React

Contents

React 2

Overview 2

Redux 5

Hooks 8

Appendix 9

React

Overview

- a React application is made up of React **components**:
    - React has a *virtual* DOM representing the app that is then injected into the actual DOM of the browser
    - only specific parts of the DOM are updated depending on changes to state or props
        - this optimization makes React’s virtual DOM extremely fast
        - in addition, pages do not have to be reloaded after fetching new data from the server
        - such **single page apps (SPAs)** only have to be loaded from the server once:
            - then, the virtual DOM re-renders the application, updating, adding, and removing components dynamically as necessary
- a syntactical extension to JavaScript called **JSX** allows components to be written like HTML templates:
    - UI structure is written expressively using HTML-like JSX, while the JS allows for dynamic functionality
        - JSX allows for defining the component UI and component state together
        - JSX also easily supports dynamic content by embedding JS in the HTML-like template with curly braces
    - JSX is supported in the browser by *transpiling* it to browser-supported JS using transpilation technologies like Babel
- **component state** describes the current state of the component, whether data or UI related state:
    - state can be updated, eg. data from backend is updated or UI element toggled
    - whenever state is changed, the component is *re-rendered* to the DOM

A simple React component:

```
import React from 'react';

class App extends React.Component {
  state = {
    name: 'John',
    age: 30
  }

  handleClick(event) {...}

  handleMouseOver = (e) => { // lexical this-binding with arrow functions
    this.setState({
      name: 'Smith',
      age: 25
    });
  }

  render() { // render a component, react funtion binds this to the component
    return ( // returning a JSX template with one root element
      <div className="app-content">
        <h1>Title</h1>
        <p>{ Math.random() * 10 }</p>
        <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>
      </div>
    )
  }
}

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

- other important considerations:
  - elements of a list in React each have to have a *unique* key prop
    - differentiates them so that the React DOM knows which items to update
  - the `react-router` is a separate module that handles routing in the application:
    - handles pathing to different components, route parameters, etc.
    - the `BrowserRouter` component wraps the entire main `App` component
      - the `Link` and `Navlink` components replace regular anchor tags
    - the `withRouter` HOC gives components access to the history and match properties through the props
  - **higher order components (HOC)** are *wrappers* for another component that extend the component with extra information:
    - implemented as a function that takes a component as an argument

- returns another function that takes in props, and *contains* the wrapped component

Redux

- the Redux framework acts as a *central* data store for all data:
  - provided through the `redux` and `react-redux` modules
  - *pros*:
    - central store can be accessed by any component
    - no longer necessary to store all state in components
    - easier to pass data between components, instead of convoluted routing between components
  - *cons*:
    - adds significant complexity and considerations to the application design
- pattern for accessing data:
  - components *subscribe* ie. listen to changes in the store
    - data is then passed down through props of the subscribed component
  - to *modify* the store:
    - component *dispatches* an action containing a *payload*
    - action is passed to a *reducer*, which then updates the central state store

Vanilla Redux example (without React integration):

```
const { createStore } = Redux;

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 the store:
store.subscribe(() => {
  console.log('state updated');
  console.log(store.getState());
});

// dispatching actions:
const dataAction = {
  type: 'ADD_DATA',
  data: 42,
};
store.dispatch(dataAction);
```

Integrating Redux with React:

```
// inside root source file:
import { createStore } from 'redux';
import { Provider } from 'react-redux';
import rootReducer from './reducers/rootReducer';
import App from 'components/App';

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

// inside another component:
import React from 'react';
import { connect } from 'react-redux';

class Home extends React.Component {...}

// allows redux store to be accessed from the props
const mapStateToProps = (state, ownProps) => {
  let id = ownProps.match.params.data_id;
  return {
    dataElement: state.data.find(elem => elem.id === id)
  }
}
```

```
// allows redux store actions to be dispatched using the props
const mapDispatchToProps = dispatch => {
  return {
    deletePost: id => dispatch({type: 'DELETE_DATA', id: id})
  }
}

// connect returns a HOC that can be applied to the component
export default connect(mapStateToProps, mapDispatchToProps)(Home)
```

Hooks

- React **hooks** are *special* functions that provide additional functionality in functional components:
  - originally, functional components in React could not use state or have access to lifecycle methods
  - hooks allowed these operations to be done in functional components rather than in the more complex class-based components
  - eg. `useState` allows for state operations, `useEffect` gives access to lifecycle methods, `useContext` makes it easier to use the context API, etc.

Using the `useState` hook:

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

const SongList = () => {
  // initial state, similar to state object
  // returns data, and function to edit state
  const [songs, setSongs] = useState([
    { title: ..., id: 1},
    { title: ..., id: 2},
    { title: ..., id: 3}
  ]);
  const [age, setAge] = useState(20);

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

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

Using the `useEffect` hook:

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

const SongList = () => {
  ...

  // runs every time component is re-rendered
  // emulates lifecycle methods in a class component
  useEffect(() => {
    console.log('useEffect hook ran');
  })

  // runs on changes in a specific state
  useEffect(() => {
    ...
  }, [songs])
  useEffect(() => {
    ...
  }, [age])
}
```

Appendix

- links:
  - React App From Scratch
  - Error Boundaries
  - ChartJS in React
  - Server-Side Rendering