route Pattern
/foo
/foo/:id
/foo/*/:id | /foo/bar/23 | | /foo/(:bar) | /foo/, /foo/234 | | /foo/** | /foo/, /foo/234, /foo/a/b/c | | /foo

Note - `**` provides a `splat` in `routeParams` with all values

Nested Routes

- You can make a multi-level route - with parent and child both owning components

```
<Route path="customer/:id" component={Customer}>
   <Route path="manager" component={CustomerManagerUI}>
     <Route path="orders" component={OrderList}/>
   </Route>
</Route>
```

- A valid route is `/customer/12/manager/orders`
- Three components are created

Suppressing a Path for a Route

- You can eliminate part of the URL path in a nested route, just don't use the `path` parameter

```
<Route path="customer/:id" component={Customer}>
   <Route component={CustomerManagerUI}>
     <Route 'orders' component={OrderList}/>
   </Route>
</Route>
```

- Now `/customer/23/orders` is used to mount all three components
- Known as a pathless route

Challenges with the Router

- Challenge: routed-to components cannot get props or state from parents, just from Router
- This is where Redux can come into play (more later)

Route Precedence

- Routes are evaluated in the order defined
- Put greedy routes last

Router History APIs - `hashHistory`

```
import { Router, hashHistory } from 'react-router';
...
<Router history={hashHistory}>...
...
```

- We've seen the `hashHistory` approach, uses a `hash` code at the end of the URL
- OK if you can't get HTML5 history support in your browser
- Unlikely ever to be used in production

Router History APIs - `browserHistory`

- The better choice for HTML5 browsers
- Defers to the HTML5 history API
- writes sane URLs
- no hash required
- support server-side rendering

- BUT: requires all server URLs to redirect to your entry URL

See this guide for configuration tips

Router History APIs - `createMemoryHistory`

TBD

Using the History API directly

- A component can call the history API to route or go back to a prior route

```
import { browserHistory } from 'react-router';
...
browserHistory.push('/accounts/234');
browserHistory.goBack();
```

Routing and Lifecycle

- Components being routed *to* will get `componentDidMount`
- Components that are being routed *from* will get `componentWillUnmount`
- Components that are part of the route but *not being navigated away from* will get `componentDidReceiveProps` and `componentDidUpdate`
- Components active but that get changed props (ex: from `customer/12` to `customer/33`) will get `componentDidReceiveProps` and `componentDidUpdate`

Accessing the router API

Just wrap your component with the `withRouter` method

```
...
import { withRouter } from 'react-router';

class MyComponent extends React.Component {
  componentDidMount() {
    // have access to router API via this.props.router
    // if you use withRouter to wrap this component
  }
```

```
}

export default withRouter(MyComponent);
```

Router APIs

call	usage
`goBack()`	move backward one step in the route history
`goForward()`	move forward one step in route history
`go(n)`	move n steps forward or backward in route history

More Router APIs

call	usage
`createLocation(uri)`	builds a location object from the information in `uri`
`push(path)`	push a route request to history
`replace(path)`	route but overwrite existing history step
`setRouteLeaveHook`	allow user to cancel route

Router, Props and Redux

- The router takes over your routed component
- *No props* can be passed from above the Router
- You can use `Redux` for shared state with `connect`
- You can also use `Context` but that's not recommended
- Let Router control your UI flow, and Redux control data state
- Don't fight it... Use Redux instead of attempting to pass props

Redux and Router

- You will have two sources of data - the router, and redux

A Router with Redux

```
<Provider store={providerStore}>
  <Router history={browserHistory} routes={routes} />
```

```
</Provider>
```

Router and Redux - Distribution of Duties

- Router knows route state
- Redux knows application state
- You may
- Pass router data into actions to make decisions
- Navigate to routes

How to Integrate with Redux

- Basic integration with `connect`
- Alternate libraries for Redux integration
- `react-redux-router`
- `redux-router`
- Integrating with the Redux library
- react-redux-router
- `withRouter` function