Tip 49: Build Applications with Component Architecture

In this tip, you'll learn how to gather related HTML, JavaScript, and CSS together using component architecture.

We'll cover the following

Organizing code

Organizing files can be a challenge. Front-end code—HTML, CSS, JavaScript—can be particularly challenging because the code is made of different languages and file types.

Do you arrange code by file type? What about when CSS is tied to a single HTML file? Do you keep them in different directories but with similar file names?

For a long time, developers would keep files separated by type. The root directory would contain a *css* directory, a *js* directory, an *img* directory, and so on.

Organizing files like this showed good intentions. Everyone wanted to keep different areas of concern separate. The HTML markup (what a site contains) is different from the CSS (how a site looks), which is different from the JavaScript (how a site responds). It seemed like they should be in separate directories.

The problem was that the pieces aren't really separate concerns. Except for a few global styles, CSS is built to work with specific markup. What happens when that markup is removed? If you have disciplined developers, they'd remove the relevant CSS. But most of the time, it stayed. It was never used—it just took up

relevant CSS. But most of the time, it stayed. It was never used—it just took up space.

Component architecture

As developer tools improved, a new pattern emerged. The new pattern is *component architecture*. A **component** is the combination of all *relevant* code into a *single* directory. You then build a web page or application by adding pieces one at a time—a button is in a sidebar, which is in a page—until you have your working application.

Component architecture isn't without problems. The biggest problem with component architecture is that it depends on *build tools* and, to a lesser extent, *frameworks*. In this tip, you'll be working with React code. You're going to use scaffolding developed by create-react-app. This means you don't have to worry about setting up the build system. You'll explore that a little in the next tip.

It's important to understand, however, that component architecture is not React specific. You can apply the idea in a variety of frameworks. Cody Lindley wrote a great article on the subject. Still, a framework saves some of the trouble of laying a foundation.

Example

To see component architecture at work, build a basic component: a *copyright* statement. A **copyright** statement contains the *current year*, a *declaration* of copyright, and some *styling*. With component architecture, you combine everything into a simple package. Here's an example:

```
import React from 'react';
import './Copyright.css';

export default function CopyrightStatement() {
    const year = new Date().getFullYear();
    return (
        <div className="copyright">
            Copyright {year}
        </div>
    );
}
```

To start off, notice that the markup is in a `return` statement, and the CSS class is called `className`. Don't worry about that. The specialized markup is called **JSX** and it's part of the React framework. You can pretend that the HTML is a separate thing. It effectively is separate. For purposes of this tip, it's just markup that

happens to live inside a JavaScript function.

Next, notice the path to the code at the top of the sample. Most of the time, you can ignore it, but in this case, it's relevant. `simplifying-js-component` is the *root* of the project. The code lives inside the `src/components` directory. There's also a public directory that will eventually contain the compiled code. A browser can't handle components, so everything will eventually combine to simpler components.

Components directory

The components directory contains every component you work with. Each component will then have its own separate directory. In this case, there's a directory called `Copyright` that contains `Copyright.css`, `Copyright.js`, and `Copyright.spec.js`. The capitalized names are also a React convention.

The `Copyright` directory contains everything the copyright component will need. If you wanted to share the component, you could put it in a separate repo or just copy and paste it in another project. If you decide you don't want the copyright anymore, you can delete the whole directory. You wouldn't need to worry that dead CSS lives somewhere else. Everything is together.

Speaking of CSS, notice how this file imports the CSS directly. Because you aren't importing an object from the CSS, you merely include the whole file. The build tools will know what to do with it. The CSS file for this example is very short. All it contains is the font-size, margin, and float.

```
.copyright {
    font - size: 10px;
    margin: 1em 1em 1em 0;
    float: left;
}
```

In the JavaScript file, you get the current year and add it to the markup. Notice how simple this is. Everything you need to know about that copyright statement is in a single place. You don't need to guess if the year is calculated or hard coded. It's right there with the markup. You don't need to search for the CSS if you need to change a margin. It's in the same directory.

Making reusable components

How about a slightly more complicated component? Think about a button that has an icon. The button will need styling and markup, but it will also need the image

asset and a click action.

This time, you also want to make the component reusable. That means you should hard code as few options as you can. Don't explicitly say what happens on click. Instead, *inject* the click action into a component. Passing in actions or assets to a component is another form of *dependency injection* that you explored in Tip 32, Write Functions for Testability. It keeps things flexible and reusable.

```
import React from 'react';
import './IdeaButton.css';
import idea from './idea.svg';

export default function IdeaButton({ handleClick, message }) {
    return (
        <button
            className="idea-button"
            onClick={handleClick}
        >
            <img
                className="idea-button__icon"
                src={idea}
                alt="idea icon"
            />
            {message}
        </button>
    );
}
```

In React, you can access the injected dependencies in the arguments of a function. And you can pull them apart using *destructuring*. The message will change depending on what's injected. The curly braces are a templating language, and they surround variable information. In other words, the button will contain the value of the `message` variable.

Notice also that you're importing an *image*. Unlike when you import the CSS without ever using it, in this case, you're importing the image to a *variable*. The variable contains the *path* of the image, so set the `src` to the variable using the curly braces.

Building a page

Now that you have the pieces, you can start building a page. In this case, the page is just another component! This page will contain the idea button and the copyright notice in a footer. You're still in React territory, so you inject the message as a special HTML attribute. Other frameworks use different conventions to inject data. But they all allow you to pass in some information—*this is what makes*

components so powerful.

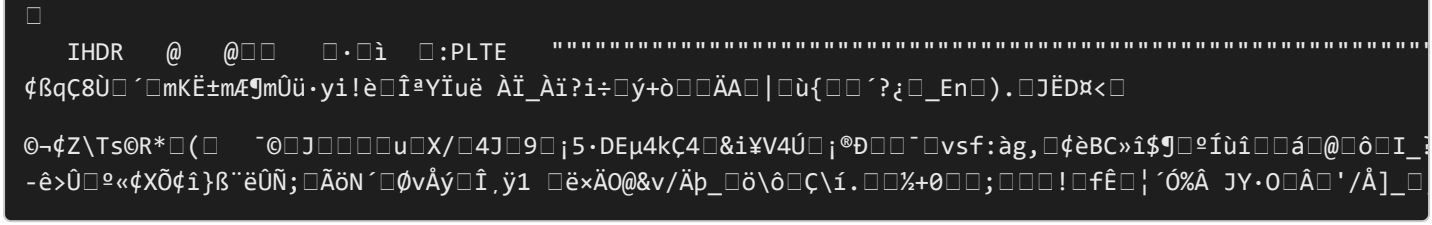
```
import React from 'react';
import './App.css';
import IdeaButton from './components/IdeaButton/IdeaButton';
import Copyright from './components/Copyright/Copyright';

function logIdea() {
    console.log('Someone had an idea!');
}

export default function App() {
    return (
        <div className="main">
            <div className="app">
                <IdeaButton
                    message="I have an idea!"
                    handleClick={logIdea}
                />
            </div>
            <footer>
                <Copyright />
                <IdeaButton
                    message="Footer idea!"
                    handleClick={logIdea}
                />
            </footer>
        </div>
    );
}
```

Because `App.js` is the main component, it lives at the *root* of the source code. Otherwise, it's the same. It imports code, it contains all the pieces, and it combines them together. In this case, you're reusing the *button* component twice. Each one will have a different message. As you can see in the following figure, the result isn't stunning, but it does show each piece.


```
  □ □ □□  □   ã□  F   □□  □   □  )□     □   9□  5□  @@  □   °□  n□  □PNG
□
   IHDR   □   □□□   (-□S   äPLTE"""""""""""""""""""2PX=r□)7;*:>H□¤-BGE□□8do5Xb6[eK□®K□¯1MU9gs3S
□
   IHDR   □   □□□   ×©ÍÊ  □ePLTE"""""""""""""""""""""""""2RZN¢¹J□«3R[J□¬)59YÁÞ0KS4W`Q«ÄL□²%+-0JR
?^q÷ñíÛ□ï.},□ìsæÝ_TttÔ¾ □1#□□/(ì□-[□□□è`□è`Ì□ÚïÅðZ□d5□□□□?ÎebZ¿Þ□i.Ûæ□□□ìqÎ□+1°□}Â□5ù  ïçd
□
   IHDR       □□   D¤□Æ  □APLTE   """""""""""""""""""""""""2RZVºÖ_ÔôU·Ñ=r□$()'25]ÎíC□□0LS<o}X
```



You can download the source code and try it out. The code contains a `README.md`, which will get you up and running with only two commands. After that, try changing some CSS. Try adding a new image. You'll see how simple it is to work with components when everything is in one logical place.

> You can try doing this in the coding widget for the application given above.

If you inspected the page, you'd notice something interesting. You combined the separate CSS files into a single file and moved that file to a separate css directory. The same thing happened to images. In this case, the build tools still separate out the pieces into different directories. That's great! There's nothing wrong with having pieces separate at the user level. The goal is to make development easier.

Intuitively, component architecture probably makes sense. Keep like things together. The only downside is that wiring everything together isn't easy. The only reason component architecture works is because you can use great tools that intelligently combine code.

In the next tip, you'll learn how to compile front-end code with build tools.