**Working with Base Lightning Components**

Base Lightning components are the building blocks that make up the modern user interfaces in Lightning Experience, Salesforce app, and Experience Builder sites.

Base Lightning components incorporate Lightning Design System markup and classes, providing improved performance and accessibility with a minimum footprint.

**Component Build Structure**

A component simply needs a folder and its files with the same name.

They're automatically linked by name and location.

All Lightning web components have a namespace that's separated from the folder name by a hyphen. For example, the markup for the Lightning web component with the folder name app in the default namespace c is <c-app>.

However, the Salesforce platform doesn't allow hyphens in the component folder or file names. What if a component's name has more than one word, like “mycomponent”? You can't name the folder and files my-component, but we do have a handy solution.

Use camel case to name your component myComponent. Camel case component folder names map to kebab-case in markup. In markup, to reference a component with the folder name myComponent, use <c-my-component>.

**Working with JavaScript**

Here's where you make stuff happen. As you've seen so far, JavaScript methods define what to do with input, data, events, changes to state, and more to make your component work.

The JavaScript file for a Lightning web component must include at least this code, where MyComponent is the name you assign your component class.

import { LightningElement } from 'lwc';

export default class MyComponent extends LightningElement {

}

The export statement defines a class that extends the LightningElement class. As a best practice, the name of the class usually matches the file name of the JavaScript class, but it's not a requirement.

**The LWC Module**

Lightning Web Components uses modules (built-in modules were introduced in ECMAScript 6) to bundle core functionality and make it accessible to the JavaScript in your component file. The core module for Lightning web components is lwc.

Begin the module with the import statement and specify the functionality of the module that your component uses.

The import statement indicates the JavaScript uses the LightningElement functionality from the lwc module.

// import module elements

import { LightningElement} from 'lwc';

// declare class to expose the component

export default class App extends LightningElement {

ready = false;

// use lifecycle hook

connectedCallback() {

setTimeout(() => {

this.ready = true;

}, 3000);

}

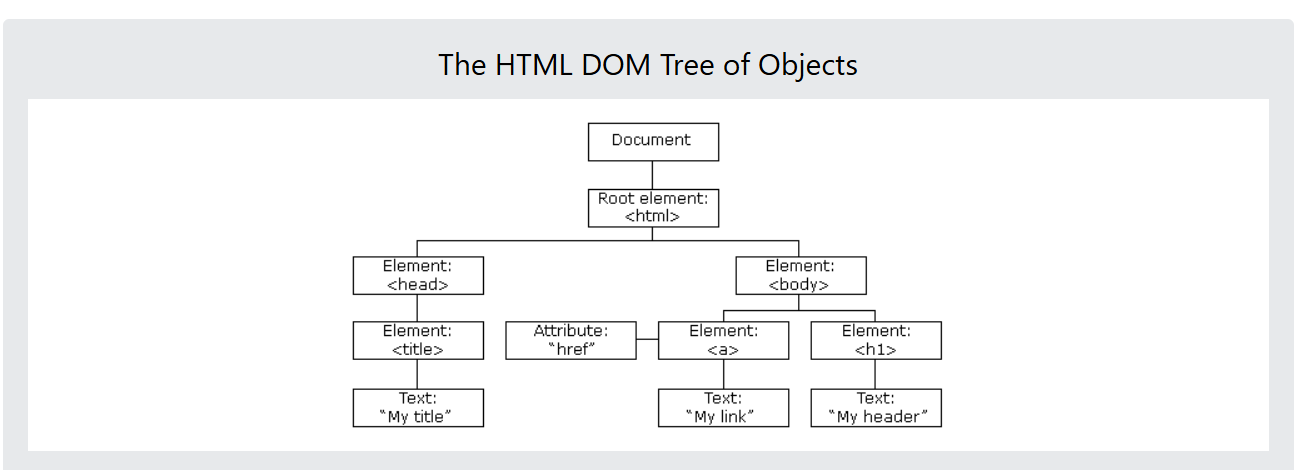
}

* **LightningElement** is the base class for Lightning web components, which allows us to use connectedCallback().
* **The connectedCallback()** method is one of our lifecycle hooks. You'll learn more about lifecycle hooks in the next section. For now, know that the method is triggered when a component is inserted in the document object model (DOM). In this case, it starts the timer.

**Lifecycle Hooks**

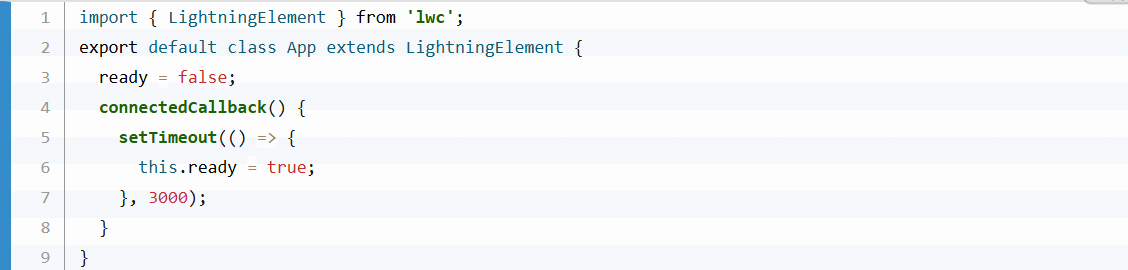
Lightning Web Components provides methods that allow you to “hook” your code up to critical events in a component's lifecycle. These events include when a component is:

* Created
* Added to the DOM
* Rendered in the browser
* Encountering errors
* Removed from the DOM



Respond to any of these lifecycle events using callback methods. For example, the connectedCallback() is invoked when a component is inserted into the DOM. The disconnectedCallback() is invoked when a component is removed from the DOM.

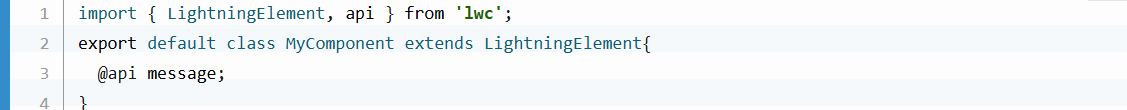
In the JavaScript file we used to test our conditional rendering, we used the connectedCallback() method to automatically execute code when the component is inserted into the DOM. The code waits 3 seconds, then sets ready to true.



**Decorators**

Decorators are often used in JavaScript to modify the behavior of a property or function.

To use a decorator, import it from the lwc module and place it before the property or function.



You can import multiple decorators, but a single property or function can have only one decorator. For example, a property can't have @api and @wire decorators.

**The Component Configuration File**

The file we haven't covered yet is the component configuration file with the extension .js-meta.xml.

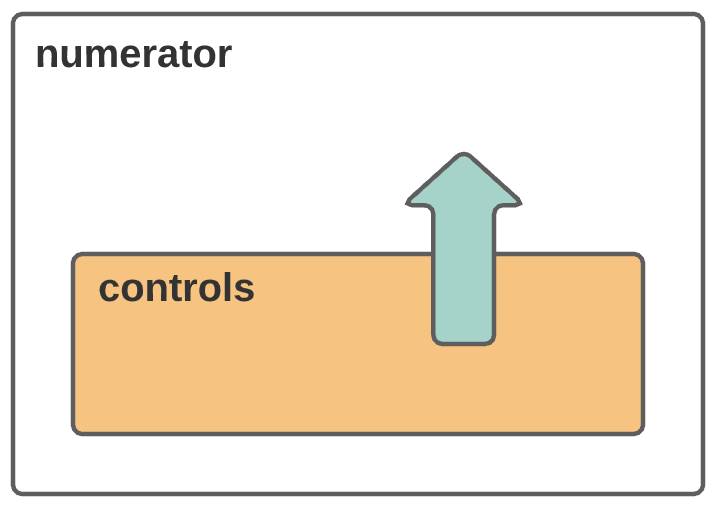
I

This file provides metadata for Salesforce, including the design configuration for components intended for use in Lightning App Builder.

**Third Module**

**Communicate with a Custom Event**

Let's start with a simple scenario. A child component dispatches a custom event that triggers an update in the parent component.



When you send an event, you can choose to send some data and you can allow the event to bubble up through the DOM. Let's start with a simple scenario. A child component dispatches a custom event that triggers an update in the parent component.

**Send Data with the Custom Event**

Next let's have the child pass some data with the event to the parent. The business wants to multiply the count. We'll give them more than one option. Use a custom event to pass data from the child to the parent.

**Use Lightning Data Service to Work with Data**

**Learning Objectives**

After completing this unit, you’ll be able to:

* Build solutions that include lightning-record-\*-form base components.
* Use Lightning Data Service (LDS) wire adapters to read data.
* Use LDS functions to modify data.

## Work with Data in Lightning Web Components

In Lightning web components, there are several ways to work with Salesforce data. Knowing which solution to use for a specific use case lets you write less code, simpler code, and more maintainable code. Using the best solution for each situation also improves the performance of your components and applications.

In this module you learn when and how to use different solutions. We review the easiest but least customizable options first. Later, we consider more complex and more customizable options.