# Week 14 Study Guide

## Table of Contents



## Getting Started with React - EOD Lecture Notes

### Explain how React uses a tree data structure called the "virtual DOM" to model the DOM

The Virtual DOM is an in-memory tree representation of the browser's Document Object Model. React's philosophy is to interact with the Virtual DOM instead of the regular DOM for developer ease and performance.

By abstracting the key concepts from the DOM, React is able to expose additional tooling and functionality increasing developer ease.

By trading off the additional memory requirements of the Virtual DOM, React is able to optimize for efficient subtree comparisons, resulting in fewer, simpler updates to the less efficient DOM. The result of these tradeoffs is improved performance.

Supporting performant Virtual DOM comparisons informs a key aspect of the React philosophy:

> In our experience, thinking about how the UI should look at any given moment, rather than how to change it over time, eliminates a whole class of bugs.

### Use React.createElement to create virtual DOM nodes

Every Virtual DOM Node is eventually constructed using the `React.createElement` function.

`createElement` accepts three arguments, which represent the three key components of every Virtual DOM Node, it's `type`, it's `props`, and it's `children` (i.e. it's contents).

```
React.createElement(
    type, // Either an html element "h1", _or_ a React Component `Component`
```

```
    props, // These are _static_ properties && any html attributes
    children // An array of Virtual DOM Nodes, or a string of content
    );
```

## Use ReactDOM.render to have React render your virtual DOM nodes into the actual Web page

`ReactDOM.render` is a simple function which accepts 2 arguments: what to render and where to render it:

```
ReactDOM.render(
    component, // The react component to render
    target // the browser DOM node to render it to
    );
```

## Use JSX to create virtual DOM nodes

## Describe how JSX transforms into React.createElement calls

JSX is a special format to let you construct virtual DOM nodes using familiar HTML-like syntax. You can put the JSX directly into your `.js` files, however you must run the JSX through a pre-compiler like Babel in order for the browser to understand it.

```
const Clock = (props) => {
    return (
        <div>
            <h1>Clock</h1>
            <div className='clock'>
                <p>
                    <span>
                        Time:
                    </span>
                    <span>
                        {props.hours}:{props.minutes}:{props.seconds}
                    </span>
                </p>
            </div>
        </div>
    );
}
```

Here we initialize a `Clock` component using JSX instead of `React.createElement`.

Using Babel this code is compiled to a series of recursively nested `createElement` calls:

```
const Clock = (props) => {
    return React.createElement(
        "div",
        null,
        [
            React.createElement("h1", null, "Clock"),
            React.createElement("div", {className: "clock"}, [
                React.createElement("p", null, [
                    React.createElement("span", null, "Time:"),
                    React.createElement("span", props, "{props.hours}:{props.minutes}:{props.seconds}")
                ])
            ])
        ]
    );
}
```

JSX is a convenience syntax, but not magic.

## Use Array#map to create an array of virtual DOM nodes while specifying a unique key for each created virtual DOM node

Since Array.prototype.map is often used to change an array of values into another array of values of equal length, it's a perfect way to convert an array of *data* or *state* into a list of React Components.

There's one *gotcha* when doing this, though. In order for React to keep track of which components in the list it's rendered and which ones it needs to re-render when the data changes, you **MUST** provide a **unique** `key` attribute to the rendered Components.

```
const rootDiv = document.getElementById("root");

const StarTrekCard = (props) => {
    return (
        <div className="card">
            <div className="card-image">
                <figure className="image">
                    <img src={props.imgUrl} />
                </figure>
            </div>
            <div className="card-content">
                <div className="content">
                    {props.content}
                </div>
            </div>
        </div>
    );
}

const StarTrekCardDeck = (props) => {
    return (
        <div>
            // Here we are mapping the cards into <StarTrekCard> components
            {props.cards.map(card =>
                <StarTrekCard
                    imgUrl={card.imgUrl}
                    content={card.content}
```

```
                    key={card.name} />
            )}
        </div>
    );
}

const cards = [
  {
    name: "Martok",
    imgUrl: "http://guide.fleetops.net/images/avatars/martok.png",
    content: "Ferocious Klingon"
  },
  {
    name: "Mijural",
    imgUrl: "http://guide.fleetops.net/images/avatars/mijural.png",
    content: "Shrike Class Romulan"
  }
  ];

ReactDOM.render(
    <StarTrekCardDeck cards={cards} />,
    rootDiv
)
```

# React Class Components Objectives

## Create a simple React application by removing items and content from a project generated by the Create React App default template

To generate a React project

```
npx create-react-app my-app-name
```

### Files you can delete

Remove the following files from the public folder:

- favicon.ico
- robots.txt
- logo192.png
- logo512.png
- manifest.json

You can replace all the content of index.html with the boilerplate HTML generated by the html:5 command. Don't forget to add a div inside body with an id of "root".

Remove the following files from the src folder:

- App.css
- App.test.js
- logo.svg
- serviceWorker.js
- setupTests.js

## Create a simple React application using a custom Create React App template

To use aA's custom template:

```
npx create-react-app my-app-name --template @appacademy/simple
```

## Create a React component using ES2015 class syntax

```
import React from 'react'

class Hello extends React.Component {
    constructor() {
        super();
        this.state = {
            name: this.pickRandomName()
        }
    }

    pickRandomName() {
        const randomNum = Math.floor(Math.random() * 3);
        const names = ['LeBron', 'Messi', 'Serena']
        return names[randomNum];
    }

    changeName = () => {
        this.setState({
            name: this.pickRandomName()
        });
    }
    render() {
        return (
            <>
                <h1>Hello {this.state.name}!</h1>
                <button onClick={this.changeName}>Change name</button>
            </>
        )
    }
}

export default Hello;
```

## Describe when it's appropriate to use a class component

Since the release of React Hooks (which we will learn about next week) any advantages for class based components have evaporated, so this Learning

Objective does not make a lot of sense anymore and you can safely ignore it.

## Initialize and update state within a class component

Initialize the state inside the component's constructor:

```
this.state = {
    name: '',
    email: '',
    password: ''
}
```

The update can happen inside any instance method except for render():

```
handleEmail = (e) => {
    this.setState({
        email: e.target.value
    });
}
```

## Provide default values for a class component's props

```
import React from 'react'

class Books extends React.Component {
    render() {
        return (
            <ul>
                {this.props.books.map(book => {
                    return (
                        <li key={book.title}>
                            {book.title} by {book.author}
                        </li>
                    );
                })}
            </ul>
        )
    }
}

Books.defaultProps = {
    books: [
        {title: "Don Quixote", author: "Miguel De Cervantes"},
        {title: "Pedro Paramo", author: "Juan Rulfo"}
    ]
};

export default Books;
```

## Add event listeners to elements

```
render() {
    return (
        <form onSubmit={this.handleSubmit}>
            <input type="text" value={this.state.value}/>
            <button>Submit!</button>
        </form>
    )
}
```

## Prevent event default behavior

```
handleSubmit = e => {
    e.preventDefault();
    const inputVal = this.state.value;
    // this.props.submitForm is an example of a method passed
    // as part of the props. It is not a built-in method.
    // The software engineer writing this code would need to write it.
    // Tipically the method will take the response and attach
    // it to the body of an AJAX request
    this.props.submitForm(inputVal);
}
```

## Safely use the this keyword within event handlers

We can use the experimental syntax that consist of using a fat arrow function:

```
changeEmail = e => {
    this.setState({
        email: e.target.value
    });
}
```

We can also bind the method inside the constructor:

```
constructor() {
    super();

    this.state = { email: ''};

    this.changeEmail = this.changeEmail.bind(this);
}
```

## Describe what the React SyntheticEvent object is and the role it plays in handling events

The SyntheticEvent object mimics the characteristics of the event passed to the callback in an event handler but adding more data to it.

These are two common uses:

```
e.preventDefault();
e.stopPropagation();
```

## Create a React class component containing a simple form

```
class Simpleform extends React.Component {
    constructor() {
        super();

        this.state = {
            response: ''
        }
    }

    handleResponse = e => {
        this.setState({
            response: e.target.value
        });
    }

    handleSubmit = e => {
        e.preventDefault();
        this.props.submitForm(this.state.response);
    }

    render() {
        return (
            <form onSubmit={this.handleSubmit}>
                <input
                    onChange={this.handleResponse}
                    type="text"
                    value={this.state.response}
                />
                <button>Submit</button>
            </form>
        );
    }
}
```

## Define a single event handler method to handle onChange events for multiple "input" elements

```
onChange = (e) => {
    const { name, value } = e.target;
    this.setState({ [name]: value });
}

render() {
    return(
        <form onSubmit={this.onSubmit}>
            <input
                onChange={this.onChange}
                name='email'
                type="text"
                value={this.state.email}
            />
            <input
                onChange={this.onChange}
                name='password'
                type="text"
                value={this.state.password}
            />
            <button>Submit</button>
        </form>
    );
}
```

## Add a "textarea" element to a form

The textarea element in React uses a value attribute instead of inner content. It is handled the same way you would handle an input element. It will be a self-closing tag.

```
<textarea value={this.state.text} onChange={this.onChange}/>
```

## Add a "select" element to a form

```
<select name='timezone' onChange={this.onChange} value={this.state.timezone}>
    <option>PST</option>
    <option>CST</option>
    <option>EST</option>
</select>
```

## Implement form validations

Create a validate method and invoke it on the submit handler method.

```
validate(spiritAnimal) {
    const validationErrors = [];

    if (!spiritAnimal) {
        validationErrors.push('Please provide a spirit animal');
    }

    return validationErrors;
}

onSubmit = e => {
    e.preventDefault();
```

```
    const { spiritAnimal } = this.state;

    const validationErrors = this.validate(spiritAnimal);

    if (validationErrors.length > 0) {
        this.setState({ validationErrors });
    } else {
        this.props.submitForm(spiritAnimal);

        this.setState({ spiritAnimal: ''});
    }
}
```

## Describe the lifecycle of a React component

A React component usually will go through three stages: mounting, updating, and unmounting.

## Recall that the commonly used component lifecycle methods include componentDidMount, componentDidUpdate, and componentWillUnmount

When a React component is added to the virtual DOM:

- The constructor method is called.
- The render method is called.
- The DOM gets updated.
- The componentDidMount method is called.

When a React component is updated:

- The render method is called.
- The DOM gets updated.
- The componentDidUpdate method is called.

Just before a React component gets removed from the virtual DOM, the componentWillUnmount method is called.

## Use the componentDidMount component lifecycle method to fetch data from an API

You should not fetch data from within the constructor. Use componentDidMount instead.

```
componentDidMount() {
    fetch(url)
        .then(response => response.json())
        .then(data => this.setState({ movieInfo: data }));
}
```

## Utilize official documentation to gain an understanding of how new technology works

React Docs

# React Router Objectives

## Use the react-router-dom package to set up React Router in your applications

```
npm install --save react-router-dom@^5.1.2
```

```
import { BrowserRouter } from 'react-router-dom`;

const App = () => {
  return (
    <BrowserRouter>
      <div> // Note that BrowserRouter can only have a single child Component!
      // Include components here you want to be controlled by the Router, including
      // the <Route> tag
      </div>
    </BrowserRouter>
  );
};
```

## Create routes using the `<Route>` component from the react-router-dom package

the `<Route>` Component takes the following arguments:

- `path` - A pattern to match the URL in the browser. You can include placeholders by prefixing them with a `:` character.

  ```
  <Route path='/users/:userId'/>
  <Route path='/users/:userId/stories/:storyId'/>
  ```

- `component` - You usually pass a single component to this attribute. When the path is matched, this component will be rendered and passed props containing `history` `location` and `match` properties.

  Example: When someone visits `/users/1` this will render the Component `UserProfile`

  ```
  <Route path='/users/:userId' component={UserProfile}/>
  ```

- `render` - An alternative to `component` this will cause a component to render, you can pass this an arrow function that renders multiple components. Usually

we use this so we can pass custom props to the component. Using the `component` tag doesn't let us do this.

Example: Assume we want to pass an extra prop called "token" to the user Profile page
We can do it like this, but we need to pass the regular props from `<Route>` down as well by using the spread operator : `{...props}` .

```
<Route
  path='/users/:userId'
  render={props => <UserProfile {...props} token={token} }/>
```

## Generate navigation links with the and components from the react-router-dom package

`<Link>` simple renders an anchor tag `<a href=""></a>` that when clicked, changes the `window.location.href` to the url, which triggers `BrowserRouter` to parse the url and look for matching `<Route>` components.

```
<Link to="/">App</Link>
<Link to="/users">Users</Link>
<Link to="/users/1">Andrew's Profile</Link>
```

`<NavLink>` is just like `<Link>` except that it has an optional `activeClassName` and `activeStyle` props that will set a css class or css styles when the `NavLink` is the currently selected hyperlink.

```
<NavLink to="/">App</NavLink>
<NavLink activeClassName="red" to="/users">Users</NavLink>
<NavLink activeClassName="blue" to="/hello">Hello</NavLink>
<NavLink activeClassName="green" to="/users/1">Andrew's Profile</NavLink>
<NavLink to="/" onClick={handleClick}>App with click handler</NavLink>
```

## Create `<Switch>` routes and manage the order of rendered components

By default React Router matches paths for the `<Route>` component by treating it as a *prefix*. This means all routes here will match when the url is `/`

```
<Route path='/' ... >
<Route path='/users' ... >
<Route path='/users/:userId' ... >
```

To change this behavior so it doesn't match all three but only the exact matching route you can either change all three to include the `exact` prop set to true, or you can wrap them in a `<Switch>` component which causes only one of the routes inside the to be rendered at any given time. By combining `exact` and `<Switch>` you can solve many complex routing problems.

Remember the first route to match in a `<Switch>` is the winner!

```
<Switch>
  <Route path="/users/:userId" component={(props) => <Profile users={users} {...props} />} />
  <Route exact path="/users" render={() => <Users users={users} />} />
  <Route path="/hello" render={() => <h1>Hello!</h1>} />
  <Route exact path="/" component={App} />
  <Route render={() => <h1>404: Page not found</h1>} />
</Switch>
```

## Use the React Router match prop to access router parameters

Given the following `<Route>` :

```
<Route path='/users/:userId/stories/:storyId' component={UserStories}/>
```

If the url were `/users/1/storties/2` then `props.match.params` inside of `UserStories` would look like this:

```
console.log(props.match.params);
// Will log this
// {
//   userId: 1,
//   storyId: 2
// }
```

## Use the React Router history prop to programmatically change the browser's URL

`props.history` is one of the props that a `<Route>` sends to your component.

There are two functions on `props.history`

- *push* - Pushes a new url onto the history stack. The user can use the Back button in the browser to go back to the previous url.
- *replace* - Replaces the current url with the new one. The back button in the browser won't work to take you to the previous url.

```
// Pushing a new URL (and adding to the end of history stack):
const handleClick = () => this.props.history.push('/some/url');

// Replacing the current URL (won't be tracked in history stack):
const redirect = () => this.props.history.replace('/some/other/url');
```

## Redirect users by using the component in a route

To redirect the user to a different URL (Like say after they've logged in)
use the `<Redirect>` component.

```
<Redirect to="/">
```

## Describe what nested routes are and how to create them

Nested routes are when we use a `<Route>` component inside a component
that hasn't been rendered yet instead of creating them in your top level
component directly under `<BrowserRouter>`.

This allows us to keep those routes close to the components they render, and it
means we are dynamically adding routes on the fly.

For instance inside the UserProfile component you might render more routes to the
sub components of that component.

```
// Destructure `match` prop
const UserProfile = ({ match: { url, path, params }) => {

  // Custom call to database to fetch a user by a user ID.
  const user = fetchUser(params.userId);
  const { name, id } = user;

  return (
    <div>
      <h1>Welcome to the profile of {name}!</h1>

      {/* Replaced `/users/${id}` URL with `props.match.url` */}
      <Link to={`${url}/posts`}>{name}'s Posts</Link>
      <Link to={`${url}/photos`}>{name}'s Photos</Link>

      {/* Replaced `/users/:userId` path with `props.match.path` */}
      <Route path={`${path}/posts`} component={UserPosts} />
      <Route path={`${path}/photos`} component={UserPhotos} />
    </div>}
  );
};
```

# React Builds Objectives

## Describe what frontend builds are and why they're needed

Frontend Builds are where we run a series of programs on the code we write to
*transpile* (convert) and *bundle* our javascript code.

Frontend Builds are needed for the following reasons:

1. To convert ( `Transpile` ) code written in newer JavaScript or JSX into code
   that is compatible with certain browsers, even if those browsers do not
   support those features of the language yet.
2. To compress your front end assets (JS, CSS, etc) into a few large files that can
   be loaded into the browser more efficiently. This can use the processes of `Minification` , `Bundling` and `Tree Shaking` to accomplish this.
3. To check your code for errors ( `Linting` )

## Describe at a high level what happens in a Create React App when you run `npm start`

Create React App runs a development web server when you run `npm start` .

This webserver is controlled by a piece of software called WebPack. The webserver
transpiles and bundles your JS, CSS and other files in memory, and then serves it up
to the browser. By doing it in emory it can support many nice features such as
hot reloading of the code, and nice error reporting in the browser.

Here's the steps that occur:

- Environment variables are loaded
- The list of browsers to support are checked
- The configured HTTP port is checked to ensure that it's available;
- The application compiler is configured and created;
- webpack-dev-server is started;
- webpack-dev-server compiles your application;
- The index.html file is loaded into the browser; and
- A file watcher is started to watch your files, waiting for changes.

## Prepare to deploy a React application into a production environment

### Set up Environment variables

You can setup the environment variables in a `.env` file just like we did
with express, the only difference is, they must be prefixed with `REACT_APP_`
to be available inside the WebPack build process.

### Set up the browser list

You can setup the list of supported browsers in your package.json file. This will
inform Webpack on which rules it should use when transpiling your code.

```
{
  "browserslist": {
    "production": [
      ">0.2%",
      "not dead",
      "not op_mini all"
    ],
    "development": [
      "last 1 chrome version",
      "last 1 firefox version",
      "last 1 safari version"
    ]
  }
}
```

### Build the app

When you get ready to deploy a React application into production, you will run the `npm run build` command, which will use Webpack to bundle and transpile your code into a folder called `build`.

You should not check this folder into source control!

You will deploy this folder onto any webserver (Express, Nginx, Apache etc.). The webserver you use doesn't matter, but it needs to be configured so it serves up the `index.html` file no matter which URL the user visits. This is because React is a Single Page Application (SPA) framework.

## React Context Objectives

Context is a way of connecting two components together so they can both access the same state. Think of context like a pipe that connects a Context.Provider to a Context.Consumer. You will still need a regular React component to store the state in, but by adding that state to the Context.Provider and pulling it out with a Context.Consumer you can allow multiple components to access the state without having to pass the piece of state down via nested props.

## Use Context to share and manage global information within a React application

### Creating the context

You create the Context by using `createContext`. We can put this in it's own module and export the created context object so we can use it in multiple components…

```
// PupContext.js
import { createContext } from 'react';

const PupContext = createContext(); // You can pass arguments here to setup
                                    // a default context if you need to.
export default PupContext;
```

## Create a wrapper component with Context.Provider to set a component's default context

### Recommended example (Using a 'Provider' component)

Using a custom provider component that can wrap any element, means our provider is reusable. We use `this.props.children` to make it so we can wrap any component with this.

```
import PupContext from './PupContext';
// PuppyProvider.js
class PuppyProvider extends React.Component {
  constructor() {
    super();
    this.state = {
      puppyType: speedy,
    };
  }

  render() {
    return (
      <PupContext.Provider value={this.state}>
        {/* this allows us to wrap this component around any other component */}
        {this.props.children}
      </PupContext.Provider>
    );
  }
}

export default PuppyProvider;
```

In `App.js` we can simply wrap this provider around any part of the app that needs to access the context. The context will be available to any component in the component tree under the Provider. (Any descendant)

```
// App.js
import React from 'react';
import PuppyProvider from './PuppyProvider';
import banana from './pups/banana-pup.jpg'
import sleepy from './pups/sleepy-pup.jpg'
import speedy from './pups/speedy-pup.jpg'

const App = ({ puppyType }) => (
  <PuppyProvider>
    <div id="app">
      <Puppy/>
    </div>
```

```
    </PuppyProvider>
  );
```

## Create a wrapper component with Context.Consumer to share the global context through render props

…Somewhere deep inside of the `<Puppy>` component might live a `<PuppyImage>` component…

There are two ways to access the puppyType stored in our Provider. With the static contextType variable, or by wrapping our component with `<PupContext.Consumer>`.

This is by far the simplest way to access context.

### `static contextType` and `this.context`

```
import React from 'react';
import PupContext from './PupContext';

class PuppyImage extends React.Component {
  static contextType = PupContext

  render() {
    <img src={this.context.puppyType}/>
  }
}
```

### Wrapping a `<PupContext.Consumer>` around our component

You must pass a `Consumer` tag a *function* that receives the value stored in the context.

```
import React from 'react';
import PupContext from './PupContext';

class PuppyImage extends React.Component {
  static contextType = PupContext

  render() {
    return (
      <PupContext.Consumer> {
        puppy => { // This is the function the consunmer will call
                   // so we can get the data from the context provider.
          return (
            <img src={puppy.puppyType}/>
          );
        }
      }
      </PupContext.Consumer>
    );
  }
}
```

This method is handy if you need to access multiple contexts in a single component… Imagine we also need the UserContext to get details on the user of the application as well so we can put a caption under the image.

```
import React from 'react';
import PupContext from './PupContext';
import UserContext from './UserContext';

class PuppyImage extends React.Component {
  static contextType = PupContext

  render() {
    return (
      <UserContext.Consumer> {
        user => {
          <PupContext.Consumer> {
            puppy => {
              return (
                <figure>
                  <img src={value.puppyType}/>
                  <figcaption>{`by ${user.name}`}</figcaption>
                </figure>
              );
            }
          }
          </PupContext.Consumer>
        }
      }
      </UserContext.Consumer>
    );
  }
}
```

As you can see this can get ugly in a hurry, but don't worry, next week we will be learning React Hooks, which will make this much easier and cleaner.

## Create and pass a method through Context to update the global state from a nested component

If you take our example PuppyProvider component from earlier, you can see how we can just add a function which updates the state of the PuppyProvider into the PuppyProvider's state.

```
import PupContext from './PupContext';
// PuppyProvider.js
class PuppyProvider extends React.Component {
  constructor() {
    super();
    this.state = {
      puppyType: speedy,
      // We can just add the function to the state
```

```
        // So we can call it from the Consumer
        setPuppyType: this.setPuppyType
      };
    }

    // This function updates the puppyType in the state
    setPuppyType = (puppyType) => {
      this.setState({
        puppyType
      });
    }

    render() {
      return (
        <PupContext.Provider value={this.state}>
          {this.props.children}
        </PupContext.Provider>
      );
    }
  }

  export default PuppyProvider;
```

Since it's now in the Provider's state, it's accessible from the Consumer.

Imagine we have a form that lets you pick a new puppy type.
We just have to call the function we added to the context to set the
puppy type when we submit the form.

```js
import React from 'react';
import PupContext from './PupContext';

class PuppyTypeForm extends React.Component {
static contextType = PupContext;

// Bonus: We can setup the initial state this way
// instead of using a constructor! Fancy!
state = {
chosenPuppyType: ''
}

updateChosenPuppyType = e => {
this.setState({
chosenPuppyType = e.target.value
});
}

handleSubmit = e => {
e.preventDefault();
// This is the magic, we reach into the context and set
// the puppy type by calling the function we stored there.
this.context.setPuppyType(this.state.chosenPuppyType);
}

render() {
return (

{this.chosenPuppyType}/ Update Puppy Type

)
}
}
```