React Components Basics

(https://github.com/learn-co-curriculum/react-hooks-components-basics) (https://github.com/learn-co-curriculum/react-hooks-components-basics/issues/new)

Learning Goals

- Understand what a React component is and what it can be used for
- Write React components and identify the DOM elements they create

Introduction

In this lesson, we'll introduce the heart of React: components. This will include explaining why they're important and examining a few examples. If the idea and application of components doesn't click immediately, *do not worry!* The different moving parts required to understand how to use them will fall into place as we move forward.

Let's examine a high level overview of what a React component is before we implement one. The official **React documentation on components** (https://reactjs.org/docs/components-and-props.html) says it best:

> Components let you split the UI into independent, reusable pieces, and think about each piece in isolation.

Components modularize both *functionality* and *presentation* in our code. In order to understand how powerful this is, consider just how intricate web applications can become. The difficulty in logically arranging, architecting, and programming these web applications increases with their size. Components are like little packages: they help us keep everything organized and predictable while abstracting the **'boilerplate'** (https://en.wikipedia.org/wiki/Boilerplate_code) code.

Components can do many things, but their end goal is always the same: they all must contain a snippet of code that describes what they should render to the DOM.

React Application Idea

Enough of a description — let's see some examples! While the possibilities of what we can do with components are endless, the first thing we need to understand about them is the ways in which they act as **code templates**. Let's start simply and build up from there using the following as an example:

Let's imagine we want a blog article describing the fact (note: not opinion) of why Bjarne Stroustrup has the **perfect lecture oration (https://www.youtube.com/watch?v=JBjjnqG0BP8)**

(https://www.youtube.com/watch?v=JBjjnqG0BP8)
. We also want our blog article to display comments made by readers.

Fork and clone the repo for this lesson if you'd like to follow along.

After forking the repo, run the `npm install` command to install all the dependencies and the `npm start` command to run the app.

Step 1: Write the Components

First, let's make a component for our article:

```
// src/Article.js
function Article() {
  return (
    <div>Dear Reader: Bjarne Stroustrup has the perfect lecture oration.</div>
  );
}
```

Take a moment to read that code line by line:

- we declare a function, `Article`

- the function has a return value of **JSX**, which is our way of telling React "Hey, when you want to put this component on the DOM, here is what it should become!"

When React creates this element and adds it to the DOM, the resulting element will look just as you would expect:

```
<div>Dear Reader: Bjarne Stroustrup has the perfect lecture oration.</div>
```

Let's see what it would look like if we were to only render this one component in the DOM:

That takes care of our `Article` part of our application. Now let's make a component to display a single user's comment:

```
function Comment() {
  return <div>Naturally, I agree with this article.</div>;
}
```

Take the time to read that component line by line. Here is the element that this would create when added to the DOM:

```
<div>Naturally, I agree with this article.</div>
```

In both of our examples, React is taking JavaScript code, interpreting that special JSX syntax within the `return()` statement, and spitting out plain old HTML that browsers will know how to represent to the user.

Once we have our components in hand, it's time to actually use them.

Step 2: Use the Components

Now that we have these components written, all we need to do is make sure some *other* component is making use of them in its **return statement**. Every React application has some top level component(s). Very often, this top level component is simply called `App`. For our example, here's what it might look like:

```
function App() {
  return (
    <div>
      <Article />
      <Comment />
    </div>
  );
}
```

Here we can see JSX coming into play a bit more. The code inside the `return()` still looks a lot like regular HTML, but in addition to rendering a regular old HTML `<div>` element, we're also rendering our two components. We've created code that is not only well structured and modular, but also a straightforward description of what we want the `App` component to do: render the article first, followed by the comment. Here is what the resulting elements will look like:

```
<div>
  <div>Dear Reader: Bjarne Stroustrup has the perfect lecture oration.</div>
  <div>Naturally, I agree with this article.</div>
</div>
```

This unpacks logically. The `App` component (being our top level component) wraps around both `Article` and `Comment`, and we already know what they look like when they are turned into HTML.

As you may expect, we refer to the `App` component as both the `Comment` and `Article` component's *parent* component. Inversely, we refer to `Comment` and `Article` as *children* components of `App`.

Naming Components

You'll notice that both of the custom components we created, `Article` and `Comment`, are declared as functions whose names start with a **capital letter**:

```
function Article() {
  return (
    <div>Dear Reader: Bjarne Stroustrup has the perfect lecture oration.</div>
  );
}
function Comment() {
  return <div>Naturally, I agree with this article.</div>;
}
```

This naming convention is important for a couple very good reasons:

- It helps React developers to easily differentiate between regular JavaScript functions and React components
- More importantly, it's a **rule that we must follow (https://reactjs.org/docs/jsx-in-depth.html#user-defined-components-must-be-capitalized)** in order for React to render our components correctly.

For instance, if we defined our `Article` component using a lower-case letter, like this:

```
function article() {
  return (
    <div>Dear Reader: Bjarne Stroustrup has the perfect lecture oration.</div>
```

```
  );
}
```

When it came time to use that component within another component, React would treat it as a regular `<article>` HTML element rather than one of our custom components:

```
function App() {
  return (
    <div>
      <article />
    </div>
  );
}

// returns these DOM elements:
// <div>
//  <article />
// </div>
```

By naming it with a capital letter instead, we get the desired DOM elements returned:

```
function App() {
  return (
    <div>
      <Article />
    </div>
  );
}

// returns these DOM elements:
// <div>
```

```
//  <div>Dear Reader: Bjarne Stroustrup has the perfect lecture oration.</div>
// </div>
```

A Note on Classes

As you're exploring the React documentation, and finding other resources on React on the internet, you'll probably notice there are multiple syntaxes you can use for creating components: **function** components and **class** components.

A **function** component looks like this:

```
function Comment() {
  return <div>Naturally, I agree with this article.</div>;
}
```

Or using the arrow function syntax:

```
const Comment = () => <div>Naturally, I agree with this article.</div>;
```

A **class** component looks like this:

```
class Comment extends React.Component {
  render() {
    return <div>Naturally, I agree with this article.</div>;
  }
}
```

For many years, the only way to work with certain key features of React — *state* and *lifecycle* — was to use **class** components. Since the introduction of **hooks (https://reactjs.org/docs/hooks-intro.html)** in React 16.8, this is no longer true, and function components can be used for (almost) everything that class components can.

React's recommendation is that components should be written as function components moving forward, but class components will continue to be supported as well. React also recently released a **beta version of their new docs (https://beta.reactjs.org)** that focuses on function

components and hooks.

It's important to learn more about class components later on, so that when you encounter them in legacy code, you'll still be able to work with them. However, for the time being, we'll just be focusing on function components.

Conclusion

We just introduced simplified, bare bones, React components. They are used to house modularized front end code. In our example — as is often the case — they contain information on how a portion of our application should be turned into HTML.

The minimum requirement for a React component is that it must be a function that starts with a capital letter and returns JSX.

Going forward, we will continue with this example and show how components can be re-used and how they can be written as templates in which content is populated dynamically.

Resources

- **React Top-Level API (https://reactjs.org/docs/react-api.html)**
- **Introducing JSX (https://reactjs.org/docs/introducing-jsx.html)**
- **React Docs (beta) (https://beta.reactjs.org)**