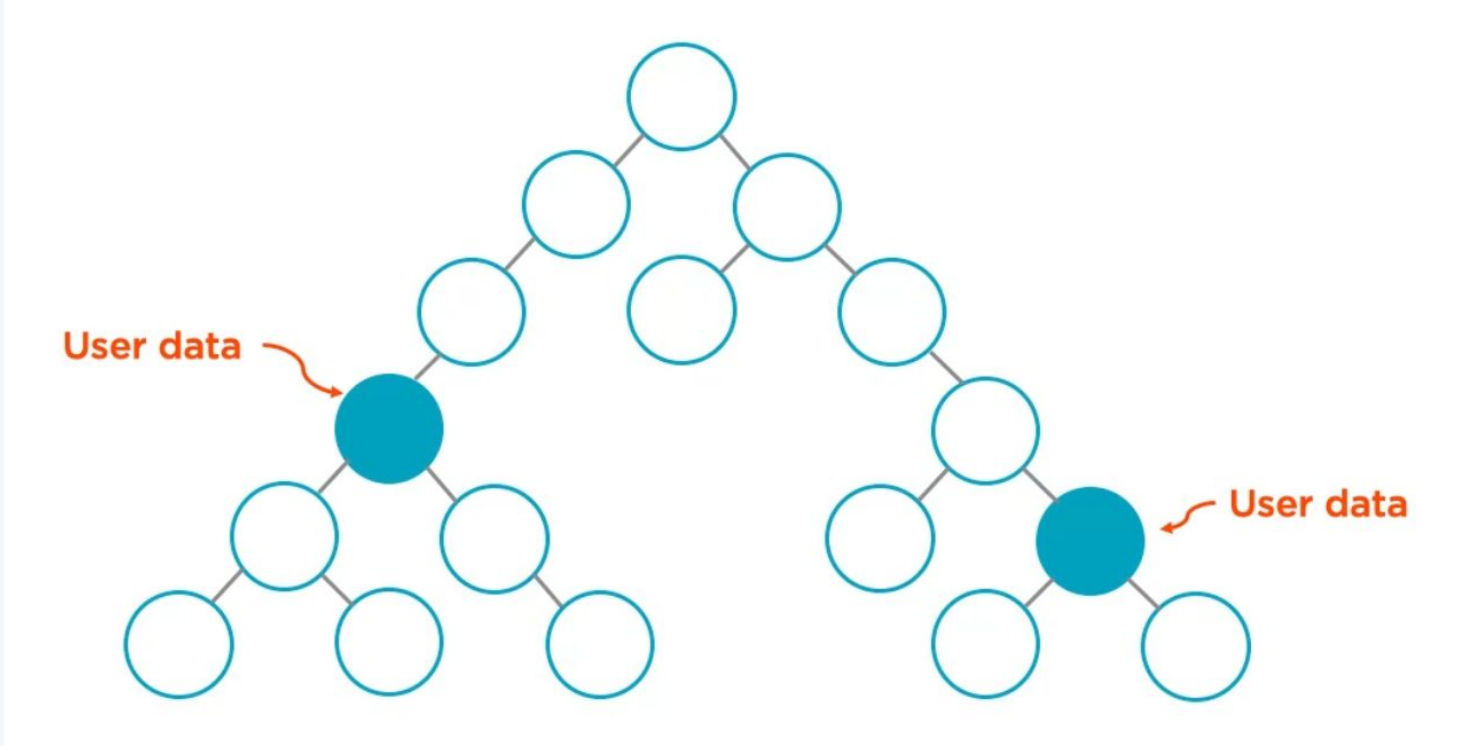
React Router & Redux

Fundamentals

New Horizons®

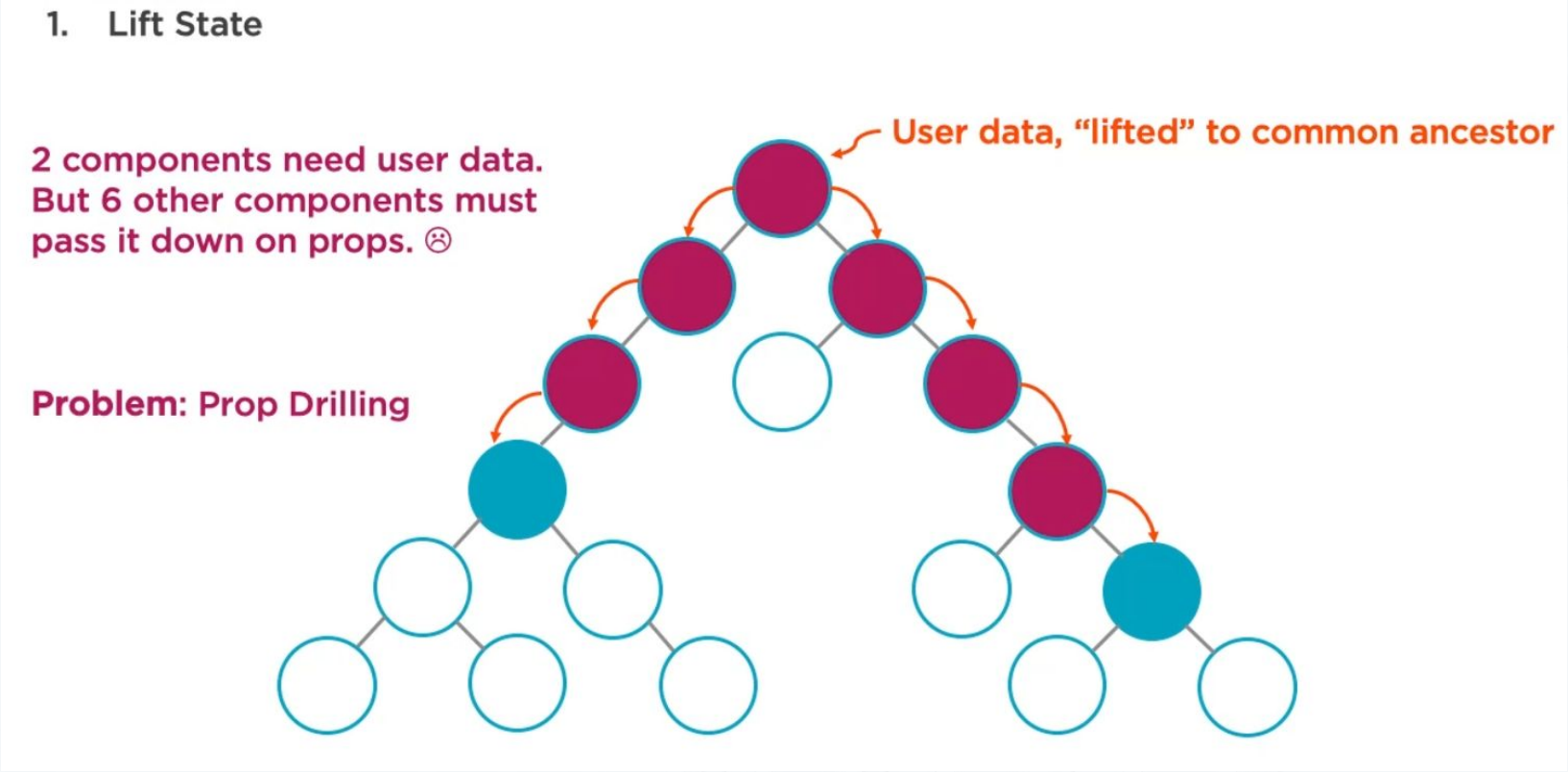
What is Redux?

- Complexity and volume in your application will only increase
- What if we need to share some user data with arbitrary components?



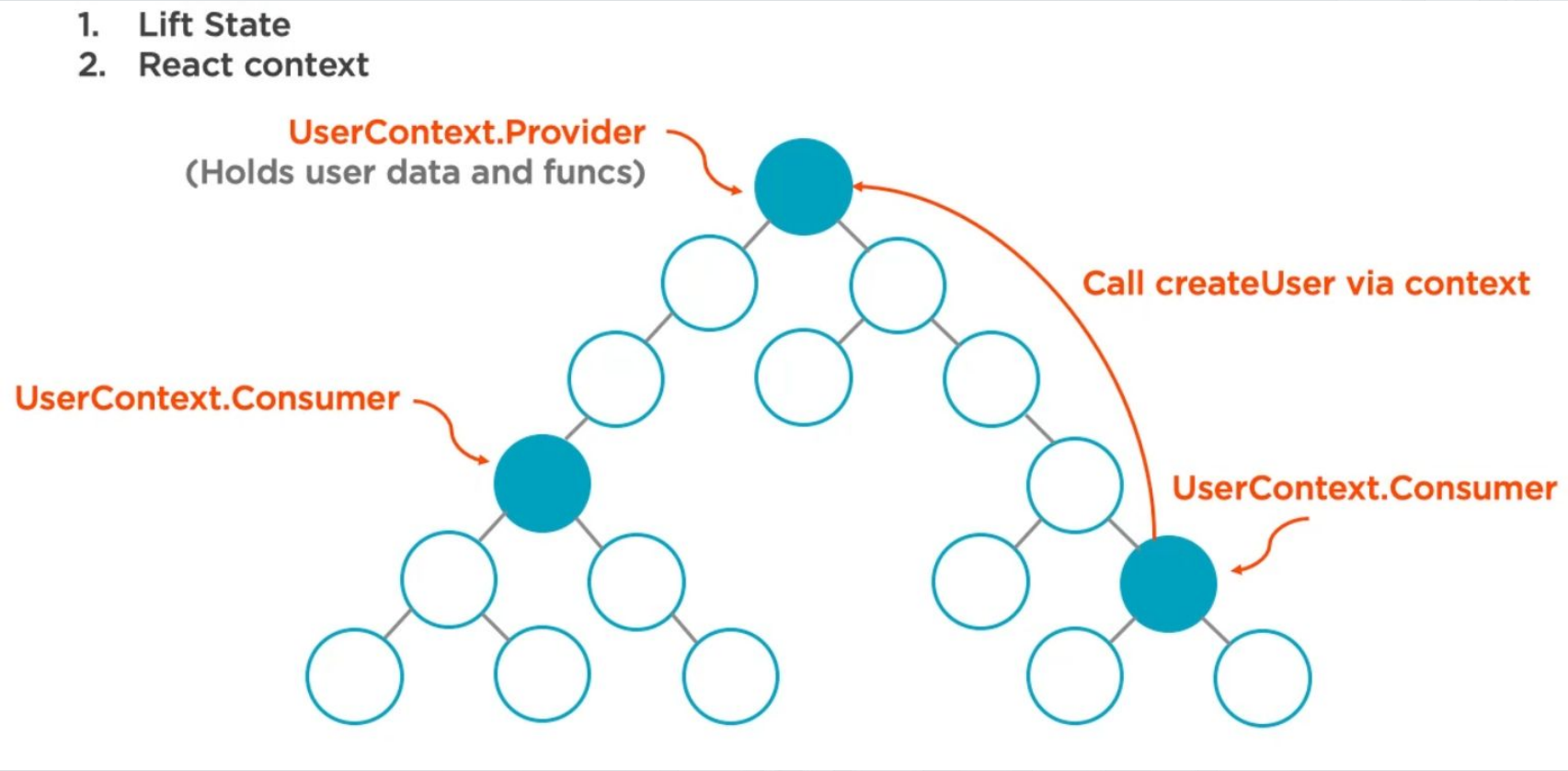
What is Redux?

- We can keep the data in the root component but then we will get into complicated prop drilling - only good for small applications



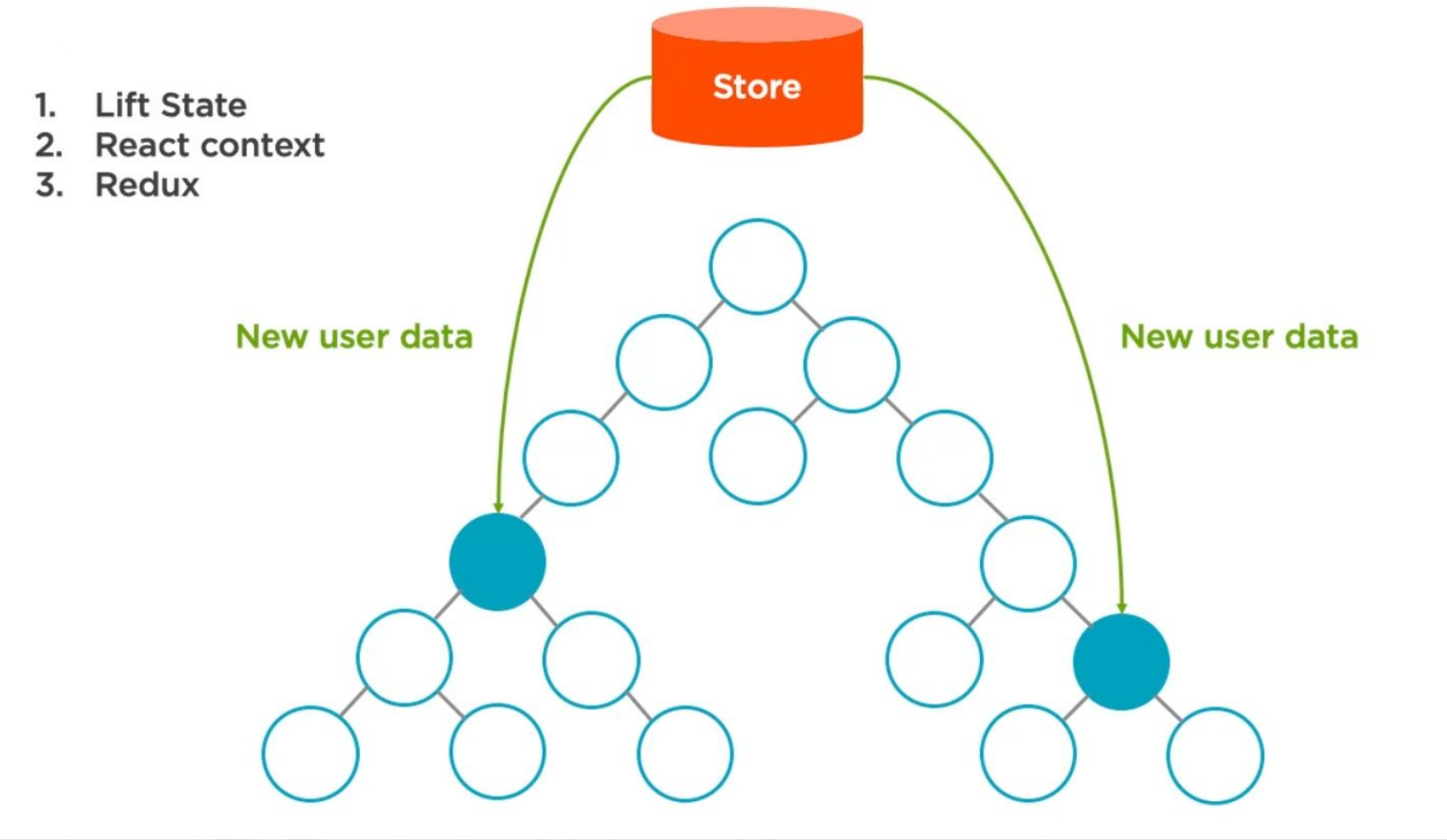
What is Redux?

- We can also use the React built in context



What is Redux?

- Or we can use Redux which is a centralized store that holds and provides access to all the data



What is Redux?

- Redux is not going to be helpful in small applications

- The Redux benefits come to light when you deal with:

  - complex data flows
  - inter-component communication
  - non-hierarchical data
  - many actions including access permissions etc.
  - same data used in many different places

What is Redux?

- So when do we use Redux?
- You can start your app without it and as soon as data sharing and prop lifting becomes a burden, you can then implement it
- Simply said: if you aren’t sure if you need it, you don’t need it!

Redux Principles

- Redux is one immutable store - data cannot be changed directly inside of it
- Changes are triggered by actions - another way to think about it is a component sending a notification to the store and requesting to change the data
- Reducers update the state - special functions that make sure data is intact and consistent

Redux Principles

- Reducers take the current state as input, modify it and then they return new state which is updated in the store
- It sounds complicated but they are just functions that execute state changes
- Once they return the new state, the Redux store notifies all components in React that use that state and they are automatically updated - examples will follow!

Redux Principles

- Actions are used to notify the store for changes
- They contain plain objects that need to have a type property with the rest of the data being up to you
- The action objects are usually return by functions that we call Action Creators:

```
rateCourse(rating) {
    return { type: RATE_COURSE, rating: rating }
}
```

Redux Principles

- The Redux store is one and the same for the whole application
- The data in the store is immutable which means that the only way to change it is to dispatch an action to the store:

  store.dispatch(action)
- You can then access the data by getting the state of the store:

  store.getState()

We will create examples shortly

What is Immutability?

- To change state, you must return a new object as the new state
- In fact, some data type objects in JavaScript are already immutable - String, Boolean, Undefined, Null, etc.
- When you try to change the values of these objects, a new object is created

What is Immutability?

- In traditional mutating state we can change values directly:

```
state = {
    name: ‘John Doe’,
    role: ‘author’
}
```

We can now do something like this:

```
state.role = ‘admin’;
```

What is Immutability?

- When dealing with immutable state we have to do:

```
state = {
    name: ‘John Doe’,
    role: ‘author’
}
```

And now to change it, we return new object:

```
state = {
    name: ‘John Doe’,
    role: ‘admin’
}
```

JavaScript Spread Operator Review

- For future reference you can easily create new objects in JavaScript and replace some of their existing properties by using the spread operator (...):

  const newState = { ...state, role: 'admin' };

- This will create newState with all existing properties in the state object and it will either replace the value of the role property if it exists or add it all together if it doesn't

JavaScript Handling Array Review

- In the same context, if your state is an array, try to avoid using JS methods like push, pop and reverse because the modify the original data and do not create new object - you must clone your array first if you want to use them
- You can instead use some of the other JS array methods that return a new array by design such as map, filter, reduce, concat, etc.

Why Immutability?

- It actually solves concurrency access issues
- The reducers act as a lock to the data so when an action notification occurs they manage them one by one in order not to corrupt the data
- It also means that if you have concurrent access, you don’t have to check if a specific property has changed
- Either the whole state object changed or everything is the same

Handling Immutability

- It’s extremely important that you do NOT change state directly into the store as that could lead to bugs
- Being careful goes a long way but you can always install extensions that will track and warn you if you do a direct store change as (npm) redux-immutable-state-invariant
- Or you can implement a library that will do it for you such as Immer, Immutable.js and others

More on Reducers

- Before we write some code, you have to make sure that your reducers return new state objects and they do not change the state directly
- They should also be pure functions without any side effects so that their behavior is predictable
- You can implement multiple reducers in your app

More on Reducers

- And how does Redux know which reducer to call once an action is dispatched?
- Well, it calls ALL of them so you need to make sure that by default you return the original state
- Based on the passed type, certain logic in a reducer can be trigger to return a different new state object instead

Container vs Presentational Components

- One last things to be aware of before we start coding:

Two Component Types

Container	Presentational
Focus on how things work	Focus on how things look
Aware of Redux	Unaware of Redux
Subscribe to Redux State	Read data from props
Dispatch Redux actions	Invoke callbacks on props

react-redux

- In fact you can use Redux with other libraries to manage your state (even with Angular or vanilla JS) as it is not exclusive only to React
- What connects Redux with React is a separate library called react-redux that you need to add as well

react-redux

- The react-redux library provides two main components
- Provider - attaches the store to the app usually used only once in your main App component
- Connect - creates container components that have access to Redux

react-redux

- The connect exposes a function which wraps a component and defines what state and what actions we want to expose to this component:

```
export default connect(mapStateToProps,
mapDispatchToProps)(AuthorPage);
```

react-redux

- The parameters to the connect() function are actually functions themselves
- mapStateToProps is a function that will get called every time data in the store changes
- It will return an object and each property defined in that object with become a property in the container class

react-redux

- The mapDispatchToProps is very similar but it returns an object with the actions that we want to expose
- This function gets the dispatch one as a parameter
- You can choose to completely ignore this and call dispatch directly as this:
  this.props.dispatch(loadCourses());
- However, you can avoid typing by using a Redux approach

react-redux

- Redux provides a bindActionCreators function that you can use with mapDispatchToProps like this:

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

- Once you do that, you can use the newly created actions prop in your component and call an action like this:

```
this.props.actions.loadCourses();
```

react-redux

- That's still a lot of typing so you can declare

  mapDispatchToProps as object to directly map your actions to props like this:

  ```
  const mapDispatchToProps = { loadCourses };
  ```

- Then in the component you will use:

  ```
  this.props.loadCourses();
  ```

Summary as Conversation

A Chat With Redux

React	Hey CourseAction, someone clicked this "Save Course" button.
Action	Thanks React! I will dispatch an action so reducers that care can update state.
Reducer	Ah, thanks action. I see you passed me the current state and the action to perform. I'll make a new copy of the state and return it.
Store	Thanks for updating the state reducer. I'll make sure that all connected components are aware.
React-Redux	Woah, thanks for the new data Mr. Store. I'll now intelligently determine if I should tell React about this change so that it only has to bother with updating the UI when necessary.
React	Ooo! Shiny new data has been passed down via props from the store! I'll update the UI to reflect this!

Redux Setup

- The initial setup can be overwhelming at first as you need to:
  - Create action
  - Create reducer
  - Create root reducer
  - Configure the store
  - Instantiate the store
  - Connect component
  - Pass props via connect
  - Dispatch action

Redux Setup

- However, once you do that, adding additional features is much easier as you will only need to:
  - Create action
  - Enhance reducer
  - Connect component
  - Dispatch action

Setting Up the Project

- Obtain the starter-project from your instructor
- Run npm install and explore the files
- Note the usage of React Router
- Next, let's start building the store backwards - starting from the data in the component

Add Course Form

- Add a state with course.title to the CoursesPage
- Add a form with text input to add a new course and submit button with event handler
- Inside the handler create a new object cloning the current state, substituting with the input value and calling setState with the newly created object

this keyword

- Make sure that your event handler uses arrow function
- In arrow functions the this keyword always refers to the class where the function is defined
- In non-arrow functions it could refer to many different things, specifically the instance of the function itself, etc.
- In our case this.state needs to refer to the governing class

this keyword

- Also note that we are talking about classes here
- If we used functions as components, we wouldn't have this problem and we shall do so later on just so you can see the difference

Create Course Action

- Next we need an action to call once the form is submitted
- Create new folder and file: redux/actions/courseActions.js
- Inside create a function that takes a course as an argument and returns an object with type and data
- You can extract the types in a separate file as constants and import them all from there so they are the same everywhere

Create Course Reducer

- The action needs a reducer to change the state so create file: redux/reducers/courseReducer.js
- It takes state and action as input parameters
- Write code that extracts the new course from the action and adds it to the state which represents array with all of our courses

Course Reducer Notes

- Make sure that the state has a default of empty array
- Make sure that your switch statement for the action contains a default which returns the default state
- Also make sure to not directly update (push) the array
- All of this covers different possible scenarios

Course Reducer Notes

- Each reducer has the goal of managing part of the entire state of the application which means we are going to have multiple reducers
- We define and register them all via combineReducers in a root reducer that is then used and implemented in the store
- Note: If you export default your reducer functions you can then use whatever name you want to import them in the root

Create Root Reducer

- Create redux/reducers/index.js file to hold our root reducer
- Define and export it as a const that gets the object returned from combineReducers
- Add the course reducer as a param to that function - only one for now but we will create more later

Create the Store

- The next step is to create the store
- This is as simple as calling createStore with the rootReducer and initialState as parameters
- However, since we do this only once, let's make things crazy and add library that will warn us if we accidentally mutate any state plus support for Redux dev tools

Create the Store

- We can use redux-immutable-state-invariant library and call its reduxImmutableStateInvariant function as parameter to the core Redux applyMiddleware method which allows us to apply third party functionalities
- We can also implement (the crazy looking) composeEnhancers function that adds support for the Redux dev tools

Create Store

- We can create the store in redux/configureStore.js
- Call createStore with rootReducer and initialState
- Implement reduxImmutableStateInvariant
- Apply the composeEnhancers as middle ware
- We only do this once - set it and forget it

Instantiate the Store

- Call configureStore() in our application's main index.js
- Import the Provider (you can call it ReduxProvider for clarity) and wrap your App render with it passing the store as a prop
- Now every component in your App can access the store

Wiring the CoursesPage as Container Component

- We can now connect the CoursesPage to the store and call an action to update the state there
- We will use the connect function that we mentioned before:

  connect(mapStateToProps, mapDispatchToProps)(CoursesPage);
- Why two parenthesis? Simply connect() returns a function and we then call this function passing our CoursesPage class as input parameter

Connect Container Component

- Implement the connect() function in CoursesPage
- Define mapStateToProps to return a courses property from our state
- Define mapDispatchToProps to use bindActionCreators and map the courseActions to a actions property
- Implement handleSubmit for the form and call the createCourse action in there with the state from the page

Finishing Touches

- Let’s display the courses prop that we created below the form so we can have a visual on the store data
- We can also define propTypes for our CoursesPage to make sure we know what to expect from the already defined props coming from the store
- You can trace the data flow by setting some debugger; statements in your code and refreshing the browser
- You can also add Redux DevTools extension for Chrome now and explore the Redux tab when you press F12

Finishing Touches

- Also note that we used bindActionCreators and we extracted all (right now it's only one) actions from the courseActions.js into a prop called actions
- This will allows to not make any changes as we add more actions in the future as they will become part of the actions property when we add them

Finishing Touches

- You can also use the object approach instead like this:

```
const mapDispatchToProps = {
    createCourse: courseActions.createCourse
}
```

- You can then call the method directly as `this.createCourse()` from the component
- However, if we add more actions to the app, we will have to come back and update that object before using them

Async in Redux

- Let’s create a Mock API and implemented it in our App
- Get the tools folder from your instructor and explore
- apiServer.js sets mock api via Express and json-server
- mockData.js is exactly our mock data to use
- createMockDb.js will create a mock DB file each time we run npm start which will then be read by our json server

Async in Redux

- To utilize the new mock api we need to add some scripts to the package.json file:
  “prestart:api”: “node tools/createMockDb.js”,
  “start:api”: “node tools/apiServer.js”
- Note that prestart will automatically execute when we run start as it has the same name with pre in front
- We now start the api with: npm run start:api

Async in Redux

- This whole setup will create a clean DB file every time

  we start the application so we don’t have to worry about corrupting or polluting the original data during dev

- You can test the api by going to localhost:3001

- If we want to start everything at the same time we can

  use the run-p command that runs batch of scripts →

Async in Redux

- In package.json rename the “start” script to “start:dev”
- Add a new script on top like this:

  “start”: “run-p start:dev start:api”,
- Now typing npm start will automatically start your server and the api at the same time

Async in Redux

- Next get the src/api folder from your instructor
- Explore the files - they contain functions that call and work with the endpoints from the mock api that we can easily use in our application
- Note: fetch is built into modern browsers so you can freely use it with JavaScript without the need of additional library to make API calls

Async in Redux

- One final note: the api files use a variable up top:

  process.env.API_URL

- This is defined in the webpack.config.dev.js file

- If you don’t have it, you can add the definition to the

  plugins section (make sure to import webpack up top)

Middleware & Async Libraries

- We already introduced the notion of middleware but this is basically a piece of code that runs between your action call and the execution of the reducer code
- You can write your own or you can use third party ones
- We can use some premade ones to handle async calls in Redux

Middleware & Async Libraries

- The most popular Redux async libraries are:

  redux-thunk - returns functions from action creators
  redux-promise - uses promises for async
  redux-observable - uses RxJS observables

- Thunks are relatively easy to learn so let's focus on them

- Definition of thunk: A function that wraps an expression to delay its evaluation

Middleware & Async Libraries

- If that sounds like too much: Middleware isn't required to support async operations in Redux
- However, you probably want it as it sounds scary at first but it will save you a lot of code and trouble if you use it
- Without it, you'd have to pass dispatch as parameter to your api calls to execute it once they finish

Refactoring the CoursesPage

- Now that we have an API, we would like to actually have at least two components - one container component that will manage the state, make API calls and work with Redux and another presentational component that will simply display a list of courses
- Let’s go ahead and remove the form and its handlers functionality from the CoursesPage for a start

Thunks

- In order to use thunks we need to include them as middleware to our configuration at configureStore.js
- We first need to import the library:

  import thunk from "redux-thunk";
- Then we need to add it as a parameter to the applyMiddleware function:

  applyMiddleware(thunk, reduxImmutableStateInvariant())

Thunks

- Let’s create our first thunk in courseActions.js action
- Create a loadCourses() function that prompts the API for the list of courses
- Since this is a thunk, we will return a function and that function takes dispatch as input parameter
- You can dispatch the result from the API as a new action

Thunks

- Note that you can call directly dispatch like this:

  dispatch({ type: “LOAD_COURSES”, courses});

- However, you can also create a separate action function

  for concern separation and clarity in your code so adding functions like loadCourseSuccess or even loadCourseError is the preferred way to go

Thunks

- Make sure to update actionTypes.js to include our new constant LOAD_COURSES_SUCCESS
- We now need to handle the new action in our courseReducer.js by adding new case to the switch
- Remember that whatever you return from the reducer becomes the new state!

Thunks

- Note that you can call directly dispatch like this:

  dispatch({ type: “LOAD_COURSES”, courses});

- However, you can also create a separate action function

  for concern separation and clarity in your code so adding functions like loadCourseSuccess or even loadCourseError is the preferred way to go

Thunks

- All we need to do now is call the loadCourses action

- We can do that on load of the CoursesPage using hook:

```
componentDidMount() {
    this.props.actions.loadCourses().catch(error ⇒ {
        alert("Loading courses failed" + error);
    });
}
```

Thunks

- Note that we can call loadCourses() without any modifications as we already used bindActionCreators
- Now let’s create a CourseList.js component to display our data
- You can work with your instructor or just copy and paste the file

Destructuring

- Side note: every component gets props as input: (props)
- You can then create variables to extract data from it:

  const courses = props.courses;

- Instead, you can destructure the props and only get the ones you are interested in upon component initialization:

  const CourseList = ({ courses }) => (

Redux Workflow

- To practice everything, let's add functionality to display the author's name next to the course list names
- Add new constant to actionTypes.js
- Create new action in: redux/actions/authorActions.js
- Populate the file to make API call to get the authors and a thunk to dispatch an action with the new data

Redux Workflow

- Create authorReducer.js to react on the LOAD_AUTHORS_SUCCESS action and return new data
- Update the root reducers/index.js to include the new authors reducer
- In CoursesPage.js rework the mapDispatchToProps so we can use the loadAuthors function as well

Redux Workflow

- We can then add the call to loadAuthors() in componentDidMount similar to the loadCourses() one
- Our courses contain the authorId so now we can connect the two states from the store and display the author's name instead by modifying mapStateToProps → see next slide for details

Redux Workflow

- In mapStateToProps we can refactor by adding the name to each course from the list:

```
courses: state.courses.map(course ⇒ {
    return {
        ...course,
        authorName: state.authors.find(a ⇒ a.id ===
course.authorId).name
}})
```

Redux Workflow

- You also need to make sure that authors length is not 0

  a.k.a. we have some data to work with

- You can use ternary operator before the return

- Let’s also expose the authors as prop at the end

Neat Tricks

- You will notice that clicking on the navigation and going back to the courses page makes an API call every time when we display the list
- However, the data is already there - in the store
- So we end up refreshing with the same data over and over again

Neat Tricks

- We can avoid that by just checking if we have loaded data when the component mounts and make an API call only if we do not
- In the componentDidMount we can add a check before the API call to get courses like this:

```
if (courses.length === 0) {
```

Neat Tricks

- You can also use destructuring in the componentDidMount function to reduce some code:

  ```
  const { courses, authors, actions } = this.props;
  ```

- You can then remove the this.props in your code in that function all together

Neat Tricks

- Finally, you can declare a file reducers/initialState.js which will represent the full picture of the values in your store with their initial state
- This is helpful if you get into this project and you want a high level overview of how the store is structured
- You can then import this new file into the other reducers and get the initial default parameter values from it

Async Writes

- Next up we want to utilize the endpoints that will allow us to create and update courses
- Let's create new component called ManageCoursePage.js
- We can start by copying over CoursesPage.js and leave the componentDidMount the same along with simple render function, the mapStateToProps and mapDispatchToProps along with the connect function at the end

More Neat Tricks

- Let's convert mapDispatchToProps into an object:

```
const mapDispatchToProps = {
    loadCourses: courseActions.loadCourses,
    loadAuthors: authorActions.loadAuthors
}
```

- Modify modify your componentDidMount to extract these functions directly from the props and use them without having actions first
- You will need to modify your propTypes too

More Neat Tricks

- We've seen that before but we can simplify this even further by tweaking our imports and destructuring as soon as we import:

```
import { loadCourses } from "../../redux/actions/courseActions";
import { loadAuthors } from "../../redux/actions/authorActions";
```

- We can now go and change the mapDispatchToProps to this:

```
const mapDispatchToProps = {
    loadCourses,
    loadAuthors
}
```

Add Routing

- Go to App.js and add two new routes for the newly created component:

  ```
  <Route path="/course/:slug" component={ManageCoursePage} />
  <Route path="/course" component={ManageCoursePage} />
  ```

- Note again that the order here matters so you want your slug route first, otherwise it will never be matched

Hooks

- Hooks allow us to handle state and side effects (think lifecycle methods) in functional components
- As mentioned previously, function components are often preferred so let's convert our ManageCoursePage.js from a class to function
- We will have to start using hooks inside

Hooks

- We can start by converting the class to a function and destructuring the props as parameters instead of doing that in the code (in ComponentDidMount()):
  `function ManageCoursePage({ courses, authors, loadAuthors, loadCourses }) { ...`
- We can then replace ComponentDidMount with the useEffect hook
- We can also remove the render() function and leave the return statement to execute at the end of our function component

useEffect Reminder

- If you remember from before - the useEffect will now run every time the component renders
- We only want it to run once so we can add our dependency array at the end
- For now, it can be empty but that will guarantee that the code will run only once: , []);

Creating Course Form

- Let’s create a form that we can fill in to generate course data
- We can create custom component to handle input fields and reuse throughout our application instead of hard-coding them
- Get and explore the following files from your instructor:

  courses/CourseForm.js

  common/TextInput.js

  common/SelectInput.js

Creating Course Form

- In order to render the form on the screen, initially we are going to need an empty course object to pass as prop
- One is available in the mockData.js file:

  import { newCourse } from "../../../tools/mockData";
- We can add it to mapStateToProps next:

  return { course: newCourse, ...
- Don’t forget to update your propTypes too

Creating Course Form

- If we want to eventually track form input in that new variable as well, we will need to add a state hook:
  `const [course, setCourse] = useState({...props.course});`
- What did we put as a default? We will have a course passed in as a prop but if we destructure it as course then we have ambiguity as the state name is also course

Creating Course Form

- So instead we can destructure the props themselves with the rest operator in the initialization of the function:
  function ManageCoursePage({courses, authors, loadAuthors, loadCourses, ...props}) {
- The ...props there means that we store any unreferenced properties so far in a variable called props
- Then we can extract the default from the props as default like this:
  const [course, setCourse] = useState({ ...props.course });
- You can potentially use an alias instead: course: initialCourse

Creating Course Form

- Note here that we are using React state to hold the forms data and not Redux
- You want to use Redux for global data only that is shared between components
- If you have local values that you need to hold, there's no need to put them in the store and bloat it

Rendering the Form

- We can now render the form on the screen in the return:

  `<CourseForm course={course} errors={errors} authors {authors} />`

- We don't have the errors defined yet so let's make another state variable up top:

  `const [errors, setErrors] = useState({});`

- Errors are empty for now, we will work with them later

- We can test by manually going to localhost:3000/course as we don't have a link to that form anywhere yet

Universal Change Handler

- If you explore the form you will see that all change events are passed to the same change handler function
- That allows us to create one universal function that will process the change event and update the local state with the data that changed for the course

Universal Change Handler

- The function that handles all changes looks like that:

```
function handleChange(event) {
    const { name, value } = event.target;
    setCourse(prevCourse => ({
        ...prevCourse,
    [name]: name === "authorId" ? parseInt(value, 10) : value
  }));
}
```

- Don’t forget to add: <CourseFrom ... onChange={handleChange} />

Universal Change Handler Notes

- We destructure the event so we have local variables that we can use inside setCourse - if we don’t do that we might get an error as the event object is usually getting garbage collected and it’s not available in the function
- The [name] convention just binds the name of the changed event property to the property of the object that we pass to setCourse

Saving Data Action

- Before we connect the form to save data, we need to add API functionality to do so
- Let's add saveCourse(course) action to courseActions.js
- Note that you will need to dispatch different actions based on whether a course is being created or updated
- Hint: check if the course.id is set or not

Saving Data Reducer

- Next, of course, we need to update the courseReducer
- We need to add the two actions for create (we can update the old one) and update
- Update your actionTypes to include the new ones if you haven’t done so already

Saving Data - Dispatching

- We can now add saveCourse to our mapDispatchToProps and propTypes in the ManageCoursePage.js component
- Destructure the saveCourse and call it from a handleSave() function that you need to create
- Pass it as prop to: <CourseForm ... onSave={handleSave} />
- It should all work although we get no redirect on save

Adding More Routing Options

- Every component that was loaded by React Route gets a history object passed in on its props
- We can use that object to browse the history OR we can add new items to it and thus redirect:

```
saveCourse(course).then(() => {
    history.push("/courses");
});
```

- Make sure to destructure history up top and add it to propTypes

Adding More Routing Options

- We can also use the Redirect component from React Router to navigate to a new page
- Let's go to the CoursesPage.js and add the following on top of our render:
  {this.state.redirectToAddCoursePage && <Redirect to="/course" />}
- If the left expression is true, it will execute the redirect and ignore the rest, otherwise it will ignore the redirect and render the rest

Adding More Routing Options

- Now let's add this new variable as state to the component:

```
state = {
    redirectToAddCoursePage: false
};
```

- And we can also add a button above the courses list inside of the render which will change the default value from false to true when clicked thus trigger the redirect

Populate the Form

- One thing left to do: if we click on existing course we go to the form but it's empty - we need to pre-populate it
- The form is empty because we pass an empty course inside of mapStateToProps in ManageCoursePage.js
- We can parse the URL to figure out if we are loading a fresh form or one for an already existing course

Populate the Form

- mapStateToProps takes a second parameter called ownProps which lets you access the component's props
- We can use this to read the URL data injected on the props by the React Router
- Let's try to read the slug from the URL like this:

```
function mapStateToProps(state, ownProps) {
    const slug = ownProps.match.params.slug;
```

Populate the Form

- Based on the value of slug we want to determine whether to pass an empty course or an existing one:

```
const course =
    slug && state.courses.length > 0
        ? getCourseBySlug(state.courses, slug)
        : newCourse;
```

- We then pass that course value below instead of newCourse

Populate the Form

- We also need to create that function that finds a course from our list using a slug:

```
export function getCourseBySlug(courses, slug) {
    return courses.find(course ⇒ course.slug === slug) || null;
}
```

- Functions like this are called selectors as they select data from the Redux store
- You can place such functions in the reducers as well so you can reuse them in other components

Populate the Form

- Bonus problem: try to load the form directly from the browser's URL without clicking a link first
- It shouldn't populate because when we first load the form our courses (and authors) lists are empty so the check that we just wrote passes empty course to the form
- After that useEffect() fires up and gets the data but that's already too late for the form to populate

Populate the Form

- We can fix that by first adjusting our dependency array for the useEffect() as this:

```
}, [props.course]);
```

- Why does that work? The course prop is changed by the mapStateToProps once the state.courses changes

Populate the Form

- We still need to change our local course state so it propagates down to the rendered form so we can add the following in useEffect():

```
} else {
    setCourse({ ...props.course });
}
```

- This will make sure that when useEffect is triggered by the props.course change, we will update the local state as well

Optional Lab

- If time allows you can work on adding a Spinner to display while your API calls are working

Testing React Frameworks

- There are several options out there:

  Jest

  Mocha

  Jasmine

  Tape

  AVA

- Jest is created by Facebook, it’s popular, fast, easy to configure and offers excellent command line interface

Testing React Frameworks

- There are also testing helper libraries such as:

  React Test Utils - by Facebook but quite verbose

  Enzyme

  React Testing Library

- These libraries provide utilities as shallowRender (no DOM, fast and simple) or renderIntoDocument (DOM required and supports interactions - clicks, hovers, etc.)

Configure Jest

- Let’s configure Jest and run some tests
- Modify your package.json file to have a test script
- Add a “jest” section specifying to use jsdom environment to test and what files to ignore
- Acquire the tools/fileMock.js and tools/styleMock.js configuration files

Configure Jest

- You can now write your first test: src/index.test.js
- If you run npm test in cmd, Jest automatically looks for any files ending on test.js or spec.js and runs them
- We are now ready to get started and a good point to do so is the Jest snapshot testing

Snapshot Tests

- Let's create a test that checks that the label of our form Save button is saying "Saving..." if we have the saving property set to true
- We can create new file called components/courses/CourseForm.Snapshots.test.js with the name and location being arbitrary

Snapshot Tests

- Let’s add some imports - note here that having mock data for the development now allows us to use that data for the testing as well:

```
import React from "react";
import CourseForm from "./CourseForm";
import renderer from "react-test-renderer";
import { courses, authors } from "../../../tools/mockData";
```

Snapshot Tests

- We can then use the renderer function to create and render a CourseForm component
- Note that you can use {jest.fn()} to create an empty function to pass as a prop just so the render is complete without errors and you don’t need a specific implementation for that function

Snapshot Tests

- You can end the test with:

  `expect(tree).toMatchSnapshot();`

- Jest will create a snapshot upon component render and compare it with subsequent renders

- Snapshot testing is great if you want to catch accidental changes when the render output of your components changes

Snapshot Tests

- Run the test and see the output generated in the newly created __snapshots__ folder
- Create another test below the first which will test that the ‘Save’ label is displayed if saving is false
- Run the tests again and see how the second snapshot is appended to the first

Snapshot Tests

- To make sure our tests work, go into CourseForm.js and intentionally change the button label to something else
- If you run the tests again, they will fail because the newly created snapshot does not match the previous
- The output tells you to press `u` if you want to update them - this will replace the snapshots with the newly generated ones if the change was indeed intentional

Enzyme Testing

- To setup Enzyme we need a file: tools/testSetup.js
- It instantiates an adapter needed for the testing
- You also need to add this file under “setupFiles” property in the jest section of package.json
- Jest will run any files listed there

Enzyme Testing

- We can now create a file for our Enzyme tests:

  courses/CourseForm.Enzyme.test.js

- Enzyme can render components in two modes:

  - shallow - renders a single component
  - mount - renders component with children

Enzyme Testing

- This is optional but we can create a helper function that renders a CourseForm for us with default props that we can override by passing arguments to the function
- The function will use shallow rendering for now:

```
function renderCourseForm(args) { ...
```

Enzyme Testing

- We can now use Enzyme find function to browse the rendered components and assert findings
- We can write a test that ensures we have a rendered form with a header
- We can write tests to check the Save button labels as we did before

Enzyme Testing

- To demonstrate the mount rendering of Enzyme, we can render the ManageCoursePage.js parent component and simulate clicks on an empty form to see error messages

  NOTE: This will be the case if you implemented error messaging in your application

React Testing Library

- Optional: we can develop these tests in React Testing Library as well
- This library does not make difference between shallow and mount renders
- It doesn’t use expect but instead has embed methods that you can use to check output

Production Deployment

- Once we get to the point of building the code to be ready for production, we need to do some additional work to clean up the code
- We will build just 3 files - minified and combined JS bundle, minified and combined CSS file and an index.html file that includes both

Cleaning Up the Store

- The store has some stuff we don’t need in Prod such as the enhancers and the immutable state warning checks
- Rename the current store file to configureStore.dev.js
- Create new one configureStore.prod.js cleaning up the parts that we don’t want in Prod
- Create configureStore.js file to switch between the two

Production Deployment

- Next we need to duplicate the webpack.config.dev.js file and rename it to webpack.config.prod.js
- We need to additional imports up top the new file:

```
const MiniCssExtractPlugin =
require("mini-css-extract-plugin");
const webpackBundleAnalyzer =
require("webpack-bundle-analyzer");
```

Production Deployment

- Set the NODE_ENV to be production:

  process.env.NODE_ENV = “production”;

- Set the module.exports properties as well:

  mode: “production”, …

  devtool: “source-map”,

- We can remove the devServer section completely

Production Deployment

- Next we need to configure the plugins section to include few extra settings and configurations including CSS generator, bundle analyzer and others (copy / paste them)
- We also need to change the CSS settings in the module section
- Remember that this file loaders run from bottom up

Production Deployment

- We then need some scripts in the package.json file
- Script to run tests only once (ci - continuous integration)
- Script to delete the old build folder if present
- Script to build the new build folder using the new webpack configuration that we added

Production Deployment

- We can create a prebuild script to run the clean one automatically
- We can also setup scripts to start a server with the build file and even combine it with a postbuild script that will start our API server

Production Deployment

- We can finally run npm run build
- A window should open with a summary from our analyzer displaying the relative sizes of the created bundle file
- We can also test run our prod build, it should be at localhost:8080

Next Steps

- If you want to code on your own, here are some tasks:

Challenges

1. Author administration
2. Filter course list
3. Hide empty course list
4. Unsaved changes message
5. Enhance validation
6. Handle 404 on edit course
7. Show # courses in Header
8. Pagination
9. Sort course table
10. Revert abandoned changes

Questions?

Thank you!