**LWC OSS**

Salesforce’s web development framework to create single page applications for use outside of the Salesforce platform. Similar to React, Vue, or Angular.

Lightning web components are custom elements built using HTML and modern JavaScript.

The Lightning Web Components UI framework uses core Web Components standards and provides only what’s necessary to perform well in browsers. Because it’s built on code that runs natively in browsers, the framework is lightweight and delivers exceptional performance. Most of the code you write is standard JavaScript and HTML.

**Structure of a LWC OSS Application**

* whenever we create a new LWC OSS application with npx create-lwc-app command, we get a default Node.js project
* the contents of this project include
  + node\_modules
    - Node.js applications are meant to be portable by not having any dependence on the software installed on the system they’re running
    - Rather, they include any dependencies that they need in the node\_modules directory
      * These dependencies are called local packages
    - Whenever we create an LWC OSS application, it’s automatically made with the required dependencies in its node\_modules folder
    - If we want to install additional packages, we’ll do so through the use of the npm install command
    - Because this npm install command can also install any listed dependencies that we don’t yet have locally and because we want to keep our Node.js applications as small as possible, we don’t commit the contents of the node\_modules folder to version control
  + src
    - contains the source code that makes up our application (including the LWCs that we create)
    - src holds
      * modules
        + where we place our LWCs
        + the folders within the modules directory represent namespaces and then any component bundles inside of these folders represent individual components in that namespace

these nested folders are what determines the namespace for our custom components that we use when instantiating a custom component inside of another

this means that we control the namespaces of our components by whatever folder we place their component bundle folders inside of

* + - * resources
        + where we can store any media or data that our application needs (e.g. images that we want to display)
        + to reference a file in our resources from a component’s template, we give the relative path to that file from our application’s src directory (e.g. ./resources/nameOfFile.jpg)
      * index.html and index.js
        + these files serve as the entry point of our application
        + i.e. whenever someone views our application, the markup in the index.html file and the code in the index.js file determine what they initially see
  + lwc.config.json
    - this is where we put any special behavior or commands that we want the framework to observe whenever it is building or previewing our applications

**Components** = a reusable piece of user interface that we can use like lego pieces to build up our application.

**Component Bundles** = all the parts required to build a LWC component. required: html, JavaScript files. optional: CSS file. They all must share same name as component, just ending with their respective file types. .css, .js, .html

**Template tag** = html files in our component bundle must contain <template> tags to wrap everything in.

**Composition**

* when we use composition, we’re nesting custom components inside of other components
* when nesting a component inside of another, we’ll instantiate it like a standard HTML element
  + i.e. we’ll just write the name of the component with kebab case
  + e.g.

<template>

<namespace-child-component></namespace-child-component>

</template>

* + will render the component that’s named childComponent (we name components and their files using camelCase)

Sidebar: Casing Conventions

* camelCase
  + the first letter of every word except the first word in a variable name is capitalized
* pascal case
  + the first letter of every word in a variable name is capitalized
* kebab case
  + every letter in the name is lowercase and words are separated by hyphens

**Data binding**: A way to render data through binding a property in JS onto our component. We can use curly braces for the correct syntax. Ex: {myBindedProperty}

**Our html:**

<template> Hi {username}! </template>

**Our javascript:**

@import { LightningElement } from ‘lwc’;

Export default class ClassName extends LightningElement {

userName = ‘some name’;

}

**Template directives = allow additional functionality on certain elements**

**- if:true, if:false = used to conditionally render UI elements. Takes a Boolean value**

**- for:each, for:item, key attribute = used to iterate over data and display that data.**

**- iterator:it = loop through an array with special behavior for the first and last items. We have special properties we can take advantage of.**

**- value = the value of item in the list. Use this property to access the properties of the array. For example = iteratorName.value.propertyName**

**- index = the index of the item in the list**

**- first = a Boolean value indicating whether the item is first in the list**

**- last = a Boolean value indicating whether the item is last in the list**

**Render Multiple Templates**

Can create multiple HTML files in the component bundle. Import them all and add a condition in the render() method to return the correct template depending on the component’s state.

The returned value from the render() method must be a template reference, which is the imported default export from an HTML file.

**Reactivity**

All fields are reactive. If the value of a field changes and the field is used in a template or in the getter of a property used in a template, the component re-renders and the renderedCallback() lifecycle hook is called(see lifecycle hooks below). When a component re-renders, all the expressions used in the template are re-evaluated.

**Decorators = can modify the behavior of a property or function. We have these available:**

**- @track = allows you to have reactivity applied when using an object or an array**

**- @api =** makes our child component’s properties public for use by parent components

**Lifecycle Hooks**

**A lifecycle hook is a JavaScript callback method triggered at a specific phase of a component instance’s lifecycle.**

**connectedCallback() = this lifecycle hook is invoked when a component is inserted into the DOM. To access the host element, use “this”. You can’t access child elements in the component body because they don’t exist yet.**

**import { LightningElement } from 'lwc'**

**export default class New extends LightningElement {**

**connectedCallback() {**

**this.classList.add('new-class');**

**}**

**}**

**Note**

**To check whether a component is connected to the DOM, you can use this.isConnected.**

**disconnectedCallback() = is invoked when a component is removed from the DOM**

**renderedCallback() = called after every render of the component. When a component re-renders, all the expressions used in the template are reevaluated.**

**CSS considerations**

* Css file must be same name as component.
* Cannot use IDs to style components
* (see shadow DOM section below)
* A component’s style sheet can reach up and style its own element. Instead of using a selector for the name of the element, use the “:host” selector.
* We can share a style sheet by using a resource module.
  + Create a blank component with just a .css file
  + You can then reference that css file in other components
  + Just simply use the following syntax in your .css file for the component you want to share the styles with = @import ‘my/nameOfComponentThatHasCssFile’;

**Shadow DOM**

* The shadow DOM encapsulates the internal DOM structure of a web component. It allows the protection of the component from any manipulation.
* One consequence is styling. If we have a child component, it will not inherit styles from the parent as it is in another shadow tree. We also cannot use IDs to style with either.
* We need to use this.template.querySelector() to manipulate the DOM we can’t simply use document.querySelector()

**ES Modules**

https://lwc.dev/guide/es\_modules

**Events**

Events in LWC OSS are a way to communicate between your components in the composition of your app. We can go up our composition tree, and we can go back down the tree as well. To communicate down the component hierarchy we can pass properties to a child via HTML attributes, or call its public methods. Let’s take a look at that now.

EX

todoApp.html:

<template>

<my-todo-item item-name="Milk"></my-todo-item>

<my-todo-item item-name="Bread"></my-todo-item>

</template>

todoItem.html

<template>

<div>{itemName}</div>

</template>

todoItem.js

import { LightningElement, api } from 'lwc';

export default class TodoItem extends LightningElement {

@api itemName;

}

As we can see we use the @api decorator to make our child property public for use by parent components. In this way, we can pass information down the hierarchy. We can also call methods from our children components as well.

EX:

App.html:

<template>

<button onclick={handleclick}>click to call child method!</button>

<my-child></my-child>

</template>

App.js

Import { LightningElement } from ‘lwc’;

Export default class App extends LightningElement {

handleClick() {

this.template.queryselector(‘my-child’).myChildMethod();

}

}

child.js

import { LightningElement, api } from 'lwc';

export default class TodoItem extends LightningElement {

@api myChildMethod() {

Console.log(“this is my child method!”);

};

}

So we have now covered how to traverse down the tree. But what about back up the tree? We can do that by firing off events! Let’s take a look at an example.

EX:

App.html

<template>

<my-child oninputchange={handleInput}></my-child>

{inputMessage}

</template>

App.js

Import { LightningElement } from ‘lwc’;

Export default class App extends LightningElement {

inputMessage;

handleInput(e) {

this.inputMessage = e.detail;

}

}

Child.html

<template>

<input type=”text” onchange={handleChange}></input>

</template>

Child.js

Import { LightningElement } from ‘lwc’;

Export default class Child extends LightningElement {

handleChange(e) {

this.dispatchEvent(new CustomEvent('inputchange', {detail : e.target.value}));

}

}