**Advanced React Techniques**

In this unit, you will learn a variety of useful techniques that React programmers are expected to know.

You’ll learn how to make a *stateless functional component*, how to make a *propType*, how to write a form, and how to use styles.

You’ll also be introduced to your second programming pattern: dividing components into *presentational components* and *container components*.

**Inline Styles**

There are many different ways to use styles in React. This lesson is focused on one of them: *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 JavaScript into JSX. They say, “everything between us should be read as JavaScript, not JSX.”

The *inner* curly braces create a JavaScript object literal. They make this a valid JavaScript 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.

**Instructions**

**1.**

Click Run to see the <h1></h1> rendered to the browser. How can you add styles to this poor <h1></h1>?

Checkpoint 2 Passed

**2.**

Let’s give the <h1></h1> an *inline style*.

Give the <h1></h1> an attribute with a *name* of style. The attribute’s *value* should evaluate to this object:

{ background: 'lightblue', color: 'darkred' }

import React from 'react';

import ReactDOM from 'react-dom';

const styleMe = <h1 style={{ background: 'lightblue', color: 'darkred' }}>Please style me! I am so bland!</h1>;

ReactDOM.render(

  styleMe,

  document.getElementById('app')

);

**Make A Style Object Variable**

That’s all that you need to apply basic styles in React! Simple and straightforward.

One problem with this approach is that 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*, and then inject that variable into JSX.

Look in the code editor for an example. The style object is *defined* on lines 3-6, and then *injected* on line 11.

If you aren’t used to using modules, then this code may have made you twitch uncontrollably:

const style = {

color: 'darkcyan',

background: 'mintcream'

};

Defining a variable named style in the top-level scope would be an extremely bad idea in many JavaScript environments! In React, however, it’s totally fine.

Remember that every file is invisible to every other file, except for what you choose to expose via module.exports. You could have 100 different files, all with global variables named style, and there could be no conflicts.

**Instructions**

**1.**

Select **styleMe.js**. Make a new line after import ReactDOM from 'react-dom';.

On this new line, declare a new constant named styles. Set styles equal to this object:

{

background: 'lightblue',

color: 'darkred'

}

Checkpoint 2 Passed

**2.**

Change the *value* of your <h1></h1>‘s style attribute. Make style‘s value equal to your new styles variable.

Since you aren’t injecting an object *literal* anymore, you will no longer need to use double curly braces.

# Style Name Syntax

In regular JavaScript, 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 on style property names.

import React from 'react';

import ReactDOM from 'react-dom';

const styles = {

  background: 'lightblue',

  color:      'darkred',

  marginTop: '100px',

  fontSize: '50px'

};

const styleMe = <h1 style={styles}>Please style me! I am so bland!</h1>;

ReactDOM.render(

  styleMe,

  document.getElementById('app')

);

**Style Value Syntax**

In the last exercise, you learned how style *names* are slightly different in React than they are in regular JavaScript.

In this exercise, you will learn how style *values* are slightly different in React than they are in regular JavaScript.

In regular JS, style *values* are almost always strings. Even if a style value is numeric, you usually have to write it as a string so that you can specify a unit. For example, you have to write "450px" or "20%".

In React, if you write a style value as a *number*, then the unit "px" is assumed.

How convenient! If you want a font size of 30px, you can write:

{ fontSize: 30 }

If you want to use units other than “px,” you can use a string:

{ fontSize: "2em" }

Specifying “px” with a string will still work, although it’s redundant.

A few specific styles will *not* automatically fill in the “px” for you. These are styles where you aren’t likely to use “px” anyway, so you don’t really have to worry about it. [Here is a list of styles that don’t assume “px”.](https://facebook.github.io/react/tips/style-props-value-px.html)

**Instructions**

**1.**

In your styles object, change any property values that end in “px” from strings into numbers.

# Separate Container Components From Presentational Components: Explanation

In this lesson, you will learn your second programming pattern: separating presentational components from display components.

Click Run. In the browser, navigate to https://localhost:8000.

You are looking at an rendered <GuineaPigs /> component.

<GuineaPigs />‘s job is to render a photo carousel of guinea pigs. It does this perfectly well! And yet, it has a problem: it does too much stuff.

We can break <GuineaPigs /> into smaller components, but before we do: how do we know that GuineaPigs does too much stuff? How can you tell when a component has too many responsibilities?

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

# Separate Container Components From Presentational Components: Apply

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

Here’s the basic idea behind it: 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.

Instead of rendering HTML-like JSX, the component should render another component. It should be that component’s job to render HTML-like JSX.

Following this pattern separates your business logic from your presentational logic, which is a [Good Thing](http://www.dictionary.com/browse/good-thing).

**Instructions**

**1.**

**GuineaPigs.js** contains a lot of logic! You can’t even see the render function unless you scroll down.

It makes sense that GuineaPigs would have so much work to do. It has to select the correct guinea pig to render, wait the right amount of time before rendering, render an image, select the next correct guinea pig, and so on.

Let’s divide GuineaPigs in a presentational component and a container component. Near the top left of the code editor, click on the folder icon.

Create a new folder named containers. containers should be next to components.

Inside of containers, create a new file named GuineaPigsContainer.js. Make sure that GuineaPigs is plural, but Container is singular!

Once you have made containers/GuineaPigsContainer.js, click Run.

Checkpoint 2 Passed

**2.**

Good!

Open **components/GuineaPigs.js**.

You want to separate this component class into two: one component for rendering, and one for logic. How do you do that?

To start, just make a copy. After that, you can delete the appropriate parts from each copy.

Highlight the entire contents of **components/GuineaPigs.js**, and copy it to the clipboard (cmd-A/cmd-C for Mac, ctrl-A/ctrl-C for Windows).

Now, open **containers/GuineaPigsContainer.js**. Click inside the empty file, and paste. **containers/GuineaPigsContainer.js** and **components/GuineaPigs.js** should be identical.

Checkpoint 3 Passed

**3.**

Select **components/GuineaPigs.js**, and then close the file browser.

Look at the GuineaPigs component class, starting on line 11. This is going to be your presentational component class. That means that its only job will be to render HTML-like JSX.

On line 2, delete import ReactDOM from 'react-dom'.

At the bottom of the file, delete the ReactDOM.render() call.

Export GuineaPigs by adding the keyword export to the beginning of class GuineaPigs.

Checkpoint 4 Passed

**4.**

Good! But why did you just do that?

Because a presentational component will always get rendered by a container component.

GuineaPigs will get rendered by GuineaPigsContainer. Any component that gets rendered by a different component should use export.

Select **containers/GuineaPigsContainer.js**.

Make a new line after line 2. On your new line, import GuineaPigs.

This will be slightly different from what you’ve done before! As you saw when you opened the file navigator, **GuineaPigs.js** and **GuineaPigsContainer.js** are not neighbors. The filepath that you pass to import will have to navigate up one level, and then down into the components folder.

Checkpoint 5 Passed

**5.**

Select **containers/GuineaPigsContainer.js**.

Look at the GuineaPigs component class, starting on line 12. This is going to be your container component class.

On line 12, change the component class’s name from GuineaPigs to GuineaPigsContainer.

In the ReactDOM.render call near the bottom of the file, change <GuineaPigs /> to <GuineaPigsContainer />.

Checkpoint 6 Passed

**6.**

GuineaPigsContainer contains a lot of logic. It shouldn’t also have to render HTML-like JSX.

Delete any HTML-like JSX from GuineaPigsContainer‘s render function’s return statement. Instead, return an instance of GuineaPigs. The new render function should look like this:

render() {

const src = GUINEAPATHS[this.state.currentGP];

return <GuineaPigs />;

}

Checkpoint 7 Passed

**7.**

Once your container component has chosen a guinea pig, it must pass that guinea pig to the presentational component.

In GuineaPigsContainer‘s render function, pass the chosen guinea pig by giving <GuineaPigs /> a prop of src={src}.

Checkpoint 8 Passed

**8.**

Select **components/GuineaPigs.js**.

This component is supposed to be the opposite of GuineaPigsContainer. Its only job is to render HTML-like JSX.

Delete everything inside of the GuineaPigs component class, except for the render function. When you’re done, the object passed to React.component should only have one property: render!

Inside of the render function, delete this line of logic:

const src = GUINEAPATHS[this.state.currentGP];

… and replace it with this:

const src = this.props.src;

Lastly, delete the GUINEAPATHS array.

Checkpoint 9 Passed

**9.**

That should do it!

You divided GuineaPigs into two separate component classes: GuineaPigs and GuineaPigsContainer.

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!

You can find a lot of intelligent articles written about this pattern. [Here’s](https://medium.com/@learnreact/container-components-c0e67432e005#.gacsoomn1) a nice one to start with.

import React from 'react';

import ReactDOM from 'react-dom';

import { GuineaPigs } from '../components/GuineaPigs';

const GUINEAPATHS = [

  'https://s3.amazonaws.com/codecademy-content/courses/React/react\_photo-guineapig-1.jpg',

  'https://s3.amazonaws.com/codecademy-content/courses/React/react\_photo-guineapig-2.jpg',

  'https://s3.amazonaws.com/codecademy-content/courses/React/react\_photo-guineapig-3.jpg',

  'https://s3.amazonaws.com/codecademy-content/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() {

    let src = GUINEAPATHS[this.state.currentGP];

    return <GuineaPigs src={src} />;

  }

}

ReactDOM.render(

  <GuineaPigsContainer />,

  document.getElementById('app')

);

int

# Stateless Functional Components

In the code editor, take a look at GuineaPigs from the last lesson.

Notice that its instructions object only has one property: render().

When you separate a container component from a presentational component, the presentational component will always end up like this: one render() function, and no other properties.

If you have a component class with nothing but a render function, then you can rewrite that component class in a very different way. Instead of using React.Component, you can write it as a JavaScript function!

A component class written as a function is called a stateless functional component. Stateless functional components have some advantages over typical component classes. We’ll cover those advantages in this lesson.

Click on **Example.js** to see a stateless functional component in action.

**Instructions**

**1.**

Select **Friend.js**.

Rewrite the Friend component class as a stateless functional component.

Use **Example.js** as a guide. Make sure to delete the original Friend component class when you’re done.

import React from 'react';

import ReactDOM from 'react-dom';

export const Friend = () => {

  return <img src='https://s3.amazonaws.com/codecademy-content/courses/React/react\_photo-octopus.jpg' />;

}

ReactDOM.render(

  <Friend />,

  document.getElementById('app')

);

# Stateless Functional Components and Props

Stateless functional components usually have props passed to them.

To access these props, give your stateless functional component a parameter. This parameter will automatically be equal to the component’s props object.

It’s customary to name this parameter props. Read **Example.js** to see how it works.

Not only are stateless functional components more concise, but they will subtly influence how you think about components in a positive way. They emphasize the fact that components are basically functions! A component takes two optional inputs, props and state, and outputs HTML and/or other components.

You’ll be seeing a lot of stateless functional components in the next React course!

**Instructions**

**1.**

Open **GuineaPigs.js**.

After you divided GuineaPigs into GuineaPigs and GuineaPigsContainer, GuineaPigs was left with only a render function. That means that GuineaPigs can be rewritten as a stateless functional component!

Rewrite the GuineaPigs component class as a stateless functional component. Use **Example** as a guide.

Make sure to delete the original GuineaPigs class when you’re done.

import React from 'react';

export const GuineaPigs = (props) => {

  let src = props.src;

  return (

    <div>

      <h1>Cute Guinea Pigs</h1>

      <img src={src} />

    </div>

  );

}

# Apply PropTypes

In the code editor, take a look at MessageDisplayer‘s render function.

Notice the expression this.props.message. From this expression, you can deduce that MessageDisplayer expects to get passed a prop named message. Somewhere, at some time, this code is expected to execute:

<MessageDisplayer message="something" />

If a component class expects a prop, then you can give that component class a propType!

The first step to making a propType is to search for a property named propTypes on the instructions object. If there isn’t one, make one! You will have to declare it after the close of your component declaration, since it will be a static property.

See the example of a propTypes property on lines 11-13. Notice that the value of propTypes is an object, not a function!

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

MessageDisplayer only expects one prop: message. Therefore, its propTypes object only has one property.

**Instructions**

**1.**

Select **BestSeller.js**.

Give the BestSeller component class a propTypes property. For now, set propTypes equal to an empty object literal.

import React from 'react';

export class BestSeller extends React.Component {

  render() {

    return (

      <li>

        Title: <span>

          {this.props.title}

        </span><br />

        Author: <span>

          {this.props.author}

        </span><br />

        Weeks: <span>

          {this.props.weeksOnList}

        </span>

      </li>

    );

  }

}

BestSeller.propTypes = {};

# Add Properties to PropTypes

In the code editor, look at the property on MessageDisplayer‘s propTypes object:

message: React.PropTypes.string

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. In our case, MessageDisplayer expects a prop named message, so our property’s name is message.

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

React.PropTypes.expected-data-type-goes-here

Since message is presumably going to be a string, we chose React.PropTypes.string. You can see this on line 12. Notice the difference in capitalization between the propTypes object and React.PropTypes!

Each property on the propTypes object is called a propType.

Select the next file in code editor, **Runner.js**. Find Runner‘s propTypes object.

Runner has six propTypes! Look at each one. Note that bool and func are abbreviated, but all other datatypes are spelled normally.

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

Try to find all six props from the propTypes object in Runner‘s render function: this.props.message, this.props.style, etc.

**Instructions**

**1.**

Select **BestSeller.js**.

In BestSeller‘s propTypes object, write one propType for each prop that BestSeller is expecting: title, author, and weeksOnList.

Make title and author strings. Make weeksOnList a number. Make all three isRequired.

If you get stuck, look to **Runner.js** for guidance.

Checkpoint 2 Passed

**2.**

Good! You just gave BestSeller three propTypes.

In the code editor, open the last file, **BookList.js**.

At the bottom of the file, render <BookList /> using ReactDOM.render.

# PropTypes in Stateless Functional Components

Remember stateless functional components? You can see some familiar ones in **Example.js**.

How could you write propTypes for a stateless functional component?

// Usual way:

class Example extends React.component{

}

Example.propTypes = {

};

...

// Stateless functional component way:

const Example = (props) => {

// ummm ??????

It turns out the process is fairly similar. To write propTypes for a stateless functional component, you define a propTypes object as a property of the stateless functional component itself. Here’s what that looks like:

const Example = (props) => {

return <h1>{props.message}</h1>;

}

Example.propTypes = {

message: React.PropTypes.string.isRequired

};

**Instructions**

**1.**

Select **GuineaPigs.js**.

You can see your GuineaPigs stateless functional component from earlier. Let’s give it a propType.

After the GuineaPigs class declaration, define a propTypes property on GuineaPigs. Use the example code above as a guide.

GuineaPigs is only expecting one prop, so it should only get one propType.

Give GuineaPigs one propType, matching its expected prop. Make the propType isRequired.

If you aren’t sure what prop GuineaPigs is expecting, check the render function in **GuineaPigsContainer.js**.

GuineaPigs.propTypes = {

  src: React.PropTypes.string.isRequired

};

# React Forms

This unit’s final lesson is about forms.

Think about how forms work in a typical, non-React environment. A user types some data 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 JavaScript 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 JavaScript app with many moving, interdependent parts, this kind of conflict can easily lead to problems.

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

# Input onChange

A traditional form doesn’t update the server until a user hits “submit.” But you want to update the server any time a user enters or deletes any character.

**Instructions**

**1.**

Look at **Input.js**.

Can you find the bug? It’s somewhere in the render function. Scroll down once you have a guess.

…

…

…

…

…

…

…

There’s a self-closing tag without a forward slash! Find the missing forward slash and fill it in.

Checkpoint 2 Passed

**2.**

Look at Input‘s render function. View the result on the screen. Try typing into the <input /> in the browser.

Once Input has been set up correctly, then you will be able to change the <h1></h1>‘s inner text by typing into the <input /> in the browser.

Checkpoint 3 Passed

**3.**

You want to respond to any entered or deleted character in the <input /> field!

The best way to do that is by listening for a “change” event on the <input />.

Give <input /> an onChange attribute. Set onChange‘s value equal to {this.handleUserInput}. Don’t worry about the fact that there is no handleUserInput function yet - you’ll make one!

import React from 'react';

import ReactDOM from 'react-dom';

export class Input extends React.Component {

  render() {

    return (

      <div>

        <input type="text" onChange={this.handleUserInput} />

        <h1>I am an h1.</h1>

      </div>

    );

  }

}

ReactDOM.render(

  <Input />,

  document.getElementById('app')

);

# Write an Input Event Handler

In this exercise, you will define a function that gets called whenever a user enters or deletes any character.

This function will be an event handler. It will listen for change events. You can see an example of an event handler listening for change events in **Example.js**.

**Instructions**

**1.**

Select **Input.js**.

Before Input‘s render function, write a new method named handleUserInput.

Give this method one parameter, named e.

Inside of this function’s body, call this.setState. Set the state’s userInput property equal to e.target.value.

e.target.value will equal the text in the <input /> field. You are setting this.state.userInput equal to whatever text is currently in <input />.

  handleUserInput(e) {

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

  }

# Set the Input's Initial State

Good! Any time that someone types or deletes in <input />, the .handleUserInput() method will update this.state.userInput with the <input />‘s text.

Since you’re using this.setState, that means that Input needs an initial state! What should this.state‘s initial value be?

Well, this.state.userInput will be displayed in the <input />. What should the initial text in the <input /> be, when a user first visits the page?

The initial text should be blank! Otherwise it would look like someone had already typed something.

**Instructions**

**1.**

Give Input a constructor function which takes a parameter of props and calls super(props) on its first line.

Then, still in the constructor, set state equal to this object:

{ userInput: '' }

Feel free to use the example code as a guide.

Checkpoint 2 Passed

**2.**

Next, at the end of the constructor, bind .handleUserInput() to the current value of this.

 constructor(props) {

    super(props);

    this.state = { userInput: '' };

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

  }

# Controlled vs Uncontrolled

There are two terms that will probably come up when you talk about React forms: controlled component and uncontrolled component. Like automatic binding, controlled vs uncontrolled components is a topic that you 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 type='text' /> element. It appears onscreen as a text box. If you need to know what text is currently in the box, then you can ask the <input />, 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 /> keeps track of its own text. You can ask it what its text is at any time, and it will be able to tell you.

The fact that <input /> 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 you ask it for information about itself, then it will have to get that information through props. Most React components are controlled.

In React, when you give an <input /> a value attribute, then something strange happens: the <input /> BECOMES controlled. It stops using its internal storage. This is a more ‘React’ way of doing things.

You can find more information about controlled and uncontrolled components in the [React Forms documentation](https://reactjs.org/docs/forms.html).

# React Forms Recap

Great work! You just wrote your first React form.

Notice that you didn’t use a submit button. You didn’t even use a <form> element! Your “form” was actually just an <input />.

That won’t always be the case. You will still sometimes want a <form> element and a submit button, especially if you need to differentiate between a finished form and an in-progress form. But in some cases, it’s fine to have a “form” that is really just an input field.

This is because, unlike in the traditional form paradigm, in React you re-send your form on every single character change. That removes the need to ever “submit” anything.

That marks the end of this unit! You’ve learned a wide variety of important techniques: inline styles, separating container and presentational components, stateless functional components, proptypes, and forms. You’ll review all of it in the next course! There is only one major tool still missing from your toolbelt: lifestyle methods. We’ll cover those in this course’s final unit.