

React and Redux Bootcamp:

Sean al-Baroudi

June 6, 2022

First React Application:

- To start a React Project, most developers will utilize the create-react-app project template (a node package). Simply install this package and run it to start. Specifically:

```
npx create-react-app <project name>
mkdir <project name>
npm start
```

to install and get the local server running.

- **Project Structure:**

- **./node_modules:** All locally installed node dependences are stored here.
- **./public:** index.html (front page, robots.txt and favicons stored here).
- **./src:** Contains the React Apps (self-contained pieces of interface+code), and support files, including:
 - App.js (contains template React App) - .test files (to run tests later) - index.js: Where we Insert our App DOM object into the index.html DOM tree (at root).
- The following minimal code is needed to insert our content into the index.html DOM tree:

```
import React from 'react';
import ReactDOM from 'react-dom';
ReactDOM.render(<div>Our React Element </div>,
document.getElementById("root"));
```

- In react, we typically bundle reusable UI elements into **Components**.
- **Components:** reusable pieces of React code to control part of the user interface. Components capture the structure of UI, and can have internal data to track the user behavior throughout the lifetime of the app.

```
import { Component } from 'react';
```

- Note that Classes, extended Sub-Classes and object Instances are used heavily in React - so be familiar with OOP.
- A typical design pattern in React will be to extend template component classes, and add additional fields/methods to implement your application.
- React automatically detects saved changes, and the local server updates. So you can update accordingly.
- **State Field:** Subclasses of the Components class have a state field that we can use. We can link an object to this field, to maintain current state as the Component runs.
- **State:** Dynamic data in a React component. This is often used to track variables that will be re-rendered in the UI based on events that occur in the application.
- **React Rule: Never** Directly Modify the State property, if your intention is for changes to occur in the render method based on the updated state values. Use `setState()` instead.
- **But Why??** React is *designed to re-render()* a given component when the `setState()` function is called. If you modify state directly, the changes will **not** be reflected in browser.
- **Props:** Just stands for "Property". Data in a React component that gets passed down from its parent. In the parent component, it will pass data down to the child component through attributes in the child component's JSX.
- **How does the passing work?** See the example below:

```
class Profile extends Component {
  const {img, link} = this.props.profile

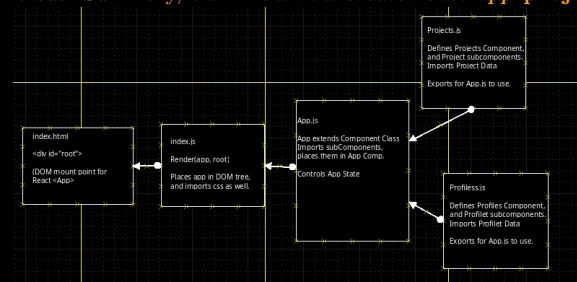
  ...

}
```

```
class SocialProfiles extends Component {
  render() {
    ...
    SOCIALPROFILES.map( prof => {
      return )
    <Profile key={prof.id} profile={prof} />
    ...
  }
}
```

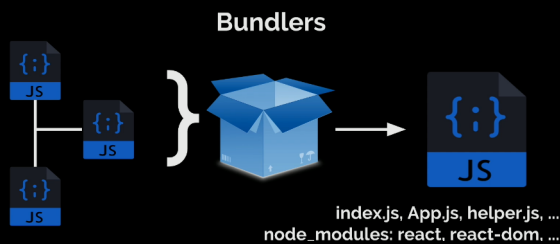
Note that the SOCIALPROFILES data object gives us our data for this example. We map() on each element of the object. Here, our Profile class defines a component that is a child to the SocialProfiles Class. We instantiate an object of the Profile class using the <Profile tag>. In the tag, we have placed properties (id, profile) that are fed into the object instance of Profile. We then can access these properties with `this.props.profile...`

- Visual Summary, our modified **create-react-app** project:



Breaking Down React: React and Web Development:

- Much of the structure in **create-react-app** is not necessary to get a minimal example to run.
- All an app needs is one index.html page, and index.js page.
- You don't even need to run "npm start" or "node react" - you can use CDNs to load React and ReactDOM libraries straight into the browser.
- **The Minimal Example is as follows:**
 - React CDN script tags at the end of index.html
 - A div root tag in index.html, so React can insert our <App> into the DOM tree.
- In index.js you can just build an app, render() and insert it as normal.
- **JSX:** This is syntactic sugar that is transpiled to code that the browser will understand, when we run our react project (with npm start).
- Transpilation is typically done with the **Babel package**.
- **Why use Babel?** Javascript compliance is entirely Browser dependent - and JS core language development is completely separate from Browser development. When we want new features that Browsers may not yet support - we can transpile code to cruder JS so that it can be used in current browsers.
- **Bundling and Transpiling:** Note that our React project will run in the browser, but ultimately Chrome/FF will not support our import/export system, nor will it support JSX. So we use a **bundler package (Parcel)** that utilizes Babel to produce a low-level enough JS file so that Chrome/FF can run our app.
- Parcel traces our application code, and finds all possible dependencies, and then places all imported code into one mega-js file. Any syntactic sugar (such as JSX) is transpiled into lower level JS code that a browser can understand.



- **Why Use Parcel?** The alternative is to import multiple scripts into the index.html page. These are loaded in order. As our project increases in size+complexity we may deal with loading issues (namespace clobbering, circular dependencies, etc). To avoid this, Parcel will make one large JS file where dependency/NS issues are alleviated.

- **Basic Parcel Usage:**

```
parcel <index.html>
```

It will trace back all code dependencies from here.

- **Parcel Folders (in project):** These will appear after you run parcel the first time:
- **.cache:** Stores compiled elements to minimize recompilation later.
- **dist:** Folder where bundled+transpiled JS code is stored - will rewrite index.html script links to point to this directory!
- Using Parcel for Dev and Production: Parcel has a hot-wire feature - we can run it with the index.html, and it will auto-update all our code as we develop it.
Parcel however is heavier-weight - we don't use it when we launch production code. The following package.json scripts would be seen for a parcel build:

```
"scripts": {
  "dev": "parcel src/index.html",
  "build": "parcel build src/index.html",
  "start": "npm run build && live-server dist"
}
```

A lighter server (live-server) is chosen to strip down the size of the compiled production build, for launch.

- **VirtualDOM:** React has its own stripped-down DOM model, so that it can make dynamic changes to the application page. Note: You don't use native DOM methods (like createElement, getElementById(...)) in React - it has its own framework calls to do manipulations.

Continued Main React Concepts:

- **Component Lifecycle:** Components have various event methods, that can be triggered during their operation. In particular, we are interested in when the component starts up, and when it shuts down with two methods:
- **componentDidMount():** Called after component is loaded. If the method has to perform setup work or a fetch(), write this code in here.
- **ComponentWillUnmount():** Called just before a Component is unmounted and destroyed. Use this as a takedown/cleanup/signalling method.
- Both methods above are written in the Component extended class - use non-first class functions only. "function" keyword is not required (it's a JS class!).
- **Stateless Functional Components:** Simplified Notation can be used, if our components contain no state. See Example Below:

```
const Project = props => {
  //destructure
  const { title, image, description, link } = props.project;

  return (
    <div style={{}>
      <h3> {title} </h3>
      <img src={image} alt='profile' />
      <p>{description}</p>
      <a href={link}>{link}</a>
    </div>
  )
}
```

Notice that for static/one-time components, we don't even need the render() method. We can even skip the return and encase the functional body with () - however we did destructuring above, so in this case it cannot be made that simple. Most simple example below:

```
const SimpleComp = props => {
  <div>   ~SOME JSX~   </div>
}
```

- **fetch():** Default Node/React method for making HTTP calls to external sources.
- fetch() returns a JS Promise - and is an asynchronous function.
- **Fetch Example 1:** Get a JSON object and print it:

```
fetch("http://localhost:3005/random_joke")
  .then(response => console.log(response.json()))
  .catch(error => console.log("Error:", error.message));
```

Will print out json object to console. Basic errors also handled.

- Example 2: Extract an array of JSON objects from promise, and work with them:

```
...
state = {...jokes: []};

fetchJokes = () => {
  fetch("http://localhost:3005/random_ten")
    .then(response => response.json())
    .then(json => this.setState({jokes:json}))
}

...

{ //At site where we want to print objects...
  this.state.jokes.map(joke => (
    <Joke key={joke.id} joke={joke}/>
  ))
}
```

We use the state and setState to pull the jokes array.

- **Higher-order Components:** A higher-order component is one that takes another component as an input. The idea is to treat the component like a function. Have one component as input, and then output an entirely new component based on that input, with new properties, methods, and/or JSX!
-

Music Master 2.0

- API service was not available for this project, I just found my own data source instead and did my best. Some useful coding patterns/concepts are below:
- **Arrow callbacks - Returning the Callback itself** (instead of return() value):

```
myfunction => props => () => { return(...)}
```

This will return the inner function, with access to the outer parameters. In other words, equivalent to:

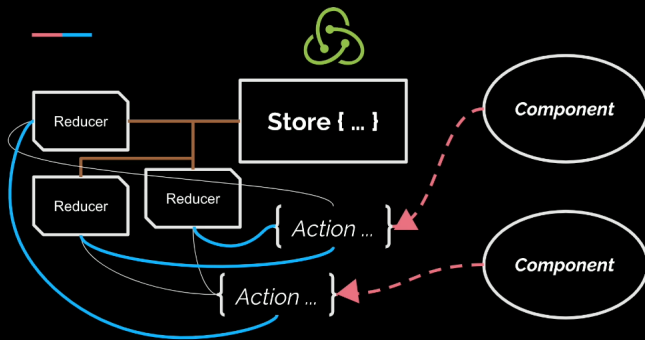
```
function (props) {
  return function() { body... }
}
```

-

Redux and more Advanced React:

- Redux is a State Management technology.
- It handles all the underlying data for your React application by enforcing a universal flow of data - that components can read from and add to using dispatchable actions.
- It passes state/data between different components in our Application.
- Parent → Child data-passing can be prohibitive in complex situations - so we use Redux for a cleaner solution.
- **Main Parts of Redux:**
- 1) **(Global) Store:** Collects the state of the entire application.
- 2) **Reducers:** Reducers split up the store and describe how sections of the store should update.

- **Actions:** Objects (emitted events) full of data, processed by Reducers.
- 4) **Dispatchers:** Components that create action objects.
- This model creates a **Unidirectional Flow** of data.



- Actions are objects full of data that the reducers listen to. These actions are dispatchable by components. And reducers will then use the action data to update the redux store.
- All reducers listen for actions (events that contain data). Each one fires, and checks the type of the action - to see if it handles it.
- It is the responsibility of the programmer to check/filter types correctly in each reducer.
- Actions are typed objects: they must contain a **type** field - this is how reducers filter them. An example of an action:

```
export const startGame = () => {
  return {type:SET_GAME_STARTED, gameStarted:true};
}
```

- Redux and React are two separate frameworks - they must be integrated together in a project. This specifically means:
- React Redux provides the connection between the redux system and the React components. Particularly:
 - The Provider component wraps the App (in index.html) gives each component access to the Redux store.
 - The connect() function allows each component to customize what parts of the redux store it wants to read, and what actions the component should be able to dispatch.
- 1) Components in React act as dispatchers for actions (Redux)
- Components in React will query the Redux Global Store, to interpret application state.
- **Redux Structuring:** The following folders and files are used, for each of the key objects in Redux:
 - 1) Actions: Stored in an ./src/actions folder.

There is a settings.js file where all arrow function actions are stored, and exported.

There is a types.js file, which lists all of our action types and exports them (so Reducers can use the types). An example of a type constant:

```
export const SET_GAME_STARTED = "SET_GAME_STARTED";
```

- 2) Reducers: Placed in the ./src/reducers folder, in a file called index.js. These are exported as well. An example of a reducer:

```
const rootReducer = (state = DEFAULT_SETTINGS, action) => {
  switch(action.type) {
    case SET_GAME_STARTED:
      return {
        gameStarted:action.gameStarted,
        instructExp:state.instruct
      };
    case SET_INSTRUCTIONS_EXPANDED:
      return { ...state, instructExp:action.instructExp };
    default:
      return state;
  }
}
```

- 3) Dispatching Setup: This is typically done in App.js. See the following code:

```
import React, { Component } from 'react';
import { connect } from 'react-redux';
```

```
class App extends Component {
  render() {
    return (
      <div>React App</div>
    );
  }
}
```

```
const componentConnector = connect();
```

```
export default App;
```

The connect() function connects the App component to Redux, so it can be used as a dispatcher.

- **MapStateToProps:** The first argument to the connect function. Allows you to specify what data the component should have from the redux store. You map the redux store state to the props object of the component.
- **MapDispatchToProps:** The second argument to the connect function. Allows you to specify what methods should be made available to the component (through props), that will be able to dispatch the results of action creators to the store.
- 4) Setting up the Provider Frame: Note that to use global state, we must wrap our App in another structure called a Provider. We do this in the project index.js file:

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

```
const store = createStore(rootReducer);
```

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

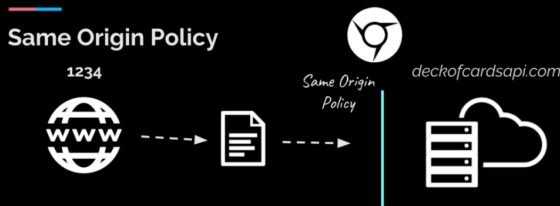
- Reducer Debug Tip: Instead of internal console.log() statements, use the subscribe() function:

```
store.subscribe(() => console.log("store.getState()",store.getState()));
```

- Redux Middleware code runs in between the point that an action gets dispatched and the reducers receive the action objects. Think of it as extensions for the redux system; it's useful for adding new capabilities to the underlying redux logic. One such middleware is redux-thunk.
- Redux **thunk** is a library that allows for asynchronous actions to be dispatched. Normally, actions are plain objects. But in the case of using the fetch method, to make an api call, an asynchronous action is useful in order to dispatch different kinds of actions based on the result of the api call.
- **LocalStorage** is a JavaScript object that can be used to store items in a key-value structure within the browser.
- **Dealing with Cross Origin Resource Sharing:** When making a fetch() request to a remote server, the request will often fail because of the CORS policy.
- **What is CORS?**
- **Definitions (for CORS):**
 - origin: the server the application is running from.
 - (an) origin: the source of a particular resource.
 - permitted origin: a sourced allowed by the browser to serve content - it lives outside the application itself.
 - cross-origin: when the provide and destination of a resource do not match.
- **Simple Request:** One defined as a GET/HEAD/POST method with safe header fields. See [1] for details.
- **Same Origin Policy (Summary):** A web application using XMLHttpRequest and Fetch() can only request resources from the

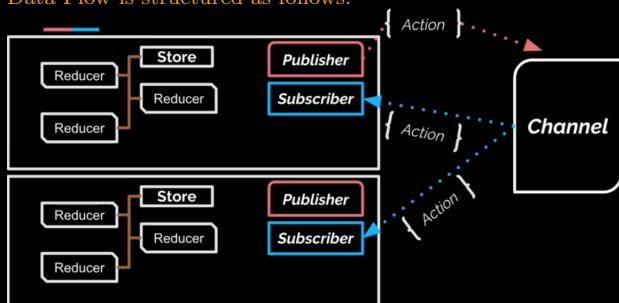
same origin the application was loaded from unless the response from other origins includes the **right CORS headers**.

- CORS only applies to HTTP requests to external sources, from scripts. This is to suppress unwanted requests from hijacked applications.
- **Different Header Options:**
- **External Server:** the Header option for CORS is *Access-Control-Allow-Origin* - this is set in the response object from the external source. If set to *"*"*, any application can request resources from the server. For better security settings - a resource provider will have a short-list of approved applications.
- **App Origin:** For a non-simple request (not a simple GET/POST...), a preflight request is sent to the external server by the browser. This consists of an OPTIONS request with the *Access-Control-Request-Method* and *Access-Control-Request-Headers* fields set. Once the response is sent back and is approved by browser, the non-simple request is sent out.



Redux Project: Reaction:

- For this application, we will implement the Publisher-Subscriber Architecture for our app, to allow subscribers to a channel to post messages, and reply with Emojis's.
- Pub/Sub: A software pattern that allows publishers to send information to channels that subscribers will listen to. A good networking option that can reduce the usual approach of keeping a list of all addresses that are connected to the network.
- We will have multiple instances of the application running. Redux Data Flow is structured as follows:



- The steps our application will take, to send a message, are as follows:

Pub/Sub with Redux

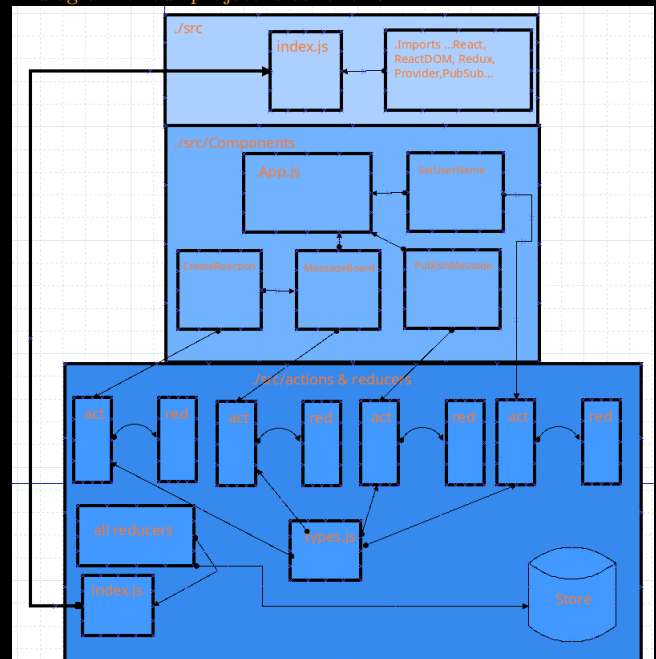
1. Create an action object
2. Publish the action object to a channel
3. App subscribers pick up the published action
4. Each subscriber dispatches the action to the local redux system
5. The local redux system picks up the data, and the app updates!

lows:

- **React Context:** A way to share data between many React components without having to pass that data down manually through props. A component will look to the nearest parent component, that is a context Provider, to get the values that it can use for its own component 'context' field.
- The Context Provider Component uses a value property to provide data to any nested component through its context field.
- The Redux devtools is an extension that allows you to examine the redux store state, and see actions being dispatched in real time.
- Connecting to React DevTools: Download app in Chromium, and use the following createStore() arguments:

```
const store = createStore(rootReducer, window.__REDUX_DEVTOOLS_EXTENSION__ && window.__REDUX_DEVTOOLS_EXTENSION__());
```

- A diagram of our project is as follows:



Create Backend APIs for React Apps:

- Express is a framework built on top of Node.js used for starting servers and building web APIs.
- Express middleware is functionality that runs before every request, to process information about the request, add headers to the response, perform logging, and any other kind of useful functionality. Code Examples Below:

```
app.use((req, res, next) => {
  res.header("Access-Control-Allow-Origin", "*");
  next();
});
```

These middleware modules act as listeners that are fired before GET() listeners are engaged. the Next() method allows us to continue processing in other middleware listeners. The RES() object allows us to respond to the client prematurely.

- Our express can serve requests with the get() method:

```
app.get('/ping', (req, res) => {
  res.send('pong')
});

app.get("/stories", (req, res) => {
  res.json(stories);
});
```

Note: Avoid appending paths with "." (Linux Habit). Error 404 will occur!

- **CORS policy for API:** Adding the 'Access-Control-Allow-Origin' with the "*" argument will allow any frontend addresses to be considered as part of the server's origin. This helps get past the same origin policy check implemented by the browser.
- API Wrappers are servers that perform inner requests to other APIs. These can be used to combine multiple API requests into one overall request, to add a more open 'Access-Control-Allow-Origin' header, and to accomplish other custom functionality!
- Serving a React Frontend can occur within the same server as its backend API. Serve a static directory to the relative path where your React code is built.
- To do this, we merge our client and api projects. This involves splicing the package.json file of the client project into the package.json of our express project. See git directory for details. Run npm i to install all packages in the root directory.

- We can get our API to serve front end files with the following code:

```
const path = require("path");

...

app.use(express.static(path.join(
  __dirname, "client/dist")));
```

- **Why does this work?**

Because the index.html file of the react app has been put in client/dist by parcel running in the background. Note our modified npm start script:

```
"start": "npm run build-client && node ./",
```

As long as the paths don't clobber, we can access React content and our API with the same localhost:3000 domain!

- **Caveat:** If we do this, dynamic page loading is disabled. If we want to develop our API and React App, and see the front end update on saves, we can run:

```
Tab1: npm run dev
Tab2: npm run dev-client
```

•

Deployment of our App (Heroku):

- **Steps for launching a Heroku App:**
- 000) Sign up for Heroku, setup 2FA, install on your desktop.
- 00) Run `*heroku login*` on command line, and connect.
- 0) Make sure this project is uploaded to its own github repository before you start.
- 1) Now Adjustments to our App need to be done:
- a) Adjust the `fetch()` URL from `"localhost:3000..."` to `'{document.location.origin}'`.
- b) Add an environment generated port number to index.js: with `const PORT = process.env.PORT || 3001;`
- 2) Once changes are complete, commit changes to github repository one more time.
- 3) Run `git push heroku main` to run the application in the cloud.
- 4) (Optional): See the Heroku setting panel to adjust port/url/mapping etc. Remember to update the github repo link if you make this change with: `git remote set-url <new heroku url>`
-

References

- [1] <https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS>