# 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.

Click Next to begin!

# 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.

# 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.

# 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 valuesare 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)

# Share Styles Across Multiple Components

What if you want to reuse styles for several different components?

One way to make styles reusable is to keep them in a separate JavaScript file. This file should export the styles that you want to reuse, via export. You can then import your styles into any component that wants them.

In the code editor, move back and forth between **facebookStyles.js** and **FacebookColorThief.js** to see a styles file in action.

**CONTAINER COMPONENTS FROM PRESENTATIONAL COMPONENTS**

# 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.

**CONTAINER COMPONENTS FROM PRESENTATIONAL COMPONENTS**

# 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).

**STATELESS FUNCTIONAL COMPONENTS**

# Stateless Functional Components

In the code editor, take a look at GuineaPigsfrom 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 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.

# 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!

**PROPTYPES**

# propTypes

In this lesson, you will learn to use an important React feature called propTypes.

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

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

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

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

Click Next to learn how to use propTypes!

# 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 propnamed 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 this it will be a static property.

See the example of a propTypes property on lines 11-13. Notice that the value of propTypesis 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.

# 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 propTypessupposed 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'tsent.

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

# 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 if 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 };

**REACT FORMS**

# 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 thirdpart 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.

# 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**.

# 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.

# Update an Input's Value

When a user types or deletes in the <input />, then that will trigger a change event, which will call handleUserInput. That's good!

handleUserInput will set this.state.userInputequal to whatever text is currently in the input field. That's also good!

There's only one problem: you can set this.state.userInput to whatever you want, but <input /> won't care. You need to somehow make the <input />'s text responsive to this.state.userInput.

Easy enough! You can control an <input />'s text by setting its value attribute.

# 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 controlledby 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 valueattribute, 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.