React

Contents

React

Overview

- some links:
  - React App From Scratch
  - Error Boundaries
  - ChartJS in React
  - Server-Side Rendering
- made up of **components** for different parts of the application
  - can use react and *virtual* DOM to inject them into the webpage
  - only updates specfic parts of the DOM, makes react fast
- components look like HTML templates (usually jsx)
  - contain **state** (data or UI)

  - javascript for dynamic functionality
  - Babel allows jsx to be supported in the browser
- **component state**: describes the state of the data or the UI
  - state can be updated, eg. data updated or UI element hidden
  - whenever state is changed, the component is re-rendered to the DOM
    * eg. with dev tools, click events such as button pushed, etc.

```
// a sample component:
class App extends React.Component {
  state = {
    name: 'John',
    age: 30
  }
  handleClick(event) {
    console.log(event.target);
  }
  handleMouseOver(e) {
    console.log(e, e.pageX);
    console.log(this.state); // this is out of scope,
  }             // in js this keyword tied to when the function is called
  handleMouseOver = (e) => {
    this.setState({          // changing the state
      name: 'Smith',
      age: 25
    });
    console.log(this.state); // binding this is fixed with arrow functions
  }
  handleCopy(e) {
    console.log("something");
  }
  render() { // render a component, react funtion binds this to the component
    return ( // returning a jsx template
      <div className="app-content"> // only one root element at the top
        <h1>Our Title</h1>
        <p>{ Math.random() * 10 }</p> //dynamic javascript
        <p>My name is: { this.state.name } and I am { this.state.age }</p>
```

```
        <button onClick={ this.handleClick }>Click Me</button>
        <button onMouseOver={ this.handleMouseOver }>Hover Me</button>
        <p onCopy={ this.handleCopy }>e=mc^2</p>
      </div>
    )
  }
}

ReactDOM.render(<App />, document.getElementById('app'));
```

An example react form:

```
class App extends React.Component {
  state = {
    name: 'Ryu',
    age: 30
  }
  handleChange = (e) => {
    this.setState({
      name: e.target.value
    });
  }
  handleSubmit = (e) => {
    e.preventDefault();
    console.log('form submitted', this.state.name);
  }
  render() {
    return (
      <div className="app-content">
        <h1>My name is {this.state.name}</h1>
        <form onSubmit={this.handleSubmit}>
          <input type="text" onChange={this.handleChange}/>
          <button>Submit</button>
        </form>
      </div>
```

```
    )
  }
}
```

Other Topics

- elements of a list in react have to have a unique key prop
  - differentiates them so that React DOM knows which items to update
- *virtual DOM*: React is fast because pages do not have to be re-rendered after fetching from server
  - single page apps (SPAs) only have to be loaded from server once
  - then, the React Virtual DOM intercepts the request and re-renders or nests additional components
- *class-based componenets* are components that do have to save state
  - only class-based components have access to lifecycle hooks / functions
    - eg. componentDidUpdate(), componentDidMount(), constructor()
- *functional components* are UI components that do not have to save state (stateless)
- *React router* is a separate module
  - exports BrowserRouter, Route, Link, NavLink, Switch
  - BrowserRouter wraps the entire app component
  - Links and Navlinks replace anchor tags
  - handles routing (re-rendering the page) to different paths and components
    - also handles route parameters
- *higher order components* (HOC)
  - a wrapper for another function, extends component with extra information
  - implemented as a function that takes a component as an argument
    - returns another function with the wrapped component that takes in props
  - eg. components loaded up by the React Router, withRouter() HOC
    - gives history and match properties in the props
  - eg. a HOC that randomizes text colors

Redux

- redux, react-redux modules
- a *central* data store for all data
  - can be accessed by any element
  - easier to pass data between components
  - no longer store state in components
- components subscribe (listen) to changes in the redux
  - data is passed down through props
- there is a process for modifying data
  - component *dispatches* an action containing a *payload*
  - action is passed to a *reducer*, which then updates the central state
    “‘javascript const { createStore } = Redux;

/* initial, default state */ const initState = { title: ’ ’, data: [] }

function myreducer(state = initState, action) { if (action.type === ‘ADD_TODO’) { /* return entire new state (nondestructive!) */ *return { …state,* / all of previous state, but override data */ data: [...state.data, action.data] } } if (action.type === ‘CHANGE_TITLE’) { return { …state, title: action.title } } }

const store = createStore(myreducer)

/* subscribing to changes in the store */ store.subscribe(() => { console.log(‘state updated’); console.log(store.getState()); })

/* action is an object */ const dataAction = { type: ‘ADD_DATA’, data: 42, };

/* dispatch action to reducer */ store.dispatch(todoAction);

```
**Integrating with react**:
```javascript
...
import { createStore } from 'redux';
import { Provider } from 'react-redux';

/* may use several reducers in large application */
import rootReducer from './reducers/rootReducer';

const store = createStore(rootReducer);
ReactDOM.render(<Provider store={store}><App /></Provider>, ...);
```
```

```
/* mapping state to props in a component: */
import { connect } from 'react-redux';
...
const mapStateToProps = state => {
  return {
    data: state.data
  }
}

/* reusing original props when mapping */
const mapStateToProps = (state, ownProps) => {
  let id = ownProps.match.params.data_id;
  return {
    dataElement: state.data.find(elem => elem.id === id)
  }
}

const mapDispatchToProps = dispatch => {
  return {
    deletePost: id => dispatch({type: 'DELETE_DATA', id: id})
  }
}

/* connect RETURNS a HOC */
export default connect(mapStateToProps, mapDispatchToProps)(Home)
```

Context and Hooks

- **context API** allow for a shared state within a component tree
  - use a *context provider* to have access to the shared context
- **hooks** allow functional components to access those shared states
- alternative to Redux

Context API

Setting up a context and provider:

```
/* ThemeContext.js */
import React, { Component, createContext } from 'react';

export const ThemeContext = createContext();

class ThemeContextProvider extends Component {
  state = {
    isLightTheme: true,
    light: { syntax: '#555', ui: '#ddd', bg: '#eee'},
    dark: { syntax: '#ddd', ui: '#333', bg: '#555'}
  }
  render() {
    return (
      <ThemeContext.Provider value={{...this.state}}>
        /* wrapping up the children */
        {this.props.children}
      </ThemeContext.Provider>
    )
  }
}

export default ThemeContextProvider;

/* in another component */
import ThemeContextProvider from 'ThemeContext';
import AuthContextProvider from 'AuthContext';
...
return (
  <div>
    <ThemeContextProvider>
      /* can nest multiple contexts */
      <AuthContextProvider>
        ...
      </AuthContextProvider>
    </ThemeContextProvider>
```

```
  </div>
)
```

Consuming context from a provider: (in a class based component)

```
import { ThemeContext } from 'ThemeContext'
class Navbar extends Component {
  /* look up component tree for the provider of this context */
  /* can only consume one context in this way */
  static contextType = ThemeContext;

  render() {
    console.log(this.context);
    ...
  }
}
```

Consuming context using a consumer: (works in a functional component)

```
import { ThemeContext } from 'ThemeContext'
class Navbar extends Component {
  render() {
    return (
      /* can consume multiple contexts */
      <AuthContext.Consumer>{authContext => {
        <ThemeContext.Consumer>{themeContext => {
          return (
            console.log(themeContext);
            console.log(authContext);
            ...
          )
        }}</ThemeContext.Consumer>
      }}<AuthContext.Consumer>
    )
  }
```

```
}
```

Updating context data:

```
/* simply add another function to change the state */
class ThemeContextProvider extends Component {
  state = {
    isLightTheme: true,
    ...
  }
  toggleTheme = () => {
    this.setState({isLightTheme: !this.state.isLightTheme});
  }
  render() {
    return (
      <ThemeContext.Provider value={{...this.state, toggleTheme: this.toggleTheme}}>
        {this.props.children}
      </ThemeContext.Provider>
    )
  }
}
```

React Hooks

- *special* functions
- allow us to do additional things in functional components
  - eg. use state
  - useState(), useEffect(), useContext()

useState Hook:

```
import React, { useState } from 'react';

const SongList = () => {
  /* initial state, similar to state object */
  /* returns data, and function to edit */
```

```
  const [songs, setSongs] = useState([
    { title: ..., id: 1},
    { title: ..., id: 2},
    { title: ..., id: 3}
  ]);

  /* can have multiple states */
  const [age, setAge] = useState(20);

  const addSong = () => {
    setSongs([...songs, {...}]);
  }

  return (
    <div onClick={addSong}>
      console.log(songs);
      ...
    </div>
  );
}
```

useEffect Hook:

```
import React, { useEffect } from 'react';

const SongList = () => {
  ...
  /* runs every time component is re-rendered */
  /* emulates a life-cycle function in a class component */
  useEffect(() => {
    console.log('useEffect hook ran');
  })

  /* watches a change in a specific state */
  useEffect(() => {
```

```
    ...
  }, [songs])
  useEffect(() => {
    ...
  }, [age])
  ...
}
```

useContext Hook: (cleaner alternative to using consumer)

```
import React, { useContext } from 'react';

const BookList => {
  const {isLightTheme, light, dark} = useContext(ThemeContext);

  /* can consume multiple contexts */
  const {isAuth, toggleAuth}= useContext(AuthContext);

  render (
    console.log(isLightTheme);
    console.log(isAuth);
    ...
  )
}
```

Creating context with functional components:

```
import React, { createContext, useState } from 'react';

export const BookContext = createContext();

const BookContextProvider = props => {
  const [books, setBooks] = useState([
    {title: ..., id: ...},
    {title: ..., id: ...},
```

```
    {title: ..., id: ...}
  ]);
  return (
    <BookContext.Provider value={{books: books}}>
      {props.children}
    </BookContext.Provider>
  )
}
```

Reducers

- can use reducers to consolidate functions that act upon the state
  - contains all state manipulation logic
  - **dispatch** *actions* to a *reducer*
  - action object can also contain a payload / arguments
  - reducer checks the action type, update and return the state
- the updated state is passed to the provider value
- useReducer() instead of useState()