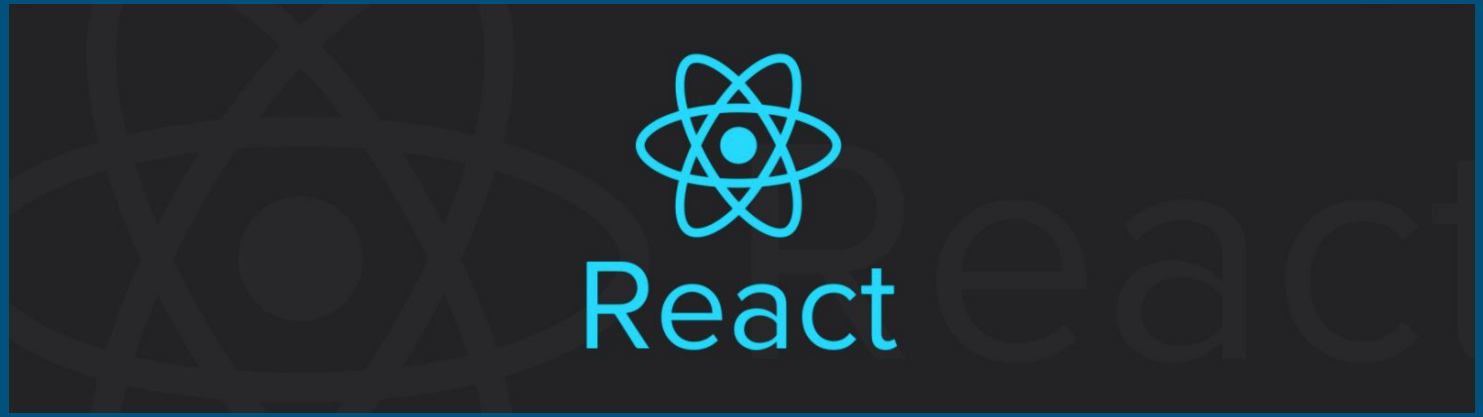
React Training Course

Beaconfire Solution

Topic 1. Intro to React

What is React?

- React is a Javascript ***library*** open sourced by Facebook for building ***UI***.
- It is used for developing Single Page Application.
- It is component-oriented.



UI library

- React only concerns about state (data) management and how to render that state.
- It doesn’t handle things like making http request, router, etc.
- We rely on lot of external libraries to achieve other functionalities. E.g. redux, axios

Component thinking in React

- Components are independent and reusable bits of code.
- Each component should only take a single responsibility in terms of the UI.
- Two types of components: functional based and class based

How many components should be build?

React history

- First created by developers at Facebook in 2012.
- Open-sourced in 2013.
- React 15.0 released to the public in 2016
- React 16.0 released to the public in 2017
- React 16.8 released in 2019, introduced React Hooks.
- React 17 released in 2020, but this release doesn’t contain changes of developer-facing API

Why react is so popular?

- React uses virtual DOM to improve its performance.
- Ability to reuse React components significantly saves time.
- One-direction data flow in React provides a stable code.
- An open-source Facebook library: constantly developing and open to the community. React has one of the strongest community around the world.

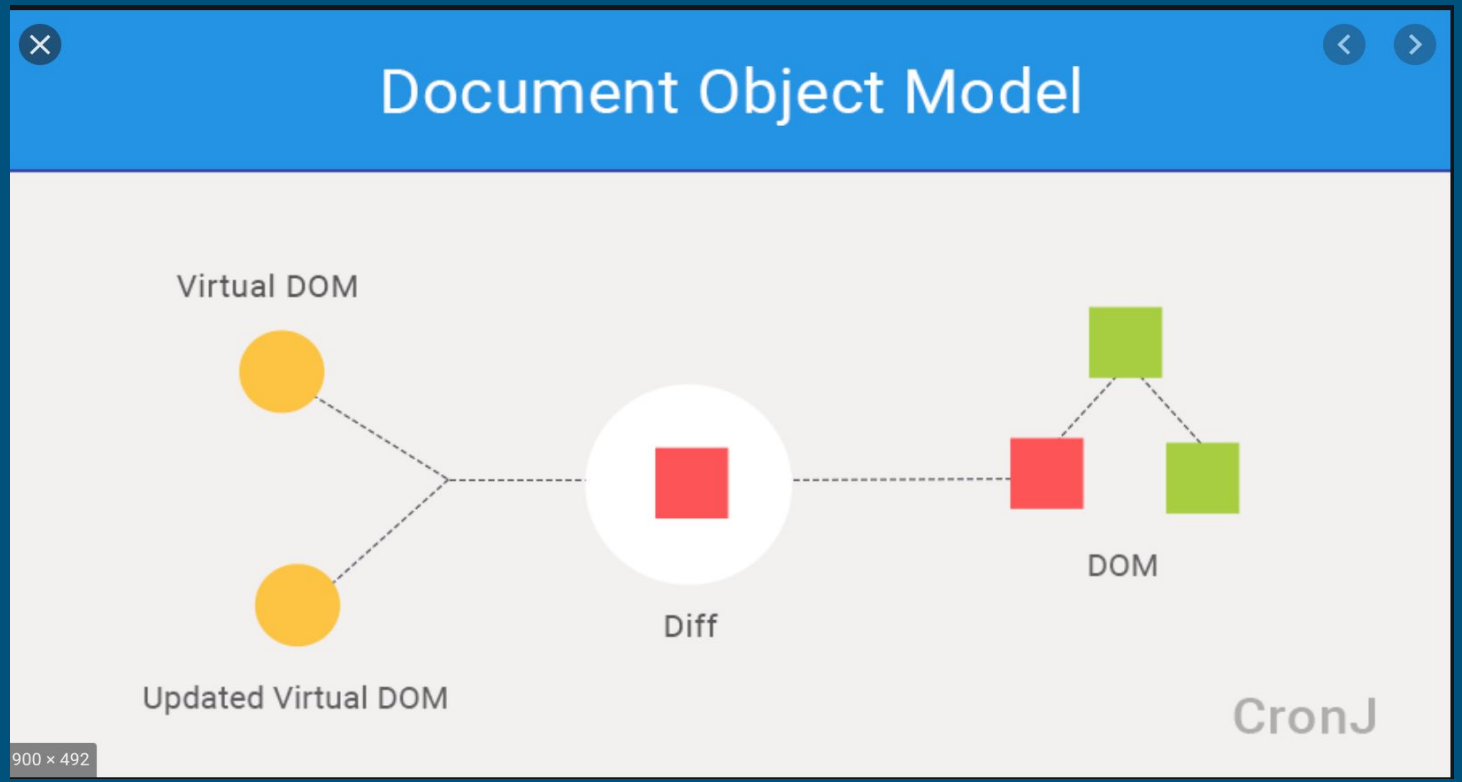
DOM

- DOM stands for Document Object Model. Elements of HTML become nodes in the DOM. It’s the data representations of a document on the web.
- Most javascript frameworks update DOM more than they have to.
- As an example, let’s say that you have a list that contains ten items. You check off the first item. Most JavaScript frameworks would rebuild the entire list. That’s ten times more work than necessary! Only one item changed, but the remaining nine get rebuilt exactly how they were before.

Virtual DOM

- In React, for every DOM object, there is a corresponding “virtual DOM object.” A virtual DOM object is a representation of a DOM object, like a lightweight copy.
- Manipulating the DOM is slow, but manipulating the virtual DOM is much faster.

How virtual DOM helps in React



Virtual DOM

In React, for every DOM object, there is a corresponding “virtual DOM object.” A virtual DOM object is a representation of a DOM object, like a lightweight copy.

A virtual DOM object has the same properties as a real DOM object, but it lacks the real thing’s power to directly change what’s on the screen.

Manipulating the DOM is slow. Manipulating the virtual DOM is much faster, because nothing gets drawn on screen. Think of manipulating the virtual DOM as editing a blueprint, as opposed to moving rooms in an actual house.

Virtual DOM Summary:

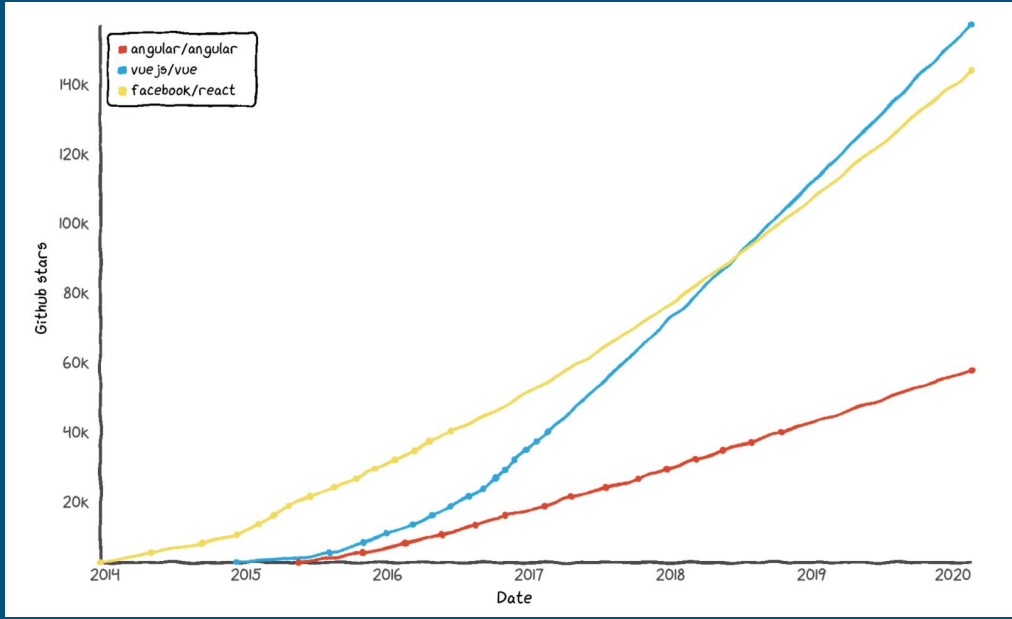
- When react renders a page, the entire virtual DOM gets updated.
- The virtual DOM gets compared to what it looked like before you updated it. React figures out which objects have changed.
- changed objects, and the changed objects only, get updated on the real DOM.
- Changes on the real DOM cause the screen to change.

The disadvantages of React

- ReactJS uses JSX. It’s a syntax extension, which allows mixing HTML with JavaScript, and it sometimes confusing.
- Poor documentation.
- Narrow focus on UI

React vs. Angular vs. Vue

Github star history:



React vs Angular

- AngularJS is an open-source JavaScript ***framework,*** which means it provides more functionalities for building web application instead of just a UI library. E.g. http, routing
- Angular provides two-way data-binding.
- AngularJS has several built-in services such as $http to make an XMLHttpRequest.
- Angular has MVC and dependency injection.

AngularJS	ReactJS	
Author	Google	Facebook Community
Language	JavaScript, HTML	JSX
Type	Open Source MVC Framework	Open Source JS Framework
Data-Binding	Bi-directional	Uni-directional
DOM	Regular DOM	Virtual DOM
App Architecture	MVC	Flux
Dependencies	It manages dependencies automatically.	It requires additional tools to manage dependencies.
Routing	It requires a template or controller to its router configuration, which has to be managed manually.	It doesn't handle routing but has a lot of modules for routing, eg., react-router.
Performance	Slow	Fast, due to virtual DOM.
Best For	It is best for single page applications that update a single view at a time.	It is best for single page applications that update multiple views at a time.

Prepare your environment

Required:

- VS code as IDE
- Node.JS

Recommended VS code plug-in:

- Prettier: auto-format code on save, highly recommended
- Bracket Pair Colorizer 2
- ES7 React/Redux/GraphQL/React-Native snippets

Recommended Chrome plug-in: React developer tools

Recap

- What’s react and its history
- DOM and Virtual DOM
- React vs. Angular vs. Vue
- React’s pros and cons

Topic 2. React fundamental

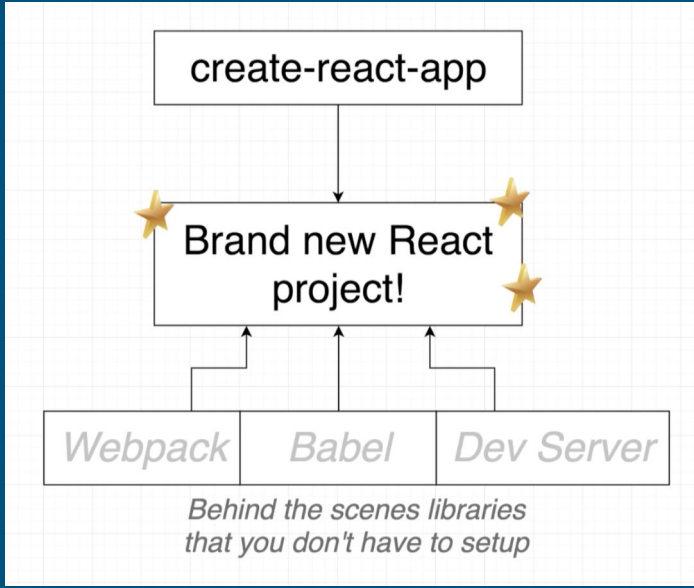
How to create a React app

- Make sure a recent Node.js version is installed
- npx create-react-app my-app
- cd my-app: cd into your project
- npm install: installing dependencies
- npm start: run your application

```
npm WARN tsutils@3.19.1 requires a peer of typescript@>=2.8.0 || >= 3.2.0-dev || >= 3.3.0-dev || >=
3.4.0-dev || >= 3.5.0-dev || >= 3.6.0-dev || >= 3.6.0-beta || >= 3.7.0-dev || >= 3.7.0-beta but none
 is installed. You must install peer dependencies yourself.

removed 1 package and audited 1938 packages in 14.191s

126 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities


Created git commit.

Success! Created my-app at /Users/xiangli/Desktop/BeaconFire/react-course/my-app
Inside that directory, you can run several commands:

  npm start
    Starts the development server.

  npm run build
```

React app structure

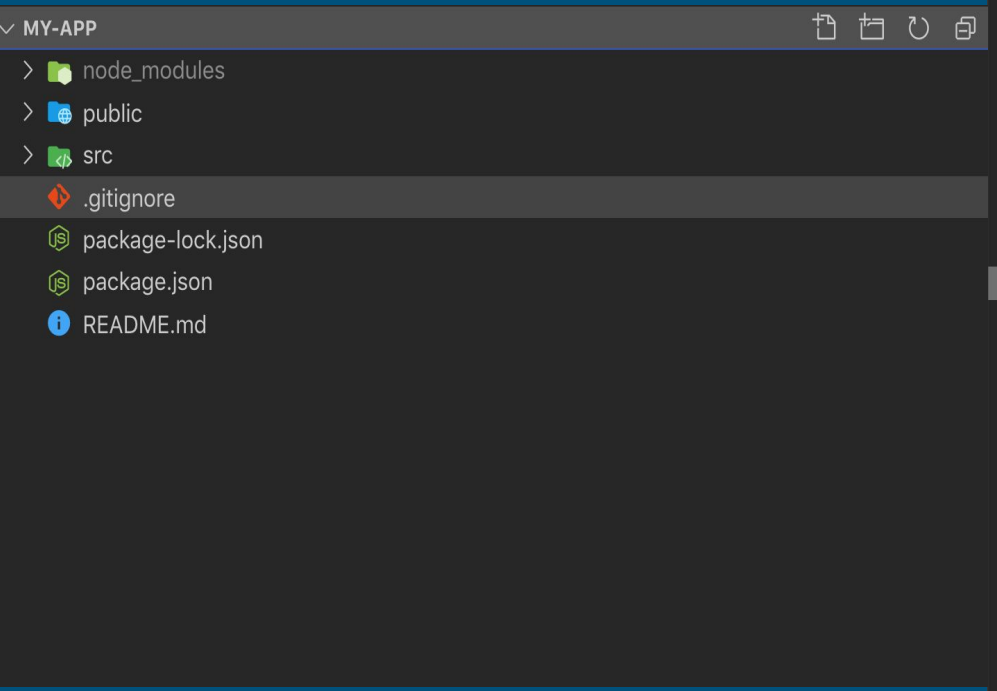
- Under a standard React application:
- Webpack
- Babel
- Dev server

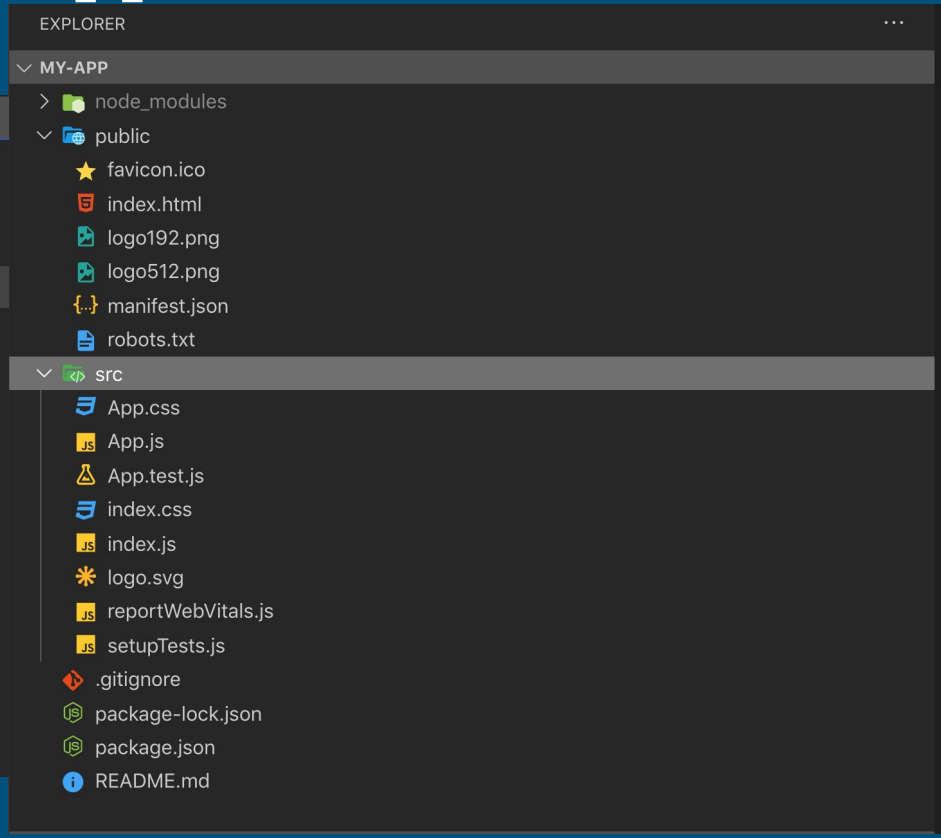


Babel

- Babel: Babel is a toolchain that is mainly used to convert ECMAScript 2015+ code into a backwards compatible version of JavaScript in current and older browsers or environments.

File structure of a React app





File structure of a React app

- Node_modules
- public
- Src: we usually creates a components folder under src for storing our components.
- package.json

File structure of a React app

- Package.json:
- Dependency defines the version of required dependencies in the application
- Scripts defines the npm commands you could run

package.json — my-app

EXPLORER

index.html | package.json

MY-APP

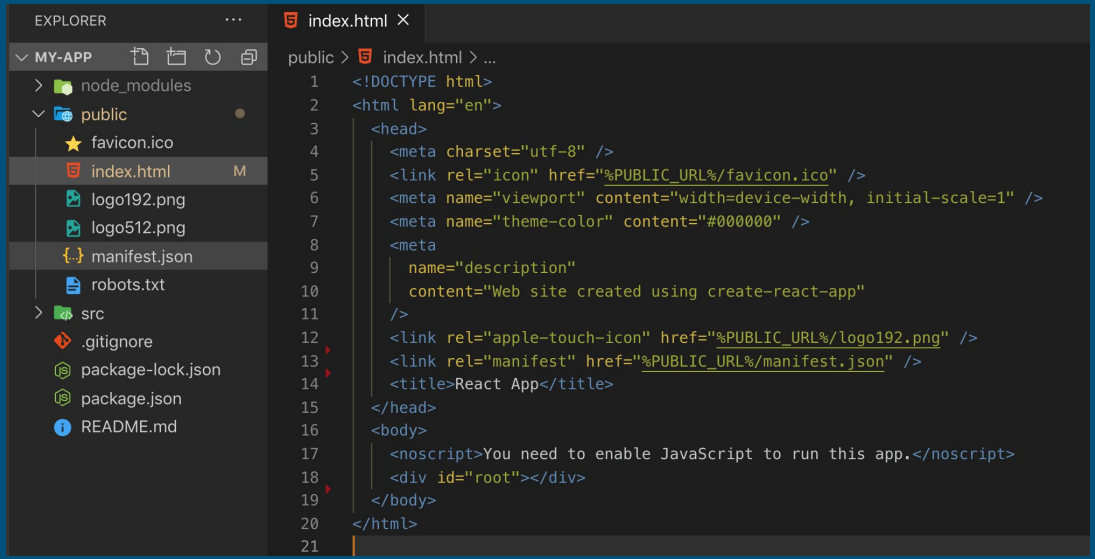
- node_modules
- public
- src
- .gitignore
- package-lock.json
- package.json
- README.md

package.json > ...

```
{
  "name": "my-app",
  "version": "0.1.0",
  "private": true,
  "dependencies": {
    "@testing-library/jest-dom": "^5.11.8",
    "@testing-library/react": "^11.2.3",
    "@testing-library/user-event": "^12.6.0",
    "react": "^17.0.1",
    "react-dom": "^17.0.1",
    "react-scripts": "4.0.1",
    "web-vitals": "^0.2.4"
  },
  Debug
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test",
    "eject": "react-scripts eject"
  },
  "eslintConfig": {
    "extends": [
      "react-app",
      "react-app/jest"
    ]
  },
  "browserslist": {
    "production": [
      ">0.2%",
      "not dead",
      "not op_mini all"
    ],
    "development": [
      "last 1 chrome version",
      "last 1 firefox version",
      "last 1 safari version"
    ]
  }
}
```

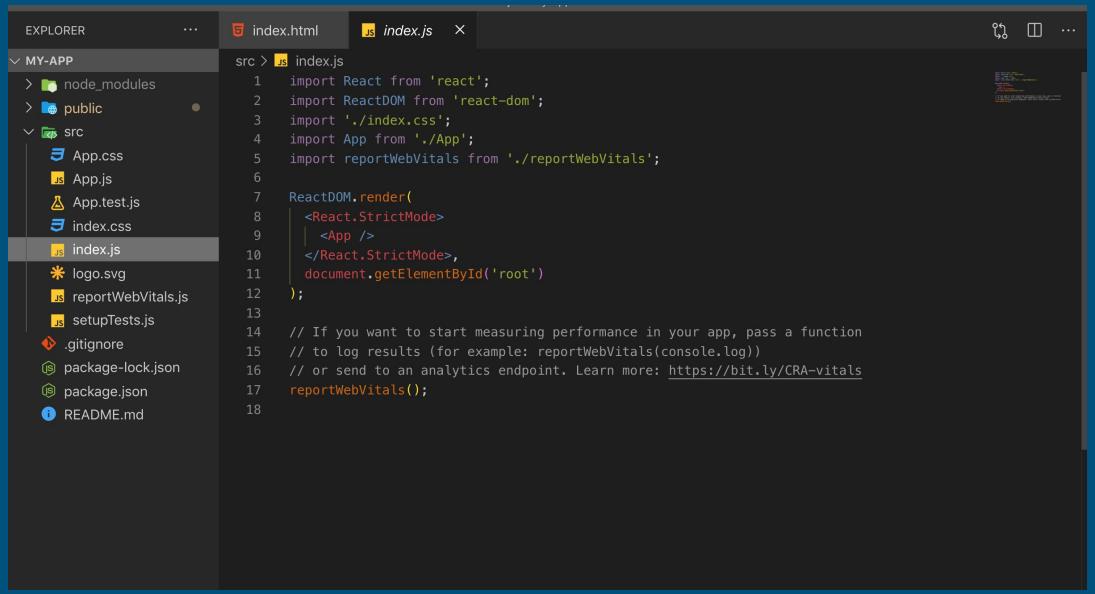
How React works?

1. Starts with index.html under public folder. This is the only html file in a react application, which makes it a SPA.



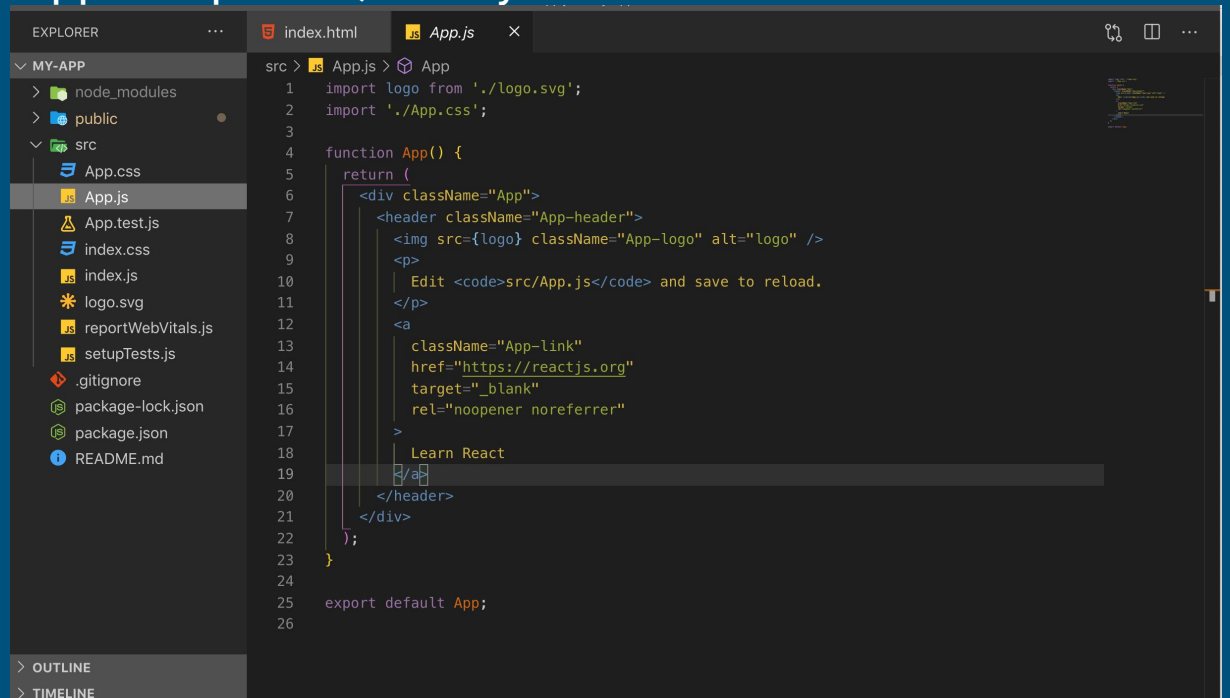
How React works?

2. Index.js under src folder retrieve the div element with ID “root” in the html file. Renders the App component



How React works?

3. In the App component, write your actual content.



JSX: “const element = <h1>Hello, world!</h1>;”

- Consider this line of code or the code we saw in the previous slides.
- The syntax is neither a string nor HTML. It is JSX.
- Although JSX looks like HTML, we are actually dealing with a way to write JavaScript. Under the hood, JSX returned by React components is compiled into JavaScript.
- The compiling is handled by Babel

JSX

- JSX is "XML-like", which means that every tag needs to be closed.
-
 is legal in HTML, but illegal in JSX. You have to write <br />.

Embedding Expressions in JSX

- You can put any valid JavaScript expression inside the curly braces in JSX.
- In the example below, we declare a variable called name and then use it inside JSX by wrapping it in curly braces.

```
const name = 'Josh Perez';
const element = <h1>Hello, {name}</h1>;

ReactDOM.render(
  element,
  document.getElementById('root')
);
```

Create components

- Functional based component

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}
```

Create components

- class based component
- These two types of components are equivalent
- We starts with class-based component

```
class Welcome extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>;
  }
}
```

State

- State is the single source of truth in a react component. All the data being display should come from state.
- State cannot be changed directly
- Must always use setState

index.html | App.js | Header.js | Input.js

src > components > Header.js > ...

```
1   import React, { Component } from 'react';
2
3   export default class Header extends Component {
4     state = { name: "Fan"}
5
6     render() {
7       return (
8         <div>
9           <h1>My name is: {this.state.name}</h1>
10        </div>
11      );
12    }
13  }
14
```

Set state

- In class-based components, we can use this.setState to update state.
- this.setState({ name : ‘Test’ })
- React will update UI when calling set state

Set state

- Do Not Modify State Directly
- this.state.comment = 'Hello'; (WRONG)

Set state

- State Updates May Be Asynchronous
- React may batch multiple setState() calls into a single update for performance.
- Solution:
  - Callback functions
  - 2nd forms of set state

```
// Correct
this.setState((state, props) => ({
  counter: state.counter + props.increment
}));
```

Props (property)

- React supports one-way data flow
- Props is used to pass data from parent component to child component
- In the parent component App, passed a prop called “name”, with value comes from the state.

Props

```
export default class App extends Component {
  state = { name: "Fan"}
  render() {
    return (
      <div className="App">
        <Header name={this.state.name} />
      </div>
    );
  }
}
```

Props

- In the child component Header, we can access the props through this.props.xxx, because props is passed as argument to the child component

```
export default class Header extends Component {
  render() {
    return (
      <div>
        <h1>My name is: {this.props.name}</h1>
      </div>
    );
  }
}
```

Props

- Sometimes we need to abstract props to state

```
export default class Header extends Component {
  state = {
    name: this.props.name
  };
  render() {
    return (
      <div>
        <h1>My name is: {this.state.name}</h1>
      </div>
    );
  }
}
```

Event handler

- We set the value of the button's onClick attribute to be a reference to the handleIncreaseClick function defined in the code.

```
export default class App extends Component {
  state = { count: 0 };

  handleIncreaseClick = () => {
    this.setState({ count: this.state.count + 1 });
  };
  render() {
    return (
      <div className="App">
        <div>{this.state.count}</div>
        <button onClick={() => this.handleIncreaseClick()}>increase</button>
      </div>
    );
  }
}
```

Event handler

- The event handler function can also be defined directly in the value assignment of the onClick-attribute

```
export default class App extends Component {
  state = { count: 0 };

  // handleIncreaseClick = () => {
  //   this.setState({ count: this.state.count + 1 });
  // };
  render() {
    return (
      <div className="App">
        <div>{this.state.count}</div>
        <button onClick={() => this.setState({ count: this.state.count + 1 })}>
          increase
        </button>
      </div>
    );
  }
}
```

Event Handler

- What if we do this:

```
export default class App extends Component {
  state = { count: 0 };

  // handleIncreaseClick = () => {
  //   this.setState({ count: this.state.count + 1 });
  // };
  render() {
    return (
      <div className="App">
        <div>{this.state.count}</div>
        <button onClick={this.setState({ count: this.state.count + 1 })}>
          increase
        </button>
      </div>
    );
  }
}
```

Event handler

Error: Maximum update depth exceeded. This can happen when a component repeatedly calls setState inside componentWillUpdate or componentDidUpdate. React limits the number of nested updates to prevent infinite loops.

```
▶ 4 stack frames were collapsed.

App.render
src/App.js:14

  11 | return (
  12 |   <div className="App">
  13 |     <div>{this.state.count}</div>
> 14 |     <button onClick={this.setState({ count: this.state.count + 1 })}>
  15 | ^     increase
  16 |     </button>
  17 |   </div>

View compiled
```

Event handler

- An event handler is supposed to be either a ***function*** or a ***function reference***
-
```
onClick={this.setState({ count: this.state.count + 1 })}
```
- In this particular situation, it is actually a function call.
- When react renders the component, it executes the function call setState
- This will cause the component to be re-rendered
- React will execute the SetState function call again….

Render collection

- Suppose you have a list of to dos, how to display them?
- Todo: ['finish hw', 'watching video', 'playing games']

Render collection

```
render() {
  const item = this.state.todo.map((element, index) => {
    return <li key={index}>{element}</li>;
  });

  return (
    <div>
      <ul>{item}</ul>
    </div>
  );
}
```

Render collection

```
render() {
  return (
    <div>
      <ul>
        {this.state.todo.map((element, index) => {
          return <li key={index}>{element}</li>;
        })}
      </ul>
    </div>
  );
}
```

Key attribute

- React uses the key attributes of objects in an array to determine how to update the view generated by a component when the component is re-rendered.
- Using index as key is anti-pattern, learn more about it

Conditional rendering

```
render() {
  const item = this.state.todo.map((element, index) => {
    return <li key={index}>{element}</li>;
  });

  return <div>{this.state.todo.length > 0 && <ul>{item}</ul>}</div>;
}
```

Topic 3. Forms and HTTP request

Forms

- Let’s image we want to add more to do through user input
- Prevent default
- How do we access the value in input?

```
export default class List extends Component {
  state = {
    todo: []
  };

  addTodo = (e) => {
    e.preventDefault();
    console.log(e);
  };

  render() {
    return (
      <div className="List">
        <form onSubmit={(e) => this.addTodo(e)}>
          <input />
          <button type="submit">save</button>
        </form>
      </div>
    );
  }
}
```

Controlled components

- State should be the only source of truth

```
export default class List extends Component {
  state = {
    todo: [],
    newTodo: ''
  };

  addTodo = (e) => {
    e.preventDefault();
    // do API call
  };

  handleInputChange = (e) => {
    this.setState({ newTodo: e.target.value });
  };

  render() {
    return (
      <div className="List">
        <form onSubmit={(e) => this.addTodo(e)}>
          <input
            value={this.state.newTodo}
            onChange={(e) => this.handleInputChange(e)}
          />
          <button type="submit">save</button>
        </form>
      </div>
    );
  }
}
```

Controlled Components

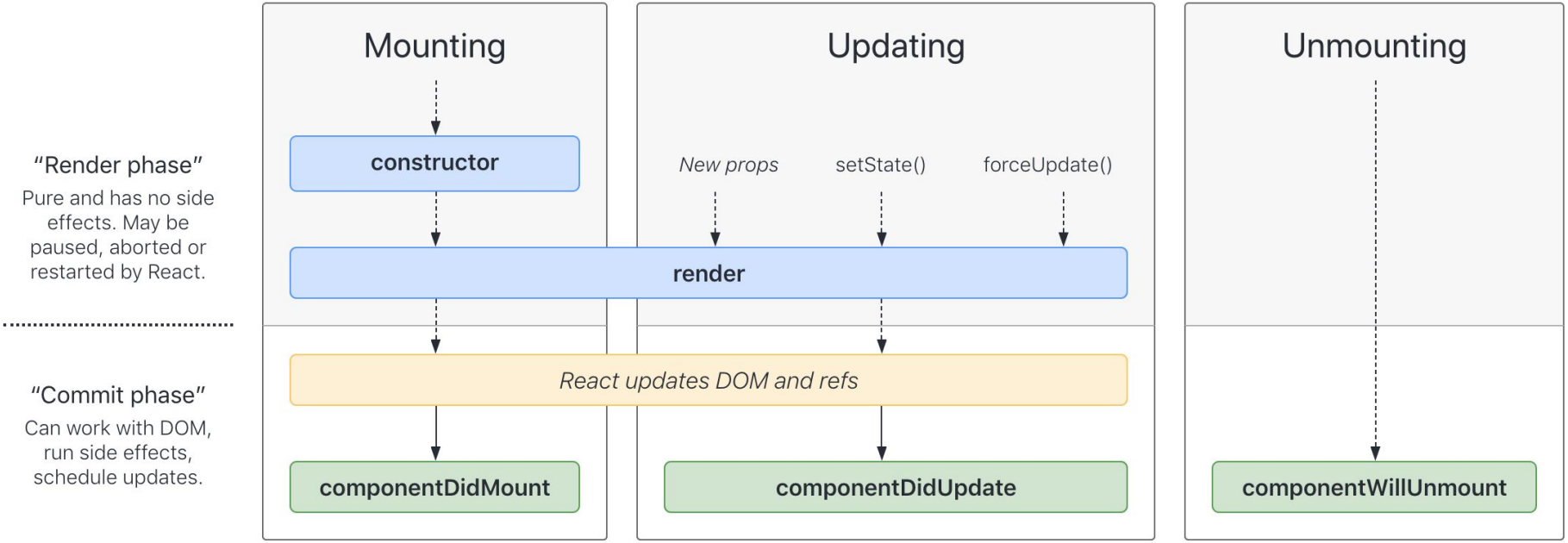
```
addTodo = (e) => {
  e.preventDefault();
  this.setState({
    newTodo: '',
    todo: [...this.state.todo, this.state.newTodo]
  });
};

handleInputChange = (e) => {
  this.setState({ newTodo: e.target.value });
};

render() {
  return (
    <div className="List">
      <form onSubmit={(e) => this.addTodo(e)}>
        <input
          value={this.state.newTodo}
          onChange={(e) => this.handleInputChange(e)}
        />
        <button type="submit">save</button>
      </form>
      <ul>
        {this.state.todo.map((item) => {
          return <li>{item}</li>;
        })}
```

Communicate with remote server

- Instead of store posts in memory, we are going to get the data from a remote server
- Where should we call the API?

React lifecycle methods

- Each component in React has a lifecycle which you can monitor and manipulate during its three main phases.
- You can think of it ask a series methods from the birth of component until its death.
- The three phases are: ***Mounting***, ***Updating***, and ***Unmounting***.


"Render phase"
Pure and has no side effects. May be paused, aborted or restarted by React.
"Commit phase"
Can work with DOM, run side effects, schedule updates.
Mounting
Updating
Unmounting
constructor
New props
setState()
forceUpdate()
render
React updates DOM and refs
componentDidMount
componentDidUpdate
componentWillUnmount

render

- It renders the component to the UI
- It happens during the mounting and updating of your component.
- React requires that your render() is pure. No setState inside render method

componentDidMount

- componentDidMount() is called as soon as the component is mounted and ready.
- Calling the setState() here will update state and cause another rendering
- but it will happen before the browser updates the UI. This is to ensure that the user will not see any UI updates with the double rendering.

componentDidUpdate

- This lifecycle method is invoked as soon as the updating happens. It could be changes in state or props
- The most common use case for the componentDidUpdate() method is updating the DOM in response to prop or state changes.

componentWillUnmount

- As the name suggests this lifecycle method is called just before the component is unmounted and destroyed.
- Typically do some cleanup actions, including clearing timers, cancelling api calls, or clearing any caches in storage.

AJAX

```
componentDidMount() {
  fetch('https://jsonplaceholder.typicode.com/todos')
    .then((res) => res.json())
    .then(
      (result) => {
        console.log(result);
        this.setState({ remoteTodo: result });
      },
      (error) => {
        console.log(error);
      }
    );
}
```

Axios

```
componentDidMount() {
  axios
    .get('https://jsonplaceholder.typicode.com/todos')
    .then((resp) => {
      console.log(resp);
      this.setState({ remoteTodo: resp.data });
    })
    .catch((error) => {
      console.log(error);
    });
}
```

Adding styles

- Traditional way
  - import '../css/Header.css';
- Inline style through the style property:

```
render() {
  return (
    <div className="header" style={{ textDecoration: 'none' }}>
      <h1>A simple to do app made by {this.state.name}:</h1>
    </div>
  );
}
```

Topic 4. Functional-based Component

Hooks

- Hooks are a new addition in React 16.8. They let you use state and other React features without writing a class.

useState hook

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

function App() {
  const [count, setCount] = useState(0);

  const increaseCount = () => {
    setCount(count + 1);
  };

  return (
    <div className="App">
      {count}
      <button onClick={() => increaseCount()}>incrase</button>
    </div>
  );
}
```

Effect hooks

- The Effect Hook lets you perform side effects in function components.
- Data fetching, setting up a subscription, and manually changing the DOM in React components are all examples of side effects.
- By default, effects run after every completed render, but you can choose to fire it only when certain values have changed.

Effect hooks

```
const hook = () => {
  console.log('effect')
  axios
    .get('http://localhost:3001/notes')
    .then(response => {
      console.log('promise fulfilled')
      setNotes(response.data)
    })
}

useEffect(hook, [])
```

Effect hooks

- The second parameter of useEffect is used to specify how often the effect is run.
- If the second parameter is an empty array [], then the effect is only run along with the first render of the component.