React:

Week 4 Workshop Presentation

Workshop Agenda

Activity	Estimated Duration
Set Up & Check-In	10 mins
Week 4 Review	50 mins
Assignment	60 mins
Break	15 mins
Assignment	90 mins
Check-Out (Feedback & Wrap-Up)	15 mins

Set up

Along with Zoom, please also be logged into Slack and the learning portal!

Check-in

Check-In

- How was this week? Any challenges or accomplishments?
- Did you understand the Exercises, and were you able to complete them?
- You must complete all Exercises before beginning the Workshop Assignment.

Week 4 Review

Overview

MVC	Redux: Data flow
Flux	React-Redux
Redux	React-Redux: useSelector()
Array.reduce()	React-Redux: useDispatch()
The useReducer hook	Redux Toolkit
Redux: Three Principles	RTK: createSlice()
Redux: Main concepts	RTK: configureStore()

MVC – Model-View-Controller

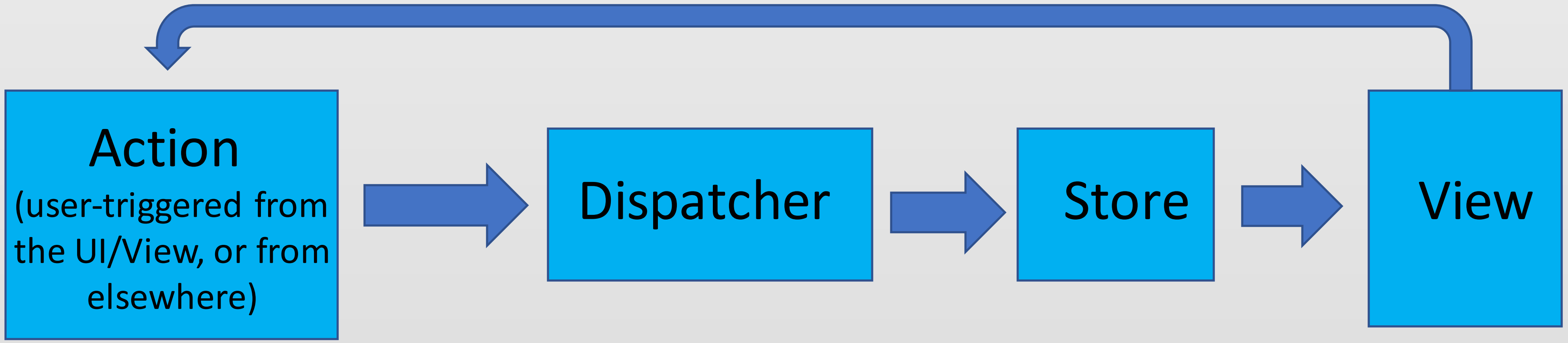
- Software architecture pattern
  - *Not* a library or framework
  - A conceptual approach for writing software
- Key feature of MVC: ***Separation of Concerns*** – independent development, testing & maintenance for each concern:
  - **Model** – manages data/logic
  - **View** – the user interface (UI)
  - **Controller** – handles user input and updates model/view
- Widespread pattern used in both desktop and web app development
  - Thus, while React doesn't use MVC, at least be aware of this pattern
  - First proposed 1978 for designing GUIs
  - Many variants – MVVM (Model View View Model), MVW (Model View Whatever), etc

What the Flux?

- React originally designed as "View" part of MVC
- MVC does not care about data flow direction
- React engineers found that being unopinionated about data flow direction caused issues at scale
- Facebook engineers created Flux pattern to use with React in 2014
- Key feature: ***Unidirectional (one-way) data flow***

Flux

- Flux is "a pattern for managing data flow in your application"
- Key parts: Actions, Dispatcher, Store, View



Redux

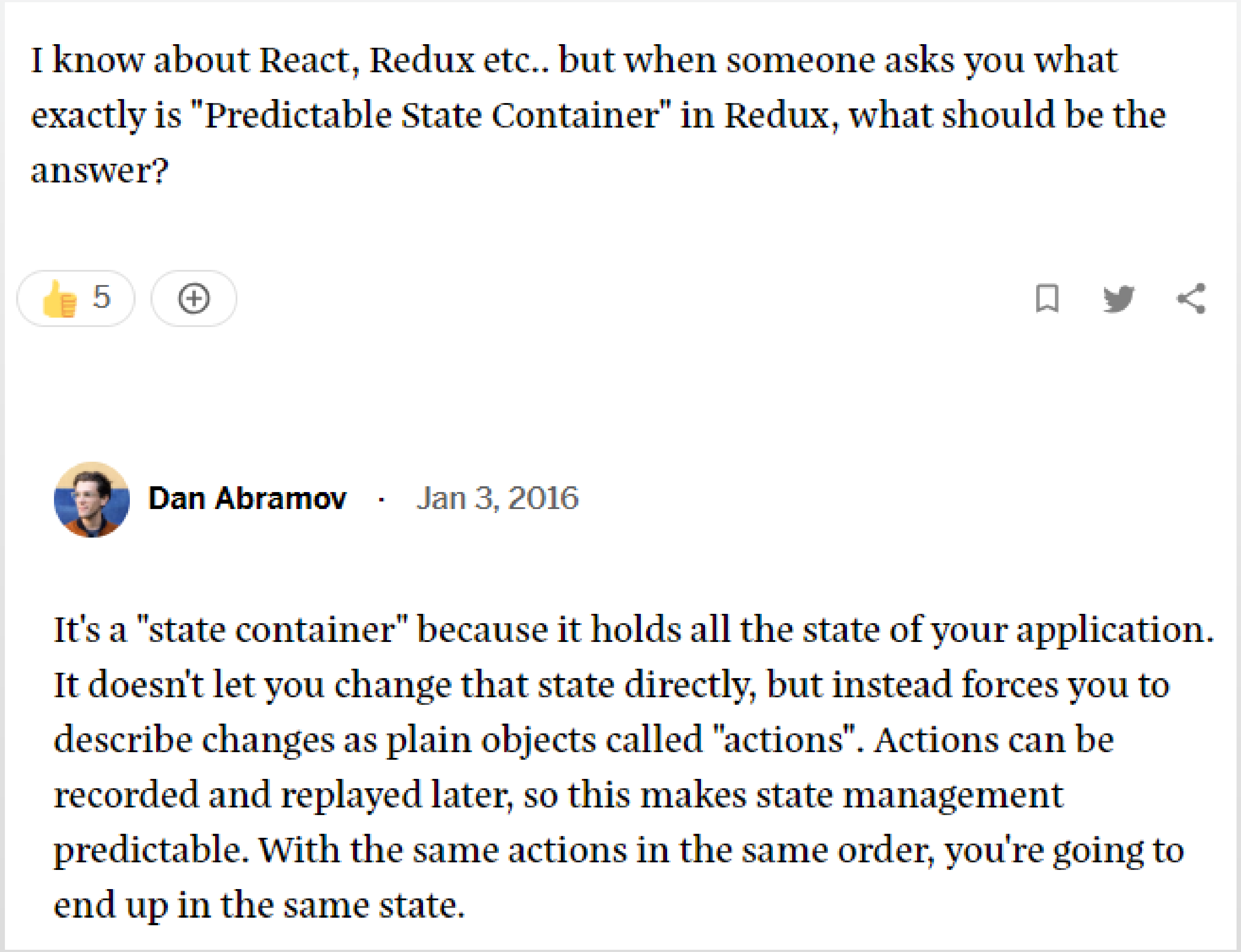
- Created 2015 - inspired by **Flux** pattern
- Flux is a ***pattern***, Redux is a ***library*** that can be considered as an implementation of Flux for application state management.
- While both Flux & Redux were developed for use with React, they are separate from React and can be used with other projects.
    - You can also use React without using Flux pattern or Redux library!
- Redux is popularly used for more complex apps with many components & global state changes. Not all apps need Redux (nor any external library for state management)

Flux vs Redux

Flux	Redux
Pattern (no actual code)	Library (actual code)
Multiple stores	Single store
Mutability	Immutability
Single central dispatcher object, all actions must go through it, central manager for multiple stores	No single central dispatcher object, has a central store and a dispatcher function to dispatch actions

Redux

- Redux is "**a predictable state container**" for JavaScript apps - you will hear/read this a lot. What does this mean?
- Here's Redux's creator's soundbite on it:



I know about React, Redux etc.. but when someone asks you what exactly is "Predictable State Container" in Redux, what should be the answer?

Dan Abramov · Jan 3, 2016

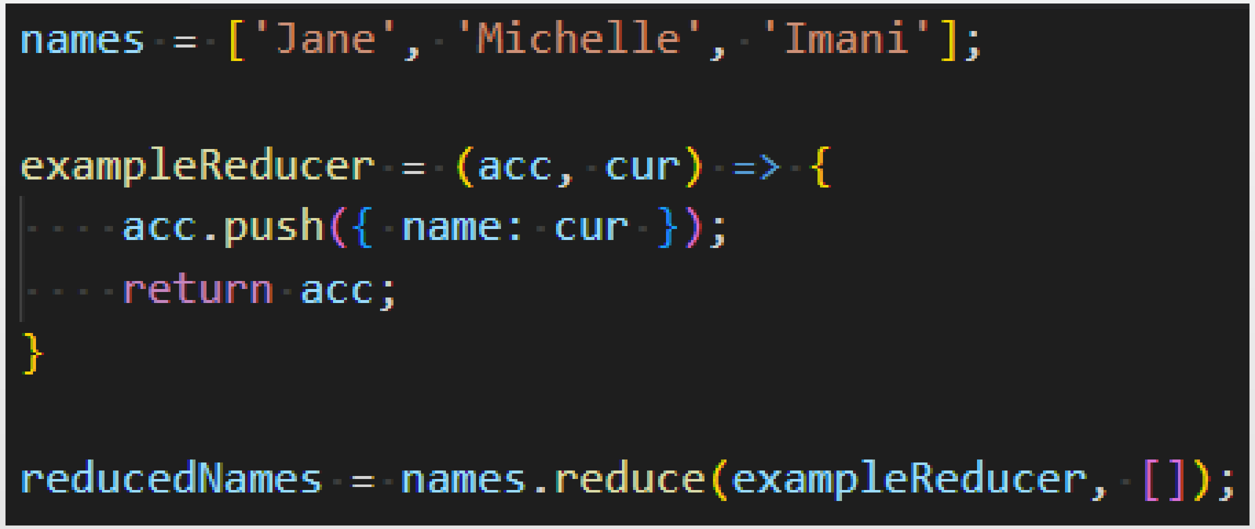
It's a "state container" because it holds all the state of your application. It doesn't let you change that state directly, but instead forces you to describe changes as plain objects called "actions". Actions can be recorded and replayed later, so this makes state management predictable. With the same actions in the same order, you're going to end up in the same state.

JavaScript: Array.reduce()

- Non-mutating, higher-order array method
- Required 1st argument: A callback function known as a **reducer**, which has two parameters: an **accumulator** value, and a **current** value
- Optional 2nd argument: An **initial value** for the **accumulator**
- **Array.reduce()** will iterate over an array, potentially updating the accumulator value at each iteration based on the value of each array item, then return the final accumulator value at the end
- The power of **Array.reduce()** is that its return value can be anything you want, depending on how you write the reducer and initial value

JavaScript: Array.reduce()

- Example:

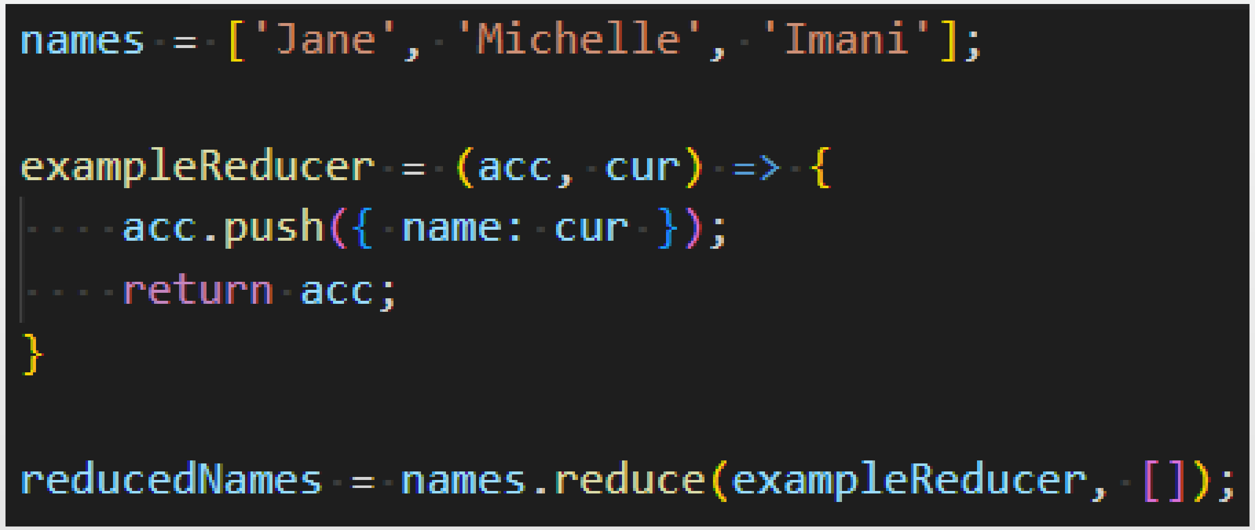


```
names = ['Jane', 'Michelle', 'Imani'];

exampleReducer = (acc, cur) => {
    acc.push({ name: cur });
    return acc;
}

reducedNames = names.reduce(exampleReducer, []);
```

- The array **names** is an array of strings.
- **Discuss:** What is the value returned and assigned to **reducedNames** from the call to **reduce()** shown above?

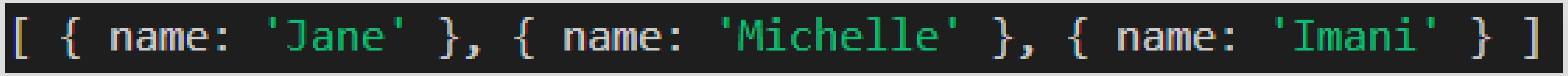
JavaScript: Array.reduce()

- Example:



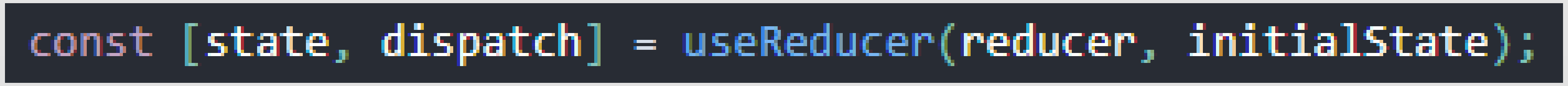
```
names = ['Jane', 'Michelle', 'Imani'];

exampleReducer = (acc, cur) => {
    acc.push({ name: cur });
    return acc;
}

reducedNames = names.reduce(exampleReducer, []);
```

- The array **names** is an array of strings.
- **Discuss:** What is the value returned and assigned to **reducedNames** from the call to **reduce()** shown above?
- Answer: An array of objects:



```
[ { name: 'Jane' }, { name: 'Michelle' }, { name: 'Imani' } ]
```

The useReducer hook

- Built into React core library
- Use to manage more complicated state than **useState**
- Pass to it a **reducer** function and an **initialState**, similar to what you would pass to Array.reduce()
- Returns the current **state** and a **dispatch** function instead of a setter function
- Example: 

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

- The dispatch function is used to send an action to the reducer, which updates the state

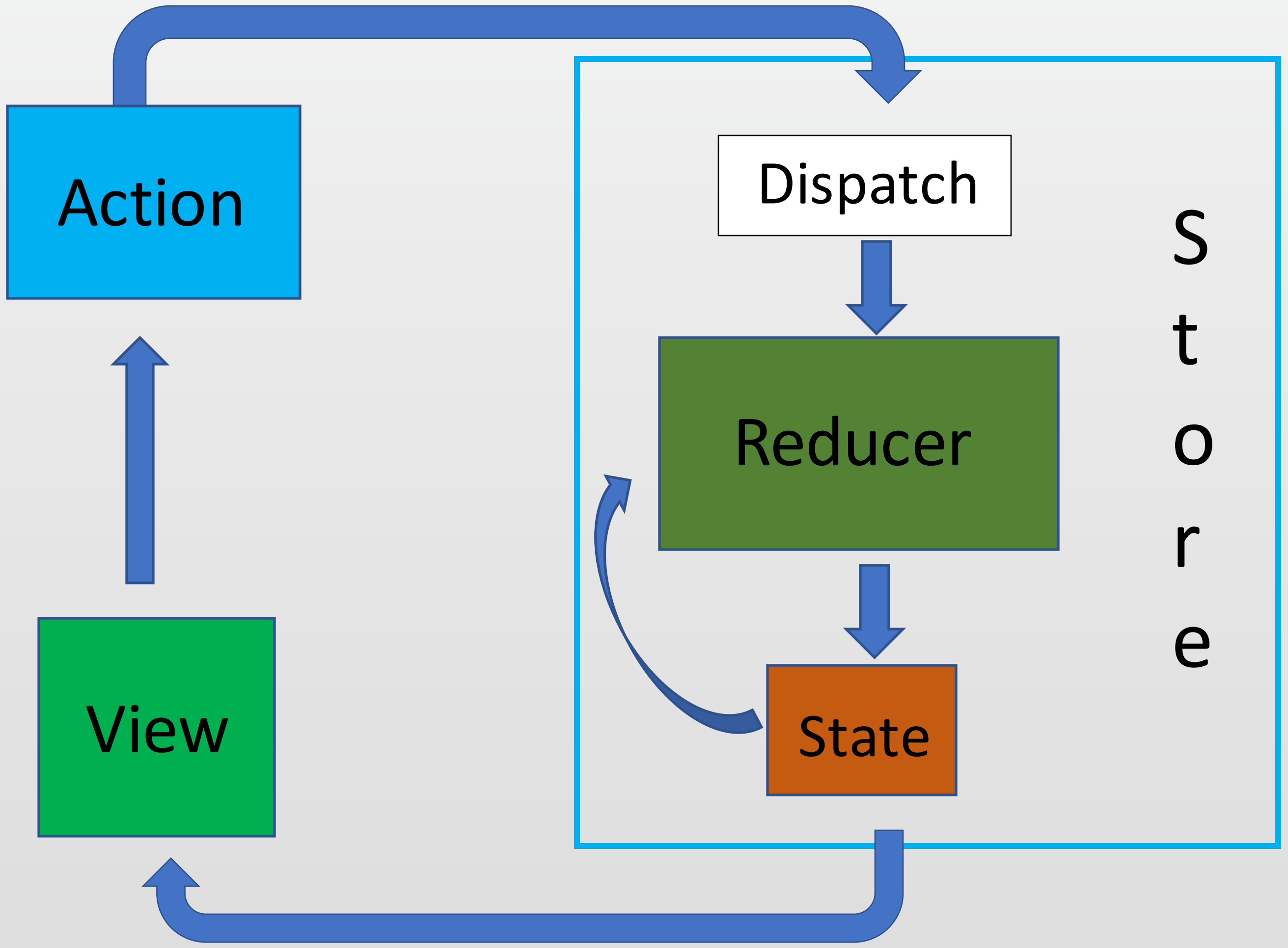
Redux: Three Principles

1. ***Single source of truth***
   - Single state object tree within a single **store**
2. ***State is read-only***
   - Changes only made through **actions**
3. ***Changes are made with pure functions (reducers)***
   - Takes previous state and action and returns next state without mutating the previous state (immutability)

Redux: Main concepts

- *State*: Stored as JS object in the Redux Store
- *Action*: JS object that prescribes what state change to make
- *Reducer*: Pure function that takes current state and action, then returns new state
  - Updates state immutably (does not modify inputs)
  - Cannot call any impure functions like Math.random(), nor perform any side effects like API calls
  - Called *reducer* because it reduces multiple inputs (current state, action) to a single return value (similar to a function that you might use with Array.reduce)

Redux data flow



React-Redux

- The react-redux library helps use Redux with React
- **Provider** component
  - Surround your root component with **<Provider></Provider>** (typically <App>)
  - Must take the store as an attribute: **<Provider store={store}>**
  - Makes it possible for all components to access the store
- The **useSelector** hook
- The **useDispatch** hook

React-Redux: useSelector()

- **useSelector()** provides functions access to the state in the Redux store
- To use it, import it, then pass to **useSelector()** the functions that need access to it
- You can define that function inline:

```
const students = useSelector((state) => state.students.data);
```

- Or separately:

```
const selectAllStudents = (state) => state.students.data);
const students = useSelector(selectAllStudents);
```

React-Redux: useDispatch()

- **useDispatch()** returns a function we use to dispatch actions to the Redux store
- Usage: Import it, assign the function to a variable name, then call the function and pass an action to it
- Remember that action objects need a **type** and an optional **payload**
- When we use **createSlice()** from **Redux Toolkit** and define reducers on it, it provides us with **action creators**, which are functions that create action objects that correspond to those reducerse
- We can pass these action creators to **useDispatch()**

useDispatch() example

- Example from the course project:

```
const dispatch = useDispatch();
dispatch(addComment(comment));
```

- The **addComment** action creator function takes the **comment** and returns an **action** object with the corresponding action **type**, and a **payload** of **comment**
- **useDispatch()** then sends that action to the Redux store

Redux ToolKit (RTK)

- A companion library for Redux
- RTK makes Redux easier to use, with less boilerplate
- RTK is now the recommended way from the Redux team to use Redux

RTK: createSlice()

- **createSlice()** creates a "slice" of the Redux state
- Pass to it a configuration object with the name of the slice, its initialState, and any reducers
- It will return an object that contains the state, reducers, action creators, and more
- **Action creators** are functions that create and return action objects
- RTK implements the **Immer** library so that you can use mutating logic inside createSlice() and it will automatically be converted to non-mutating logic

RTK: configureStore()

- Use to create and configure the Redux store
- Pass a configuration object when calling **configureStore()** that contains:
  - Either a single root reducer, or an object that contains multiple slice reducers
  - If you pass multiple slice reducers, Redux will combine them into a root reducer, and maintain separate 'slices' of the store that are each handled by the corresponding reducer
  - Optionally pass an array of middleware
- **configureStore()** returns the **store** object, which is a container for the application state
- The **store** object also provides **dispatch()** and other useful methods

Workshop assignment tips

A few tips before you begin your assignment:

- If you feel lost, use the files you created during the exercises this week for syntax hints.
- Don't forget to **export** components from their files.

- When debugging, try check the browser developer console for error messages, and try using the **React DevTools** browser extension.

- When using any JavaScript (variable, function, map method, etc) inside **JSX**, you must surround the JavaScript in **curly braces.**

- Use both start & end tags to render **JSX** elements/components (e.g. **<Col></Col>**, unless the instructions specify to use a self-closing tag (e.g. **<CampsitesList />**).

Workshop 4 Assignment

In this workshop, you will:

- Create a user feature, including **userSlice.js** and **UserLoginForm.js** files
  - Use the **createSlice()** function to create a **userSlice** object, including its initial state and a case reducer called **setCurrentUser**
  - Use the **useSelector** hook to obtain the current user
  - Use the **useDispatch** hook to set the current user
- Use **Formik** and **Reactstrap** components to create a **UserLoginForm** component
  - Write **form validation**

You must finish all the exercises for this week before beginning the Workshop Assignment.

You will be split up into groups to work on the assignment together.
Talk through each step out loud with each other, code collaboratively.
If your team spends more than 10 minutes trying to solve one problem, ask your instructor for help!

Check-out

- Submit your assignment files in the learning portal:
- Wrap up – Retrospective
  - What went well
  - What could improve
  - Action items
- If there is time remaining, review any remaining questions from Week 4, or start Week 5, or work on your Portfolio Project.