COMPONENTS RENDER OTHER COMPONENTS

Components Interact

A React application can contain dozens, or even hundreds, of components.

Each component might be small and relatively unremarkable on its own. When combined, however, they can form enormous, fantastically complex ecosystems of information.

In other words, React apps are made out of components, but what makes React special isn't components themselves. What makes React special is the ways in which components *interact*.

This unit is an introduction to *components interacting*.

A Component in a Render Function

Here is a `.render()` method that returns an HTML-like JSX element:

```
class Example extends React.Component {
  render() {
    return <h1>Hello world</h1>;
  }
}
```

You've seen render methods return `<div></div>`s, `<p></p>`s, and `<h1></h1>`s, just like in the above example.

Render methods can also return another kind of JSX: *component instances*.

```
class OMG extends React.Component {
  render() {
    return <h1>Whooaa!</h1>;
  }
}

class Crazy extends React.Component {
  render() {
    return <OMG />;
```

```
  }
}
```

In the above example, `Crazy`'s render method `returns` an *instance* of the `OMG` component class. You could say that `Crazy` renders an `<OMG />`.

Apply a Component in a Render Function

This is new territory! You've never seen a component rendered by another component before.

You *have* seen a component rendered before, though, but not by another component. Instead, you've seen a component rendered by `ReactDOM.render()`.

When a component renders another component, what happens is very similar to what happens when `ReactDOM.render()` renders a component.

Instructions

1.

You can see two files in the code editor: **ProfilePage.js** and **NavBar.js**.

In this lesson, you are going to make a `<ProfilePage />` *render* a `<NavBar />`.

How do you do that? To start, simply make `ProfilePage`'s `render()` method return a `<NavBar />` instance.

In **ProfilePage.js**, place a `<NavBar />` on line 9.

Checkpoint 2 Passed

Hint

Between the `<div>` and the `<h1>`, you'll put the following to render a navigation bar:

```
<NavBar />
```

ProfilePage.js

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

class ProfilePage extends React.Component {
  render() {
    return (
      <div>
      <NavBar />
        <h1>All About Me!</h1>
        <p>I like movies and blah blah blah blah blah</p>
        <img src="https://content.codecademy.com/courses/Rea
ct/react_photo-monkeyselfie.jpg" />
      </div>
    );
  }
}

ReactDOM.render(<ProfilePage />, document.getElementById('ap
p'));
```

NavBar.js

```
import React from 'react';

export class NavBar extends React.Component {
  render() {
    const pages = ['home', 'blog', 'pics', 'bio', 'art', 'sh
op', 'about', 'contact'];
    const navLinks = pages.map(page => {
      return (
        <a href={'/' + page}>
          {page}
        </a>
      )
    });

    return <nav>{navLinks}</nav>;
  }
}
```

Require A File

When you use React.js, every JavaScript file in your application is invisible to every other JavaScript file by default. **ProfilePage.js** and **NavBar.js** can't see each other.

This is a problem!

On line 9, you just added an instance of the `NavBar` component class. But since you're in **ProfilePage.js**, JavaScript has no idea what `NavBar` means.

If you want to use a variable that's declared in a different file, such as `NavBar`, then you have to *import* the variable that you want. To import a variable, you can use an `import` statement:

```
import { NavBar } from './NavBar.js';
```

We've used `import` before, but not like this! Notice the differences between the above line of code and this familiar line:

```
import React from 'react';
```

The first important difference is the curly braces around `NavBar`. We'll get to those soon!

The second important difference involves the contents of the string at the end of the statement: `'react'` vs `'./NavBar.js'`.

If you use an `import` statement, and the string at the end begins with either a dot or a slash, then `import` will treat that string as a *filepath*. `import` will follow that filepath, and import the file that it finds.

If your filepath doesn't have a file extension, then ".js" is assumed. So the above example could be shortened:

```
import { NavBar } from './NavBar';
```

One final, important note:
None of this behavior is specific to React! Module systems of independent, importable files are a very popular

way to organize code. React's specific module system comes from ES6. More on all of that later.

Instructions

1.

The `<NavBar />` on line 9 isn't going to work until you *import* **NavBar.js**.

In **ProfilePage.js**, on line 3, import `NavBar` from **NavBar.js**. **ProfilePage.js** and **NavBar.js** are located in the same parent directory.

Checkpoint 2 Passed

Hint

You'll place the following on line 3:

```
import { NavBar } from './NavBar.js';
```

This will import the `<NavBar />` component.

ProfilePage.js

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

class ProfilePage extends React.Component {
  render() {
    return (
      <div>
      <NavBar />
        <h1>All About Me!</h1>
        <p>I like movies and blah blah blah blah blah</p>
        <img src="https://content.codecademy.com/courses/React/react_photo-monkeyselfie.jpg" />
      </div>
    );
  }
}

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

NavBar.js

```
import React from 'react';

export class NavBar extends React.Component {
  render() {
    const pages = ['home', 'blog', 'pics', 'bio', 'art', 'sh
op', 'about', 'contact'];
    const navLinks = pages.map(page => {
      return (
        <a href={'/' + page}>
          {page}
        </a>
      )
    });

    return <nav>{navLinks}</nav>;
  }
}
```

---

export

Alright! You've learned how to use `import` to grab a variable from a file *other than* the file that is currently executing.

When you import a variable from a file that is not the current file, then an `import` statement isn't quite enough. You also need an `export` statement, written in the *other* file, exporting the variable that you hope to grab.

`export` comes from [ES6's module system,](#) just like `import` does. `export` and `import` are meant to be used together, and you rarely see one without the other.

There are a few different ways to use `export`. In this course, we will be using a style called "named exports." Here's how named exports works:

In one file, place the keyword `export` immediately before something that you want to export. That something can be any top-level `var`, `let`, `const`, `function`, or `class`:

```
// Manifestos.js:

export const faveManifestos = {
  futurist:
'http://www.artype.de/Sammlung/pdf/russolo_noise.pdf',
  agile: 'https://agilemanifesto.org/iso/en/manifesto.html',
  cyborg:   'http://faculty.georgetown.edu/irvinem/theory/Ha
raway-CyborgManifesto-1.pdf'
};
```

You can export multiple things from the same file:

```
// Manifestos.js:

export const faveManifestos = {
  futurist:
'http://www.artype.de/Sammlung/pdf/russolo_noise.pdf',
  agile:  'https://agilemanifesto.org/iso/en/manifesto.html'
,
  cyborg:   'http://faculty.georgetown.edu/irvinem/theory/Ha
raway-CyborgManifesto-1.pdf'
};

export const alsoRan = 'TimeCube';
```

In a different file, `import` the name of the `var`, `let`, `const`, `function`, or `class` from the first file:

```
// App.js:

// Import faveManifestos and alsoRan from ./Manifestos.js:
import { faveManifestos, alsoRan } from './Manifestos';

// Use faveManifestos:
console.log(`A Cyborg
Manifesto:  ${faveManifestos.cyborg}`);
```

This style of importing and exporting in JavaScript is known as “named exports.” When you use named exports, you always need to wrap your imported names in curly braces, such as:

```
import { faveManifestos, alsoRan } from './Manifestos';`
```

JavaScript’s ES6 module system goes beyond named exports and has several advanced syntax features.

Instructions

1.

Select **NavBar.js**.

On line 3, add the word `export` before the word `class`. This will export the class `NavBar`.

Now, when **ProfilePage.js** uses `import` to grab the variable `NavBar` from **NavBar.js**, it will get back exactly what it wants: the `NavBar` component class.

Checkpoint 2 Passed

Hint

Find `class NavBar` and add the word `export` before it, so it becomes `export class NavBar`.

NavBar.js

```
import React from 'react';

export class NavBar extends React.Component {
  render() {
    const pages = ['home', 'blog', 'pics', 'bio', 'art', 'sh
op', 'about', 'contact'];
    const navLinks = pages.map(page => {
      return (
        <a href={'/' + page}>
          {page}
        </a>
      )
    });

    return <nav>{navLinks}</nav>;
  }
}
```

ProfilePage.js

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

class ProfilePage extends React.Component {
```

```
  render() {
    return (
      <div>
      <NavBar />
        <h1>All About Me!</h1>
        <p>I like movies and blah blah blah blah blah</p>
        <img src="https://content.codecademy.com/courses/React/react_photo-monkeyselfie.jpg" />
      </div>
    );
  }
}

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

Component Rendering In Action

Now you're ready for `<ProfilePage />` to render `<NavBar />`!

All that's left to do is render `<ProfilePage />`.

Instructions

1.

At the bottom of **ProfilePage.js**, use `ReactDOM.render()` to render an instance of `ProfilePage`.

For `ReactDOM.render()`'s second argument, pass in `document.getElementById('app')`.

Once the result has rendered in the browser, look at the render methods of both `ProfilePage` and `NavBar`. Try to figure out exactly which parts of the browser's display come from which component class.

Checkpoint 2 Passed

Hint

Call `ReactDOM.render()` with two arguments: `<ProfilePage />` and `document.getElementById('app')`.

2.

Congratulations! It may not seem like a big deal yet, but you've just discovered the key to React's power.

By nesting components inside of other components, you can build infinitely complex architectures, even if each component is relatively simple. The relationship that you just built is the fundamental relationship of React.js.

ProfilePage.js

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

class ProfilePage extends React.Component {
  render() {
    return (
      <div>
      <NavBar />
        <h1>All About Me!</h1>
        <p>I like movies and blah blah blah blah blah</p>
        <img src="https://content.codecademy.com/courses/React/react_photo-monkeyselfie.jpg" />
      </div>
    );
  }
}

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

NavBar.js

```
import React from 'react';

export class NavBar extends React.Component {
```

```
  render() {
    const pages = ['home', 'blog', 'pics', 'bio', 'art', 'sh
op', 'about', 'contact'];
    const navLinks = pages.map(page => {
      return (
        <a href={'/' + page}>
          {page}
        </a>
      )
    });

    return <nav>{navLinks}</nav>;
  }
}
```