React 101
Codecademy
Advanced React

Style

Inline Styles

An inline style is a style that's written as an attribute, like this:

```
<h1 style={{ color: 'red' }}>Hello world</h1>
```

Notice the double curly braces. What are those for?

The outer curly braces inject JS into JSX. They say, "everything between us should be read as JS, not JSX".

The inner curly braces create a JS object literal. They make this a valid JS object:

```
{ color: 'red' }
```

If you inject an object literal into JSX, and your entire injection is only that object literal, then you will end up with double curly braces. There's nothing unusual about how they work, but they look funny and can be confusing.

Make a Style Object Variable

That's all needed to apply basic styles in React!

But there is one problem with this approach that is it becomes obnoxious if you want to use more than just a few styles. An alternative that's often nicer is to store a style object in a variable into JSX.

```
const style = {
      color: 'darkcyan',
      background: 'mintcream'
};
```

Defining a variable named style in the top-level scope would be extremely bad idea in JS environments! But in React it's totally fine.

Every file is invisible to other file, except what is exposed via module.exports.

Style Name Syntax

In regular JS, style names are written in hyphenated-lowercase:

```
const styles = {
   'margin-top': '20px',
   'background-color': 'green'
}
```

In React, those same names are instead written in camelCase:

```
const styles = {
   marginTop: '20px',
   backgroundColor: 'green'
}
```

This has zero effect on style property values, only style property names.

Style Value Syntax

In regular JS, styles values are almost always strings. Even if a style value is numeric, we usually have to write it as a string so that you can specify a unit.

In React, if we write a style value as a number then the unit 'px' is assumed.

If we want a font size of 30px, we can write: { fontSize: 30}

If you want to use units other than 'px', we can use a string: { fontSize: "2em" }

Specifying "px" with a string will still work, although its redundant.

Share styles across multiple components

In order to reuse styles across several different components we can keep them in separate JS files, which will export the styles via export keyword and then can be imported into the components.

Container Components from Presentational Components

Separate Container Components from Presentational Components

When building a React Application, we soon realize that one component has too many responsibilities, but how do we know when we have reached that point?

Separating container components from presentational components helps to answer that question. It shows when it might be a good time to divide a component into smaller components. It also shows how to perform that division.

This component renders a photo carousel and it does it perfectly well! And yet, it has a problem: it does too much stuff.

```
import React from 'react';
import ReactDOM from 'react-dom';

const GUINEAPATHS = [
  'https://content.codecademy.com/courses/React/react_photo-guineapig-1.jpg',
  'https://content.codecademy.com/courses/React/react_photo-guineapig-2.jpg',
  'https://content.codecademy.com/courses/React/react_photo-guineapig-3.jpg',
  'https://content.codecademy.com/courses/React/react_photo-guineapig-4.jpg'
];

export class GuineaPigs extends React.Component {
  constructor(props) {
    super(props);

    this.state = { currentGP: 0 };
```

```
    this.interval = null;

    this.nextGP = this.nextGP.bind(this);
  }

  nextGP() {
    let current = this.state.currentGP;
    let next = ++current % GUINEAPATHS.length;
    this.setState({ currentGP: next });
  }

  componentDidMount() {
    this.interval = setInterval(this.nextGP, 5000);
  }

  componentWillUnmount() {
    clearInterval(this.interval);
  }

  render() {
    let src = GUINEAPATHS[this.state.currentGP];
    return (
      <div>
        <h1>Cute Guinea Pigs</h1>
        <img src={src} />
      </div>
    );
  }
}

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

Create Container Component

Separating container components from presentational components is a popular React programming pattern.

If a component has to have state, make calculations based on props, or manage any other complex logic, then that component shouldn’t also have to render HTML-like JSX.

The functional part of a component can be separated into a container component.

Separate Presentational Component

The presentational component's only job is to contain HTML-like JSX. It should be an exported component and will not render itself because a presentational component will always get rendered by a container component.

GuineaPigs.js

```
import React from 'react';

export class GuineaPigs extends React.Component {
  render() {
    let src = this.props.src;
    return (
      <div>
        <h1>Cute Guinea Pigs</h1>
        <img src={src} />
      </div>
    );
  }
}
```

GuineaPigsContainer.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import { GuineaPigs } from '../components/GuineaPigs';

const GUINEAPATHS = [
  'https://content.codecademy.com/courses/React/react_photo-guineapig-1.jpg',
  'https://content.codecademy.com/courses/React/react_photo-guineapig-2.jpg',
  'https://content.codecademy.com/courses/React/react_photo-guineapig-3.jpg',
  'https://content.codecademy.com/courses/React/react_photo-guineapig-4.jpg'
];

class GuineaPigsContainer extends React.Component {
  constructor(props) {
    super(props);

    this.state = { currentGP: 0 };
```

```
    this.interval = null;

    this.nextGP = this.nextGP.bind(this);
  }

  nextGP() {
    let current = this.state.currentGP;
    let next = ++current % GUINEAPATHS.length;
    this.setState({ currentGP: next });
  }

  componentDidMount() {
    this.interval = setInterval(this.nextGP, 5000);
  }

  componentWillUnmount() {
    clearInterval(this.interval);
  }

  render() {
    const src = GUINEAPATHS[this.state.currentGP];
    return <GuineaPigs src={src} />;
  }
}

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

In this programming pattern, the container component does the work of figuring out what to display. The presentational component does the work of actually displaying it. If a component does a significant amount of work in both areas, then that's a sign that you should use this pattern!

PropTypes

propTypes

propTypes are useful for two reasons. The first reason is prop validation.

Validation can ensure that our props are doing what they're supposed to be doing. If props are missing, or if they're present but they aren't what we're expecting, then a warning will print in the console.

This is useful, but reason #2 is more useful: documentation.

Documenting props makes it easier to glance at a file and quickly understand the component class inside. When we have a lot of files, this is a huge benefit.

Apply PropTypes

If a component class expects a prop, then we can use propTypes for that component class!

In order to start using propTypes, we need to import the 'prop-types' library.

```
import PropTypes from 'prop-types';
```

Then, we can declare propTypes as a static property for our component after the component has been defined.

The second step is to add properties to the propTypes object. For each prop that our component class expects to receive, there can be one property on our propTypes object.

Add properties to PropTypes

What are the properties on propTypes supposed to be, exactly?

The name of each property in propTypes should be the name of an expected prop.

The value of each property in propTypes should fit this pattern:

```
PropTypes.expected_data_type_goes_here
```

Each property on the propTypes object is called a propType.

If we add .isRequired to a propType, then we will get a console warning if that prop isn't sent.

PropTypes in Function Components

Writing propTypes for function components:

```
// Usual way:
class Example extends React.component{

}

Example.propTypes = {

};
...

// Function component way:
const Example = (props) => {
  // ummm ??????
}
```

To write propTypes for a function component, define a propType object as a property of the function component itself.

```
const Example = (props) => {
  return <h1>{props.message}</h1>;
}

Example.propTypes = {
  message: PropTypes.string.isRequired
};
```

React Forms

React Forms

Think about how forms work in a typical, non-React environment. A user types something into a form's input fields, and the server doesn't know about it. The server remains clueless until the user hits a submit button, which sends all of the form's data over to the server simultaneously.

In React, as in many other JS environments, this is not the best way of doing things.

The problem is the period of time during which a form thinks that a user has typed one thing, but the server thinks that the user has typed a different thing. What if, during that time, a third part of the website needs to know what a user has typed? It could ask the form or the server and get two different answers. In a complex JS app with many moving, interdependent parts, this kind of conflict can easily lead to problems.

In a React form, we want the server to know about every new character or deletion, as soon as it happens. That way, our screen will always be in sync with the rest of our application.

Input onChange

A traditional form doesn't update the server until a user hits submit, but we want to update the server any time a user enters or deletes any character.

```
import React from 'react';
import ReactDOM from 'react-dom';

export class Input extends React.Component {
  constructor(props) {
    super(props);

    this.state = { userInput: '' };

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

```
  handleUserInput(e) {
    this.setState({userInput: e.target.value});
  }

  render() {
    return (
      <div>
        <input type="text" onChange={this.handleUserInput} value={this.state.u
serInput} />
        <h1>{this.state.userInput}</h1>
      </div>
    );
  }
}

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

Controlled vs Uncontrolled

There are two terms that will probably come up when we talk about React forms: controlled component and uncontrolled component. Like automatic binding, controlled vs uncontrolled components is a topic that we should be familiar with, but don’t need to understand deeply at this point.

An uncontrolled component is a component that maintains its own internal state. A controlled component is a component that does not maintain any internal state. Since a controlled component has no state, it must be controlled by someone else.

Think of a typical input element. It appears onscreen as a text box. If we know what text is currently in the box, then we can ask the input component, possibly with some code like this:

```
let input = document.querySelector('input[type="text"]');
```

```
let typedText = input.value; // input.value will be equal to whatever text is currently in the text box.
```

The important thing here is that the input component keeps track of its own text. We can ask it what its text is at any time.

The fact that input component keeps track of information makes it an uncontrolled component. It maintains its own internal state, by remembering data about itself.

A controlled component, on the other hand, has no memory. If we ask it for information about itself, then it will have to get that information throught props. Most React components are controlled.

In React when we give input component a value attribute, then input component becomes controlled. It stops using its internal storage. This is more React way of doing things.