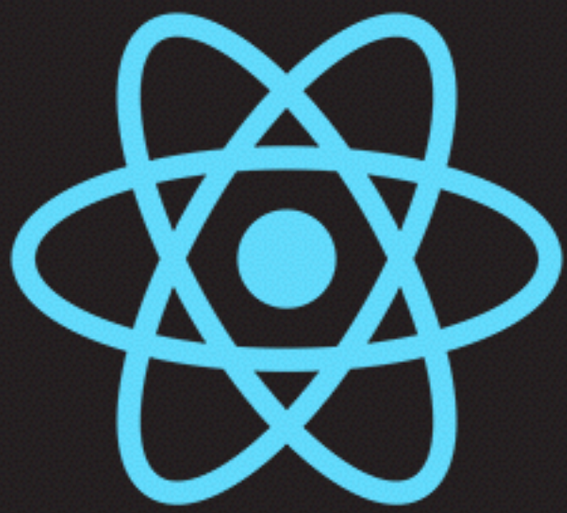
COLLEGIO TECHNOLOGIES INC.

REDUX



WHAT IS REDUX?

- Redux, in essence, is a JavaScript Library that allows us to track changing state data.

- Redux is not React, but rather is another library that integrates really well with it.

- Redux is also known as a ***State Management Library***.

WHY USE REDUX WHEN WE ALREADY HAVE STATE AVAILABLE IN COMPONENTS?

- Although using state and props in the fashion that we have been using it is perfectly fine, there are some scenarios where this will not work or be the optimal solution:
  - Apps which have more than one parent component, such as what we have seen using react-router, do not have a way to directly share state with one another.
  - In addition, having to pass down props for several components can specify a component a little too much, making it less re-usable.
- Having a centralized state management system such as Redux allows components to simply request what the needs from the state, and what it needs to be able to change on the state.

HOW DOES REDUX WORK?

- Redux is a state container, which is similar to our class-based components.
- A ***redux store*** is an object that will be set up and accessed in a similar fashion to component state objects.
  - Components can declare which stores they need to access and change.
  - By communicating with the store instead of via a long chain of props, components become much more reusable inside of the project.

SETTING UP REDUX

- Redux is installed via Yarn: **yarn add redux**
- Once it is installed, you will need to import the Redux methods that you'll be using into your file.
- You can create a new store by importing **createStore**
  - **import { createStore } from 'redux';**
    - The createStore method accepts a function as it's argument, which is passed the current state. The current state will be empty in this case, so it's up to you to set some initial values here.
- You can see the data inside of a store by using the **getState()** method.

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

const store = createStore((state = { curNumber: 0 }) => {
    return state;
});

console.log(store.getState());      // will print the state declared above
```

WORKING WITH REDUX

- Now that we know how to create our Redux store and initialize it's data, how do we actually update this data?
  - We can perform updates on our data via ***actions***
- Redux actions are objects that are sent to the store containing the type of action we want to take.
  - We send actions to the store by using the **dispatch** method, which accepts an object containing the information that you want to send.
  - Once an action is sent to the store, the passed data can then be accessed via a second passed argument to the **createStore** arrow function.

```
const store = createStore((state = { curNumber: 0 }, action) => {

    switch (action.type) {
        case 'ADD_ONE':
            return {
                curNumber: state.curNumber + 1
            }

        default:
            return state;
    }
});

store.dispatch({
    type: 'ADD_ONE'
});

console.log(store.getState());      // curNumber will be 1
```

WORKING WITH REDUX

- Whenever you call the **dispatch()** method, you always have to pass an object containing a **type** attribute, or else the app will crash.
  - However, you can also pass additional data into the object passed into **dispatch()**:

```
const store = createStore((state = { curNumber: 0 }, action) => {

    switch (action.type) {
        case 'ADD_ONE':
            return {
                curNumber: state.curNumber + 1
            }

        case 'ADD':
            return {
                curNumber: state.curNumber + action.amount
            }

        default:
            return state;
    }
});

store.dispatch({
    type: 'ADD',
    amount: 4
});
```

SUBSCRIBING TO REDUX DATA

- Now that we know how to change the data via Redux, how can we watch for specific changes in the data?
  - We do this via the **subscribe()** store method.

- The store subscribe() function is passed a function, that is executed each time a store is updated.

- Let's see how we can just output to console whenever a change is detected:

```
store.subscribe(() => {
    console.log(store.getState());
});

store.dispatch({
    type: 'ADD',
    amount: 4
});          // running this will print the amount because of subscribe()

store.dispatch({
    type: 'ADD_ONE'
});          // running this will also print this amount
```

SUBSCRIBING TO REDUX DATA

- You can **unsubscribe** from a store by passing the initial subscribe method call into a variable, and then calling it when you want to unsubscribe.

```
const unsubscribe = store.subscribe(() => {
    console.log(store.getState());
});

store.dispatch({
    type: 'ADD_ONE'
});

unsubscribe();
```

REQUIRING A TYPE OF DATA WITH REDUX DISPATCH

- It is also good practice to ensure that a specific type of data passed into your dispatch call is correct, and this can be done when you are handling the dispatch action.
  - Use the JavaScript **typeof** keyword to do this.

```
const store = createStore((state = { curNumber: 0 }, action) => {

    switch (action.type) {
        case 'ADD_ONE':
            return {
                curNumber: state.curNumber + 1
            }

        case 'ADD':
            const addAmount = typeof action.amount === 'number' ? action.amount : 0;
            return {
                curNumber: state.curNumber + addAmount
            }

        default:
            return state;
    }
});
```

REDUX ACTION GENERATORS

- In our previous examples, we were creating the object each time we dispatched an action in Redux. Although this will work, it has drawbacks:
  - Typos are easy to make when adding your action type, and very hard to find because they crash silently.
  - There is overall a lot more repetition that you need!

- **Action Generators** are what we'll use to get around this, which are functions that return action objects.

```
const incNum = () => ({
    type: 'ADD_ONE'
});

store.subscribe(() => {
    console.log(store.getState());
});

store.dispatch(incNum());   // will output 1
```

REDUX ACTION GENERATORS

- If we wanted the ability to add some optional data to our Action Generator functions, we should include it as a passed object. The passed object is typically known as the **payload** of the action.
- You can use **ES6 Object Destructuring** to further simplify the function below!

```
const addNum = (payload = {}) => ({
    type: 'ADD',
    amount: typeof payload.amount === 'number' ? payload.amount : 1
});

store.subscribe(() => {
    console.log(store.getState());
});

store.dispatch(addNum({amount: 9}));   // will output 9
store.dispatch(addNum());   // will output 10
```

REDUX ACTION GENERATORS

```
// NOW WITH ES6 DESTRUCTURING!!

const addNum = ( { amount = 1 } = {} ) => ({
    type: 'ADD',
    amount
});

store.subscribe(() => {
    console.log(store.getState());
});

store.dispatch(addNum({amount: 9}));   // will output 9
store.dispatch(addNum());   // will output 10
```

REDUCERS IN REDUX

- Reducers are the functions that determine what we are actually doing with the system state.
  - In fact, we've already been working with reducers inside of our **createStore()** function.
  - Reducers aren't typically declared directly inside of a createStore() function typically, as typical real-world applications will have multiple reducers.

- Reducers are always ***pure functions***. In addition, reducers should ***never*** change the state or action that is passed into it.
  - If you find yourself changing one of these, take a moment to think about your action, as there is almost always a way to do it without changing the value.

EXAMPLE: A BASIC REDUCER

```
const numReducer = (state = { curNumber: 0 }, action) => {

    switch (action.type) {
        case 'ADD_ONE':
            return {
                curNumber: state.curNumber + 1
            }

        case 'ADD':
            return {
                curNumber: state.curNumber + action.amount
            }

        default:
            return state;
    }
};

const store = createStore(numReducer);
```

REDUCERS IN REDUX

- The **combineReducers()** method is used to combine smaller reducers to be used in a store.
  - It's a good idea to split up your reducers to be specific to a sub-set of functionality. For instance, if you are searching entries in your data, you may want separate reducers for modifying your data, as well as filtering your data.
- Let's look at how we can set up separate reducers with this sample set of state data:
  - **players: [{**

```
    id: 1,
    name: 'Rob Myers',
    type: 'hockey',
    gender: 'Male',
    message: 'I want to have fun'
}],
filters: {
    text: 'hockey',
    skill_level: ‘all',
    sort_by: 'name'
}
```

EXAMPLE: MULTIPLE REDUCERS AND COMBINEREDUCERS()

```
// the default list of free agents, which is empty
const freeAgentDefaultState = [];

const freeAgentReducer = (state = freeAgentDefaultState, action) => {
    switch (action.type) {
        default:
            return state;
    }
};

// the default items that we can possibly filter by
const filtersDefaultState = {
    text: 'all',
    skill_level: 'all',
    sort_by: 'name'
};

const filtersReducer = (state = filtersDefaultState, action) => {
    switch (action.type) {
        default:
            return state;
    }
};

// use combineReducers to specify which data to control which reducers
const store = createStore(
    combineReducers({
        players: freeAgentReducer,
        filters: filtersReducer
    })
);

console.log(store.getState());
```

REDUCERS IN REDUX

- Once you have your reducers declared and used within your stores, whenever an action is dispatched, it will be dispatched to all reducers. However, only reducers looking for that type of action will be affected.

```
const freeAgentReducer = (state = freeAgentDefaultState, action) => {
    switch (action.type) {
        case 'ADD_PLAYER':
            return state.concat(action.player)

        default:
            return state;
    }
};
```

```
store.subscribe(() => {
    console.log(store.getState());
});

const addPlayer = ({ name, message = '', type = 'hockey', skill_level = 'basic', gender = 'Male' } = {}) => ({
    type: 'ADD_PLAYER',
    player: {
        id: uuid(),
        name,
        type,
        gender,
        skill_level,
        message
    }
})

store.dispatch(addPlayer({ name: 'Rob Myers', skill_level: 'intermediate' }));
```

THE ES6 SPREAD OPERATOR

- Since we will be moving onto using pure functions to create repeatable code, when adding new items, we will be making use of the **ES6 Spread Operator**.
  - The Spread Operator can be used to quickly and easily append items onto an array, and can be used in place of **concat()**.

- In addition, Babel has a plugin available that will allow for using the spread operator in objects, which we will install:
  - https://babeljs.io/docs/plugins/transform-object-rest-spread/
  - **yarn add babel-plugin-transform-object-rest-spread**
  - You will also need to add this plugin to your .babelrc file:

```
{
    "presets": [
        "env",
        "react"
    ],
    "plugins": [
        "transform-object-rest-spread"
    ]
}
```

CONCLUSION: REDUX

- We're now ready to work with a centralized state management system by using Redux! By focusing in on it for this lesson, we essentially separated out the data and state manipulation side of our app, and in the next lesson, can focus on actually working with it inside of our React apps.

EXERCISE: REDUX

- Let's put some of our knowledge to work! We'll set up a redux script that will create our store for the Free Agent Tracker.
- Set up reducer and action generators for the following:
  - Adding a player
  - Removing a player
  - Editing a player
  - Changing the filter text
  - Changing the filter type
  - Sorting by name
  - Sorting by skill level
- In addition, set up a **getVisiblePlayers()** function that will get all visible players based on the filter criteria.
- Note: you should set up separate reducers for your player management and player filtering.

RESOURCES

- **Redux:**
  - **https://redux.js.org/**
- **ES6 and JavaScript:**
  - **Destructuring Arrays and Objects -** https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment
  - **The Spread Operator -** https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax
  - **The String includes() method -** https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/includes
  - **The Array sort() method -** https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/sort
- **Misc:**
  - **UUID -** https://www.npmjs.com/package/uuid

DISCLAIMER

- Copyright 2018 Collegio Technologies Inc.
- All content within this presentation has been compiled by Collegio Technologies Inc. All material has been sourced through open source resources and source code and as such conforms to Canadian copyright laws and regulations. All rights reserved.