**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.

# 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 importstatement:

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](http://eloquentjavascript.net/10_modules.html) of independent, importable files are a very popular way to organize code.[React's specific module system comes from ES6](https://hacks.mozilla.org/2015/08/es6-in-depth-modules/). More on all of that later.

# 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 exportstatement, written in the other file, exporting the variable that you hope to grab.

export comes from [ES6's module system,](http://exploringjs.com/es6/ch_modules.html)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 exportimmediately 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', SCUM: 'http://www.ccs.neu.edu/home/shivers/rants/scum.html', cyborg: 'http://faculty.georgetown.edu/irvinem/theory/Haraway-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', SCUM: 'http://www.ccs.neu.edu/home/shivers/rants/scum.html', cyborg: 'http://faculty.georgetown.edu/irvinem/theory/Haraway-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](http://exploringjs.com/es6/ch_modules.html) goes beyond named exports and has several advanced syntax features.

# Component Rendering In Action

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

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