React

Thilan Tran

Summer 2019

Contents

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

Implementation

- under the hood, React revolves around four major processes:
    1. translating JSX blocks into lightweight version of DOM called VDOM
    2. rendering the VDOM by transforming it into regular DOM
    3. patching existing DOM using the `key` property
    4. handling the creting, lifecycle, and rendering of components
    - notes taken from Gooact clone
- React **elements** are a lightweight *object* representation of actual DOM:
    - holds important information like node type, attributes, children as properties
    - can be easily rendered in the future
    - *composing* elements together creats VDOM

JSX to elements:

```
const list = <ul className="list">
  <li className="list-item">One</li>
  <li className="list-item">Two</li>
</ul>;

// becomes transpiled to:
const list = createElement('ul', {className: 'list'},
  createElement('li', {className: 'list-item'}, 'One'),
  createElement('li', {className: 'list-item'}, 'Two'),
);

// simple createElement implementation, will get called at runtime:
const createElement = (type, props, ...children) => {
  if (props === null) props= {};
  return {type, props, children};
}

// creates the following VDOM:
const list = {
  "type": "ul",
  "props": {"className": "list"},
  "children": [
    {
      "type": "li",
      "props": {"className": "list-item"},
      "children": ["One"]
```

```
    },
    {
      "type": "li",
      "props": {"className": "list-item"},
      "children": ["Two"]
    }
  ]
}
```

- then, to *render* VDOM into actual visible DOM:
  - straightforward algorithm which traveerses down the VDOM
    - creates respective DOM element for each node
    - eg. using `createTextNode` or `createElement`

Rendering VDOM:

```
const render = (vdom, parent=null) => {
  const mount = parent ? (el => parent.appendChild(el)) : (el => el);
  if (typeof vdom == 'string' || typeof vdom == 'number') {
    // primitive plain-text nodes
    return mount(document.createTextNode(vdom));
  } else if (typeof vdom == 'boolean' || vdom === null) {
    return mount(document.createTextNode(''));
  } else if (typeof vdom == 'object' && typeof vdom.type == 'function') {
    // handled separately (later)
    return Component.render(vdom, parent);
  } else if (typeof vdom == 'object' && typeof vdom.type == 'string') {
    const dom = mount(document.createElement(vdom.type));
    for (const child of [].concat(...vdom.children))
      render(child, dom); // recurse
    for (const prop in vdom.props)
      setAttribute(dome, prop, vdom.props[prop]);
    return dom;
  } else {
    throw new Error(`Invalid VDOM: ${vdom}$`);
  }
}

// custom attributes in VDOM must be treated individually:
const setAttribute = (dom, key, value) => {
  if (typeof value == 'function' && key.startsWith('on')) {
    const eventType = key.slice(2).toLowerCase();
    dom.__reactHandlers = dom.__reactHandlers || {};
```

```
    dom.removeEventListener(eventType, dom.__reactHandlers[eventType]);
    dom.__reactHandlers[eventType] = value;
    dom.addEventListener(eventType, dom.__reactHandlers[eventType]);
  } else if (key === 'checked' || key === 'value' || key === 'className') {
    dom[key] = value;
  } else if (key === 'style' && typeof value === 'object') {
    Object.assign(dom.style, value);
  } else if (key === 'ref' && typeof value === 'function') {
    value(dom);
  } else if (key === 'key') {
    dom.__reactKey = value;
  } else if (typeof value !== 'object' && typeof value !== 'function') {
    dom.setAttribute(key, value);
  }
}
```

- **patching** involves reconciling existing DOM with a newly built VDOM tree to reflect changes:
  - a naive implementation would require a full render every time, ie. remove all existing nodes and re-render everything
  - instead, write a patching algorithm that requires less DOM modifications in general:
    1. build a fresh VDOM (much cheaper than rebuilding DOM)
    2. recursively compare it with existing DOM
    3. locate nodes that were added, removed, or changed
    4. patch them
  - however, comparison of two trees has $O(n^3)$ complexity:
    * instead use a heuristic $O(n)$ algorithm that makes two major assumptions:
      - two elements of different types will produce different trees
      - developer can hint at which child elements may be stabled using a `key` prop

Patching DOM with VDOM:

```
const patch = (don, vdom, parent=dom.parentNode) => {
  const replace = parent ? (el => (praent.replaceChild(el, dom)) && el) : (el => el);
  if (typeof vdom === 'object' && typeof vdom.type === 'function') {
    // handled separately (later)
    return Component.patch(dom, vdom, parent);
  } else if (typeof vdom !== 'object' && dom instanceof Text) {
    // compare text content, re-render if differ
    return dom.textContent !== vdom ? replace(render(vdom, parent)) : dom;
```

```
  } else if (typeof vdom == 'object' && dom instanceof Text) {
    // complex VDOM vs. text DOM
    return replace(render(vdom, parent));
  } else if (typeof vdom == 'object' && dom.nodeName != vdom.type.toUpperCase()) {
    // different VDOM vs. DOM type
    return replace(render(vdom, parent));
  } else if (typeof vdom == 'object' && dom.nodeName == vdom.type.toUpperCase()) {
    // recurse into children, same VDOM and DOM type
    const pool = {};
    const active = document.activeElement;
    [].concat(...dom.childNodes).map((child, idx) => {
      const key = child.__reactKey || `__index_${idx}$`;
      pool[key] = child;
    });
    [].concat(...vdom.children).map((child, idx) => {
      const key = child.props && child.props.key || `__index_${idx}$`;
      // match VDOM children to DOM nodes by key,
      // recursively patch if found, or render from scratch
      dom.appendChild(pool[key] ? patch(pool[key], child) : render(child, dom));
      delete pool[key];
    })
    // remove unpaired DOM nodes
    for (const key in pool) {
      const instance = pool[key].__reactInstance;
      if (instance) instance.componentWillUnmount();
      pool[key].remove();
    }
    for (const attr of dom.attributes) dom.removeAttribute(attr.name);
    for (const prop in vdom.props) settribute(dom, prop, vdom.props[prop]);
    active.focus(); // focus may be lost during repatching
    return dom;
  }
}
```

- finally **components** are essentially JavaScript functions that take in `props` and returns a set of elements:
  - can be implemented as a stateless function (before hooks), or derived class with internal state and set of methods and lifecycle methods

Example `Component` implementation:

```
class Component {
  constructor(props) {
```

```
    this.props = props || {};
    this.state = null;
  }

  static render(vdom, parent=null) {
    const props = Object.assign({}, vdom.props, {children: vdom.children});
    if (Component.isPrototypeOf(vdom.type)) {
      // class components must be instantiated
      const instance = new (vdom.type)(props);
      instance.componentWillMount();
      instance.base = render(instance.render(), parent);
      instance.base.__reactInstance = instance;
      instance.base.__reactKey = vdom.props.key;
      instance.componentDidMount();
      return instance.base;
    } else {
      // stateless, regular function
      return render(vdom.type(props), parent);
    }
  }

  static patch(dom, vdom, parent=dom.parentNode) {
    const props = Object.assign({}, vdom.props, {children: vdom.children});
    if (dom.__reactInstance && dom.__reactInstance.constructor == vdom.type) {
      dom.__reactInstance.componentWillReceiveProps(props);
      dom.__reactInstance.props = props;
      // patch differences *after* passing new props
      return patch(dom, dom.__reactInstance.render(), parent);
    } else if (Component.isPrototypeOf(vdom.type)) {
      const ndom = Component.render(vdom, parent);
      return parent ? (parent.replaceChild(ndom, dom) && ndom) : (ndom);
    } else if (!Component.isPrototypeOf(vdom.type)) {
      return patch(dom, vdom.type(props), parent);
    }
  }

  setState(nextState) {
    if (this.base && this.shouldComponentUpdate(this.props, nextState)) {
      const prevState = this.state;
      this.componentWillUpdate(this.props, nextState);
      this.state = nextState;
      patch(this.base, this.render());
```

```
      this.componentDidUpdate(this.props, prevState);
    } else {
      this.state = nextState;
    }
  }

  shouldComponentUpdate(nextProps, nextState) {
    return nextProps != this.props || nextState != this.state;
  }

  componentWillReceiveProps(nextProps) { return undefined; }
  componentWillUpdate(nextProps, nextState) { return undefined; }
  componentDidUpdate(prevProps, prevState) { return undefined; }
  componentWillMount() { return undefined; }
  componentDidMount() { return undefined; }
  componentWillUnmount() { return undefined; }
}
```

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

Design Patterns

- for conditional rendering, use short-circuit evaluation instead of ternaries:
  - eg. `test && <div>...</div>` vs. `test ? <div>...</div> : null`
  - for even more complex scenarios with many ternaries:
    * can use an IIFE and take advantage of `if..else` statements
    * use the upcoming `do` expression
    * or use early returns
- React batches updates and flushes them out together as a performance optimization:
  - thus state changes such as `setState` only creates a *pending* state transition
    * ie. the effects of calling state can be sync *or* async (although the method itself is always synchronous)
  - to fix, use the second parameter argument to `setState` to register a callback that will only run once the state has been updated
  - similarly, the first argument `setState` can also take a function that takes in the previous state as an argument
    * prevents any issues of retrieving old state
- dependency *injection* in React has to do with passing along dependencies or state to deeply nested components:
  - eg. centralized data store in Redux is one solution to this
  - can use a higher order component to inject the dependency
  - or use the React context API for a centralized data model
- can use *decorators* to decorate class components:
  - the same as passing the component into a function and creating a HOC

Simple switching component:

```
const pages = { home, about, user };
const Page = (props) => {
  const Handler = pages[props.page] || 404page;
  return <Handler {...props} />;
```

```
}
```

Reaching into a component using refs:

```
const Input extends Component {
  focus() { this.el.focus(); }
  render() {
    return <input ref={el => { this.el = el; }}/>;
  }
}

const SignIn extends Component {
  componentDidMount() { this.InputComponent.focus(); }
  render() {
    return (
      <div>
        <Input ref={comp => { this.InputComponent = comp; }}
      </div>
    );
  }
}
```