SALESFORCE
LIGHTNING WEB COMPONENT (LWC)

WiseQuarter Education

LIGHTNING WEB COMPONENT (LWC)

Contents

1. Introduction 2
1.1. Benefits: 2
1.2. Components: 2
2. Creating Lightning Web Component 4
2.1. Svg file (Scalable vector Graphics): 6
2.1.1. SVG in LWC using HTML Markup: 6
2.1.2. SVG using Static Resource 7
3. Data Binding in LWC 11
3.1. One-Way Data Binding 11
3.2. Two-Way Data Binding 12
4. Conditional Rendering 15
5. GETTER & SETTER IN LWC 17
6. Rendering Dom Element 22
7. List Rendering in LWC 24
7.1. for:each 24
7.2. Iterator 27
8. Decorators in Lightning Web Component 28
8.1. @api 28
8.2. @track 30
8.3. @wire 34
9. Component Composition In LWC 37
9.1. Shadow DOM 37
9.2. Composing Components 37
10. Events in LWC 46
10.1. Standard Event communication in Lightning web component (Parent to Child) 46
10.1.1. Public Properties 46
10.1.2. Public Methods 48
10.2. Custom Event Communication in Lightning Web Component (Child to Parent) 50
10.3. Publish Subscriber model in Lightning Web Component (*Two components which doesn't have a direct relation*) 58
10.3.1. Lightning message service 59
10.3.2. Publish-Subscribe Model 68
11. LWC Lifecycle Hook 69
11.1. Constructor() 70
11.2. connectedCallback() 71
11.3. disconnectedCallback() 71
11.4. render () 71
11.5. renderedCallback() 74
11.6. errorCallback() 74
12. LWC Testing with Jest Framework 76
13. LWC Best Practices 81
Resources 84

1. Introduction

Lightning Web Components are an updated web standards-based framework method for creating lightning components on the Salesforce Platform, Lightning Web Components utilize standard tech like **CSS**, **HTML**, and updated **JavaScript** without requiring a set framework, incorporating the latest innovations in JavaScript, including Shadow Document Object Model(DOM), custom elements, and web components (ECMAScript 7 is specifically the updated JavaScript language used).

1.1. Benefits:

The result of integrating specialized services with a standard web stack is:

- Large-scale modular apps are easy to develop.
- Utilizing the most current web functionalities and constructs.
- Transferable skills and a common model.
- The LWC framework can be quickly ramped up by any web developer working with modern JS frameworks.
- The interoperability of components.
- Enhanced performance

1.2. Components:

An LWC is also primarily composed of **HTML** and **JavaScript**. **CSS** is an optional component. However, additional **configuration** files are also included for LWC, which define the metadata values.

Create Lightning Web Components:

A Lightning web component is a reusable custom HTML element with its own API.

- **Define a Component**

  A Lightning web component that renders UI must include an HTML file, a JavaScript file, and a metadata configuration file. The files must use the same name so the framework can autowire them. A service component (library) must include a JavaScript file and a metadata configuration file.

- **HTML Templates**

  The power of Lightning Web Components is the templating system, which uses the virtual DOM to render components smartly and efficiently. It's a best practice to let LWC manipulate the DOM instead of writing JavaScript to do it.

- **CSS**

  To give your component the Lightning Experience look and feel, use Lightning Design System. To go your own way, write your own CSS.

- **Composition**

  You can add components within the body of another component. Composition enables you to build complex components from simpler building-block components.

- **Fields, Properties, and Attributes**

  Declare fields in your component's JavaScript class. Reference them in your component's template to dynamically update content.

- **JavaScript**

  Every component must have a JavaScript file. JavaScript files in Lightning web components are ES6 modules.

- **Access Static Resources, Labels, Internationalization Properties, User IDs, and Form Factors**

  Lightning components can access global Salesforce values, such as labels, resources, and users.

- **Component Accessibility**

  Accessible software and assistive technologies enable users with disabilities to use the products you build. Develop your components so that all users can perceive, understand, navigate, and interact with them.

- **Component Lifecycle**

  Lightning web components have a lifecycle managed by the framework. The framework creates components, inserts them into the DOM, renders them, and removes them from the DOM. It also monitors components for property changes.

Component Folder

To create a component, first create a folder that bundles your component's files.
The folder and its files must have the same name, including capitalization and underscores.

```
myComponent
├──myComponent.html
├──myComponent.js
├──myComponent.js-meta.xml
├──myComponent.css
└──myComponent.svg
```

The folder and its files must follow these naming rules;

- Must begin with a lowercase letter
- Must contain only alphanumeric or underscore characters
- Must be unique in the namespace
- Can't include whitespace
- Can't end with an underscore
- Can't contain two consecutive underscores
- Can't contain a hyphen (dash)

Lightning web components match web standards wherever possible. The HTML standard requires that custom element names contain a hyphen.

Since all Lightning web components have a namespace that's separated from the folder name by a hyphen, component names meet the HTML standard. For example, the markup for the Lightning web component with the folder name widget in the default namespace c is `<c-widget>`.

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>`.**

2. Creating Lightning Web Component

View all --> command Palet --> >SFDX: Create Lightning Web Component (VScode). (Html, javascrip,css and xml files must have the same name)

s1component.js-meta.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<LightningComponentBundle xmlns="http://soap.sforce.com/2006/04/metadata">
    <apiVersion>55.0</apiVersion>
    <isExposed>true</isExposed>
    <targets>
        <target>lightning__AppPage</target>
        <target>lightning__HomePage</target>
        <target>lightning__RecordPage</target>
    </targets>
</LightningComponentBundle>
```

Every component must have a configuration file. The configuration file defines the metadata values for the component, including supported targets and the design configuration for Lightning App Builder and Experience Builder.

The configuration file follows the naming convention *<component>*.js-meta.xml, such as myComponent.js-meta.xml.

isExposed

If isExposed is false, the component isn't exposed to Lightning App Builder or Experience Builder.

To allow the component to be used in Lightning App Builder or Experience Builder, set isExposed to **true** and **define at least one <target>**, which is a type of Lightning page.

targets

Specifies where the component can be added, such as on a type of Lightning Page or in Embedded Service Chat.

If you want your component to appear in the Lightning App Builder or in Experience Builder, specify at least one Lightning page type.

firstComponent.html

```
<template>
    <h1 style="color:red"> My First Component </h1>
    <p class="ilkCssStyle"> Style is from css </p>
</template>
```

Every UI component must have an HTML file with the root tag **<template>.**
Service components (libraries) don't require an HTML file.
The HTML file follows the naming convention ***<component>*.html**, such as **myComponent.html**.
Create the HTML for a Lightning web component **declaratively**, **within the <template> tag**.
The HTML template element contains your component's HTML.

The HTML spec mandates that tags for custom elements (components) aren't self-closing. In other words, custom elements must include separate closing tags. For example, the c-todo-item component nested in this component includes a separate closing **</c-todo-item>** tag.

```
<!-- myComponent.html -->
<template>
    <c-todo-item></c-todo-item>
</template>
```

firstComponent.js

```
import { LightningElement } from 'lwc';


export default class FirstComponent extends LightningElement {
}
```

Every component must have a JavaScript file. If the component renders UI, the JavaScript file defines the HTML element. If the component is a service component (library), the JavaScript file exports functionality for other components to use.

JavaScript files in Lightning web components are ES6 modules. By default, everything declared in a module is local—it's scoped to the module.

To import a class, function, or variable declared in a module, use the import statement. To allow other code to use a class, function, or variable declared in a module, use the export statement.

The JavaScript file follows the naming convention ***<component>*.js**, such as **myComponent.js**.

Every UI component must include a JavaScript file with at least above code.

The core module in Lightning Web Components is lwc. The **import** statement imports **LightningElement** from the **lwc** module.

LightningElement is a custom wrapper of the standard HTML element.

Extend LightningElement to create a JavaScript class for a Lightning web component. *You can't extend any other class to create a Lightning web component.*

The **export default** keywords export a MyComponent class for other components to use. The class must be the default export for UI components. *Exporting other variables or functions in a UI component isn't currently supported.*

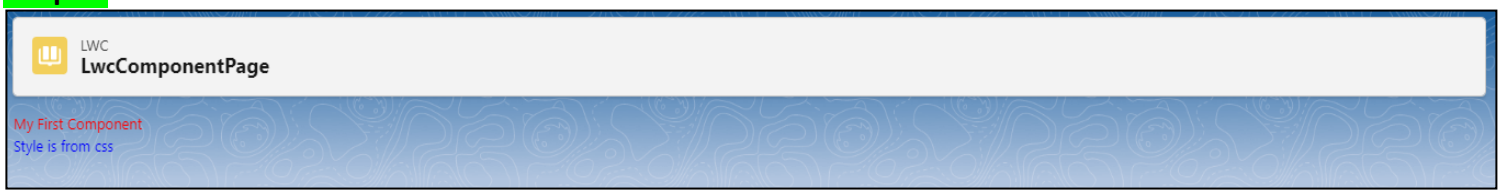
The convention is for the class name to be **Pascal Case**, where the first letter of each word is capitalized. For **myComponent.js**, the class name is **MyComponent**.

firstComponent.css

```
.ilkCssStyle {
  color:blue;
}
```

A component can include a CSS file. Use standard CSS syntax to style Lightning web components.
To style a component, create a style sheet in the component bundle with the same name as the component. If the component is called myComponent, the style sheet is myComponent.css. The style sheet is applied automatically.
To share CSS style rules between components, create a module that contains only a CSS file and a configuration file. Import the module into the CSS files of Lightning web components.
While deploying if .css file is empty, deploy wil be unsuccessful. So empty css file must be deleted before deploying.

Output:



2.1. Svg file (Scalable vector Graphics):

SVG stands for Scalable Vector Graphics. It is a custom icon resource for components used in Lightning App Builder or Community Builder.

if we want to customize this icon to some other icon, we need to create a SVG for our lightning component.

To include an SVG resource as a custom icon for your component, add it to your component's folder. It must be named <component>.svg. If the component is called myComponent, the svg is `myComponent.svg`. You can only have one SVG per folder.

SVG (Scalable Vector Graphics) is an XML-based image format that lets you define lines, curves, shapes, and text. Here's an example of a file that contains a red rectangle, green circle, and white text that says "SVG".

.svg file

```
<svg version="1.1" baseProfile="full" width="300" height="200"
xmlns="http://www.w3.org/2000/svg">
    <rect width="100%" height="100%" fill="red"></rect>
    <circle cx="150" cy="100" r="80" fill="green"></circle>
    <text x="150" y="125" font-size="60" text-anchor="middle" fill="white">SVG</text>
</svg>
```

There are two ways to include SVG in LWC (Lightning Web Component):

2.1.1. SVG in LWC using HTML Markup:

To include an SVG resource in your HTML template, enclose it in `<template>` tags like any other element.

```
<template>
    <svg version="1.1" baseProfile="full" width="300" height="200"
xmlns="http://www.w3.org/2000/svg">
        <rect width="100%" height="100%" fill="red"></rect>
        <circle cx="150" cy="100" r="80" fill="green"></circle>
        <text x="150" y="125" font-size="60" text-anchor="middle" fill="white">SVG</text>
    </svg>
</template>
```

2.1.2. SVG using Static Resource

To include SVG in Lightning Web Component using Static Resource:

1. In the **SVG** file, add an **id** attribute to the **<svg>** tag

```
<svg version="1.1" baseProfile="full" width="300" height="200"
xmlns="http://www.w3.org/2000/svg" id="logo">
    <rect width="100%" height="100%" fill="red"></rect>
    <circle cx="150" cy="100" r="80" fill="green"></circle>
    <text x="150" y="125" font-size="60" text-anchor="middle" fill="white">SVG</text>
</svg>
```

2. Upload the **SVG** resource to your org as **a static resource**.
3. In your component's JavaScript file, import the static resource. Define a field that concatenates the reference to the static resource with its id.

```
// myComponent.js
import { LightningElement } from 'lwc';

import SVG_LOGO from '@salesforce/resourceUrl/logo';

export default class myComponent extends LightningElement {
    svgURL = `${SVG_LOGO}#logo`
}
```

4. Finally, add the use tag inside the svg tag in HTML file and provide the URL generated in the previous step.

```
<!-- myComponent.html -->
<template>
    <svg xmlns="http://www.w3.org/2000/svg" width="300" height="200">;
        <use xlink:href={svgURL}></use>
    </svg>
</template>
```

Note: While deploying if .svg file is empty, deploy wil be unsuccessful. So empty svg file must be deleted before deploying.

Example: Writing a message inside component.

messageComponent.js-meta.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<LightningComponentBundle xmlns="http://soap.sforce.com/2006/04/metadata">
    <apiVersion>55.0</apiVersion>
    <isExposed>true</isExposed>
    <targets>
        <target>lightning__AppPage</target>
        <target>lightning__HomePage</target>
        <target>lightning__RecordPage</target>
    </targets>
</LightningComponentBundle>
```

messageComponent.html

```
<template>
    <lightning-card class="myClass" title="Self Study">
      I'm learning LWC
    </lightning-card>
</template>
```

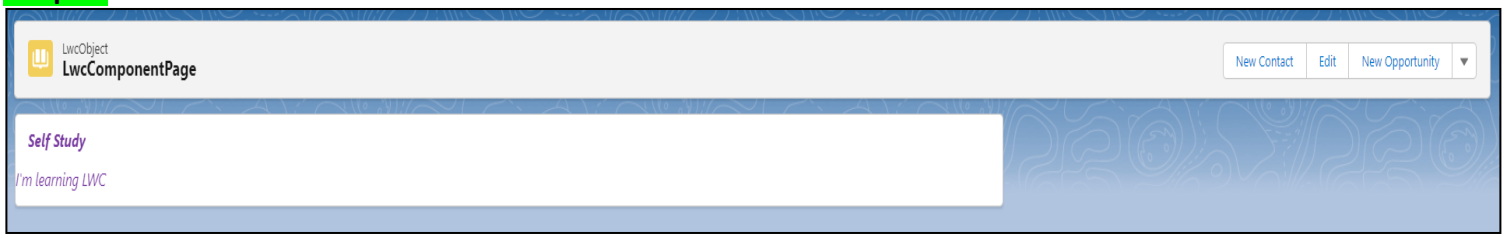
messageComponent.js

```
import { LightningElement } from 'lwc';

export default class MessageComponent extends LightningElement {
}
```

messageComponent.css

```
lightning-card{
  color:rgb(120, 59, 161);
  font-size: larger;
}
.myClass{
  font-style: italic;
}
```

Output:



Example: (xml file is the same with the previous example) (image file should be inside the component folder)

imgComponent.html

```
<template>
    <div> Title: salesforce Developer </div>
    <div class="myClass">Subject: LWC</div>
    <div>
      <img src=" https://wp.salesforce.com/en-us/wp-
content/uploads/sites/4/2021/07/salesforce-logo.jpg?w=1024" >
    </div>
</template>
```

imgComponent.js

```
import { LightningElement } from 'lwc';

export default class ImgComponent extends LightningElement {

}
```

imgComponent.css

```
div{
  color: rgb(11, 11, 12);
  text-align: center;
  font-weight: bold;
  font-size: xx-large;
}
.myClass{
  font-style: italic;
}
```

Output:

Example: (Getting some variables from Javascript file)

imgComponent.html

```
<template>
    <div> {Title} </div>
    <div class="myClass">Subject: {Subject}</div>
    <div>
      <img class="imgClass" src= {imgUrl} >
    </div>
</template>
```

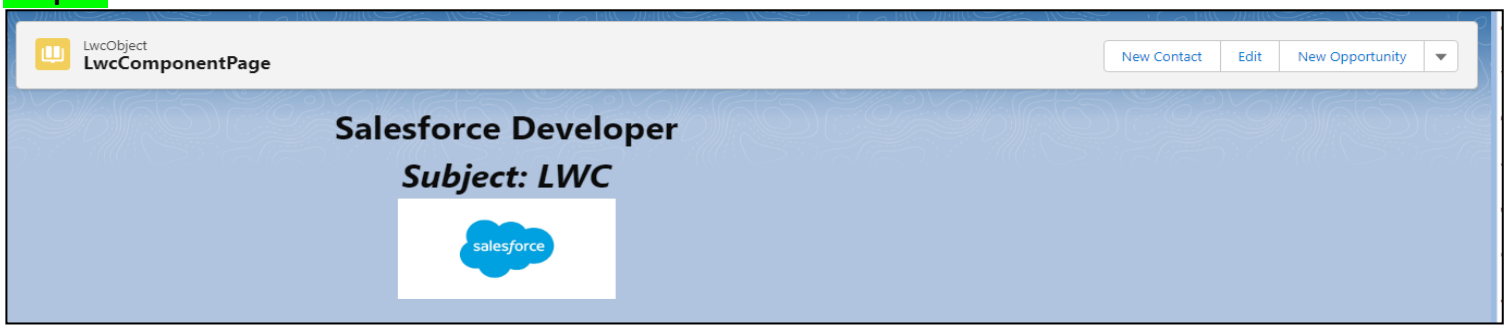
imgComponent.js

```
import { LightningElement } from 'lwc';

export default class ImgComponent extends LightningElement {
 Title = 'Salesforce Developer';
 Subject = 'LWC';
 imgUrl = 'https://wp.salesforce.com/en-us/wp-content/uploads/sites/4/2021/07/salesforce-logo.jpg?w=1024'
}
```

imgComponent.css

```
div{
  color: rgb(11, 11, 12);
  text-align: center;
  font-weight: bold;
  font-size: xx-large;
}
.myClass{
  font-style: italic;
}
.imgClass{
  width: 200px;
  height: 100px;
}
```

Output:



3. Data Binding in LWC

Data binding is the synchronization of data between business logic and view of the application.

In other words: Data binding in the Lightning web component is the synchronization between the controller(Javascript) and the template(HTML).

There are two types of data-binding:

1. One-Way Data Binding
2. Two-Way Data Binding

3.1. One-Way Data Binding

One-way data binding is a situation where information flows in only one direction in our case from the controller to the template(HTML).

binding1.js-meta.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<LightningComponentBundle xmlns="http://soap.sforce.com/2006/04/metadata">
    <apiVersion>55.0</apiVersion>
    <isExposed>true</isExposed>
    <targets>
        <target>lightning__AppPage</target>
        <target>lightning__HomePage</target>
        <target>lightning__RecordPage</target>
    </targets>
</LightningComponentBundle>
```

binding1.html

```
<template>
  <lightning-card title="Data Binding" icon-name="utility:light_bulb">
    <hr />
    <div>
      <h1>1. {subject1} </h1>
    </div>
  </lightning-card>
</template>
```

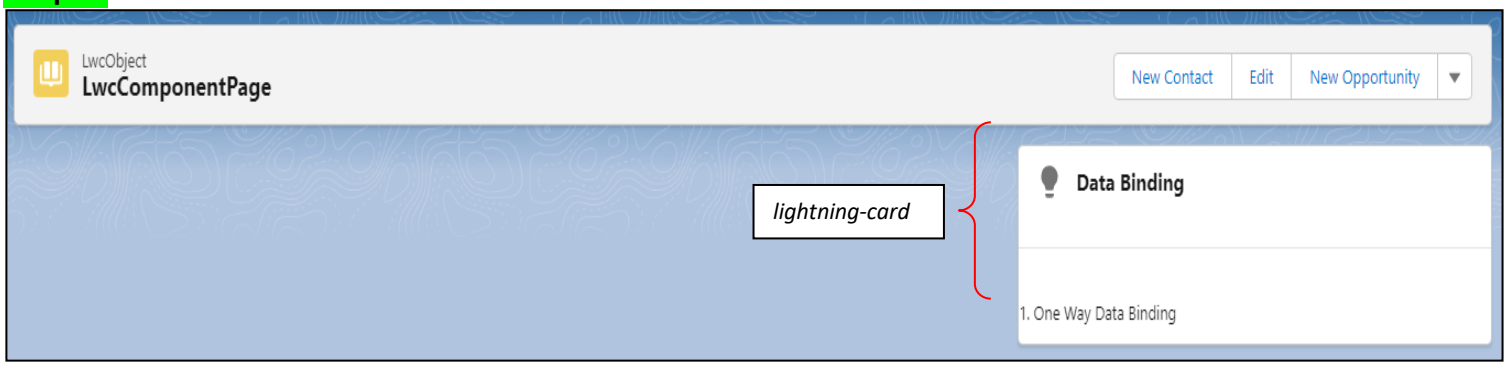
- <hr> *tag is used to specify a paragraph-level thematic break in HTML*
- *icon-name="utility:light_bulb" is for bulb icon.*
- Inside the <template>, we have used the lightning-card
- Inside the *lightning-card* We have created a <h1> tag with text and property `subject1` binding.
- {subject1} - To bind a property in a template we have to surround the property with curly braces

binding1.js

```
import { LightningElement } from 'lwc';

export default class Binding1 extends LightningElement {
  subject1 = "One Way Data Binding";
}
```

- In the first line, we are importing **LightningElement** from **lwc** module
- After that, we are creating a class binding1 ***(the class name always be pascal case)***
- Within the class, we have to define a property **subject1** and assign a value **"One Way Data Binding"** to it.
- So when a component loads, it initialize the variable **subject1** with value **"One Way Data Binding"**. After that, when HTML starts rendering in the browser, **lwc** engine looks for the **{}** tags and replace the **subject1** inside the curly braces with the properties defined inside the class binding1.

Output:



3.2. Two-Way Data Binding

Two-way data binding in LWC will help users to exchange data from the controller to the template and form template to the controller. It will help users to establish communication bi-directionally.

- The Lightning Web Components programming model has given us some decorators that add functionality to property or function.
- **@track** is the decorator that helps us to track a private property's value and re-render a component when it changes.
- Tracked properties are also called private reactive properties.
- **@track** helps us to achieve two-way data binding
- **@track** is powerful, but remember, track a property only if you want the component to re-render when the property's value changes. Don't track every private property.

bindingTwoWay.js-meta.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<LightningComponentBundle xmlns="http://soap.sforce.com/2006/04/metadata">
    <apiVersion>55.0</apiVersion>
    <isExposed>true</isExposed>
    <targets>
        <target>lightning__AppPage</target>
        <target>lightning__HomePage</target>
        <target>lightning__RecordPage</target>
    </targets>
</LightningComponentBundle>
```

bindingTwoWay.html

```
<template>
    <lightning-card title="2. Two Way Binding">
       <lightning-input label="First Name" value="Text here"
onchange={messageHandler1}></lightning-input>
       {message1}
       <lightning-input label="Last Name" value="Text here"
onchange={messageHandler2}></lightning-input>
       {message2}
       <hr>
       <h3>Your Full Name is {message1} {message2}</h3>
    </lightning-card>
</template>
```

- we have created two input box **First Name** and **Last Name** using **lightning-input** inside the **lightning-card**.
- {message1}- is used to bind **First Name**. And we will see the value under the **lightning-input**
- {message2}- is used to bind **Last Name**. And we will see the value under the **lightning-input**
- we have defined an event handler called **onchange** that is bind to **messageHandler1** and **messageHandler2** which gets triggered on every key up.
- We have used the <**h3**> tag to see the Full Name.

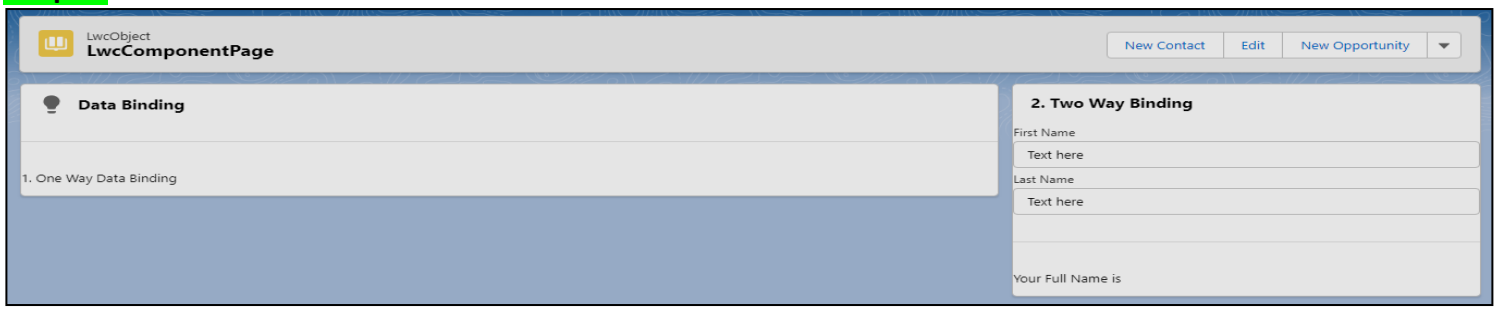
bindingTwoWay.js

```
import { LightningElement, track} from 'lwc';

export default class BindingTwoWay extends LightningElement {
  @track message1;
  @track message2;
  messageHandler1(event){
      this.message1= event.target.value;
  }
  messageHandler2(event){
    this.message2= event.target.value;
  }
}
```

- In the first line, we are importing **LightningElement** and track from **lwc** module
- After that, we are creating a class bindingTwoWay (*the class name always be pascal case*)
- Within the class, we have to define two properties **message1** and **message2**.
- @track decorator decorates both properties.
- We have defined two methods (messageHandler1 and messageHandler2) that take the value from the textbox and update the property based on the input box name.

- In this example, the user will give input and based on that input the javascript will get that value to compute and display the output to the user on HTML again.
- And the output will dynamically change based on the user input with the help of @track property as this is the private property to reflect the input change on the Screen.

Output:



4. Conditional Rendering

- Conditional rendering in the lightning web component (lwc) is a way to display components or elements based on a specific condition.
- Conditional rendering allows you to render different lwc components or elements if a condition is met.

Example:

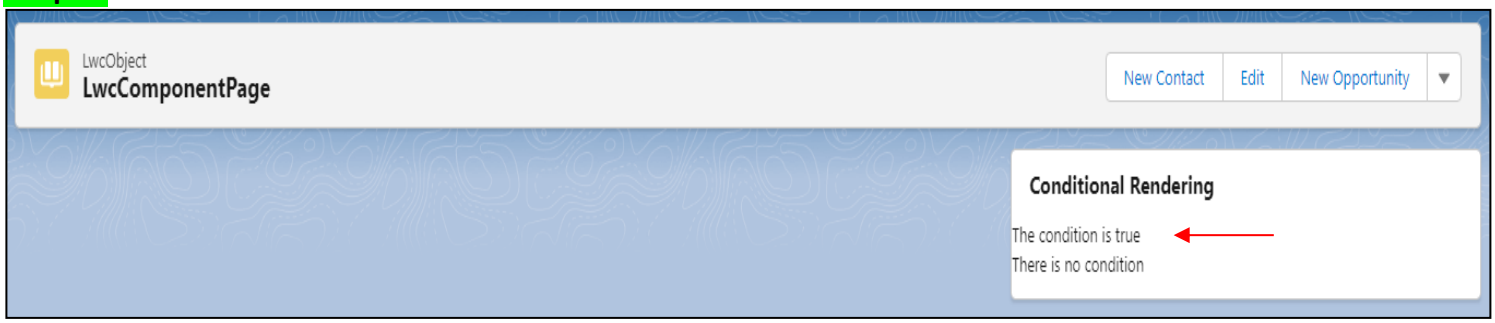
conditionalRendering1.html

```
<template>
    <lightning-card title="Conditional Rendering">
          <div if:true={condition}> The condition is {condition}</div>
          <div if:false={condition}> The condition is {condition}</div>
          <div>There is no condition</div>
    </lightning-card>
 </template>
```

conditionalRendering1.js

```
import { LightningElement } from 'lwc';


export default class ConditionalRendering1 extends LightningElement {
  condition= true;
}
```

Output:



Example:

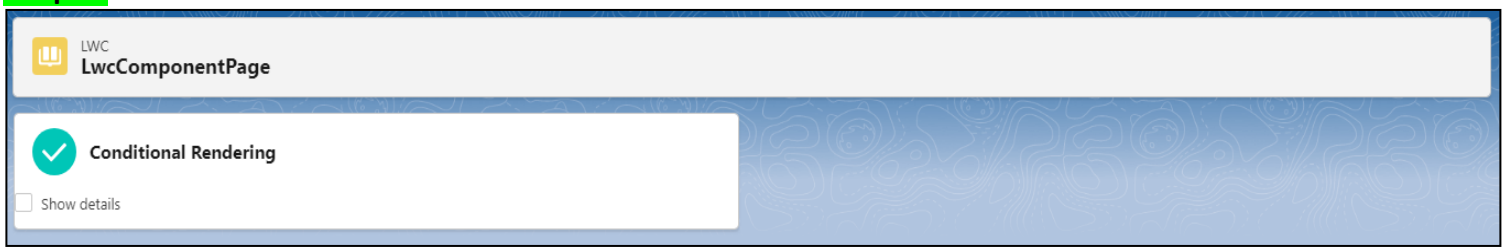
conditionalRendering2.html

```
<template>
  <lightning-card title="Conditional Rendering" icon-name="action:approval">
    <div>
        <lightning-input type="checkbox" label="Show details"
onchange={handleChange}></lightning-input>
        <template if:true={DetailsVisibility}>
            <div class="slds-text-align_center ">
                when details are uploaded, it'll be shown :)
            </div>
        </template>
    </div>
</lightning-card>
</template>
```

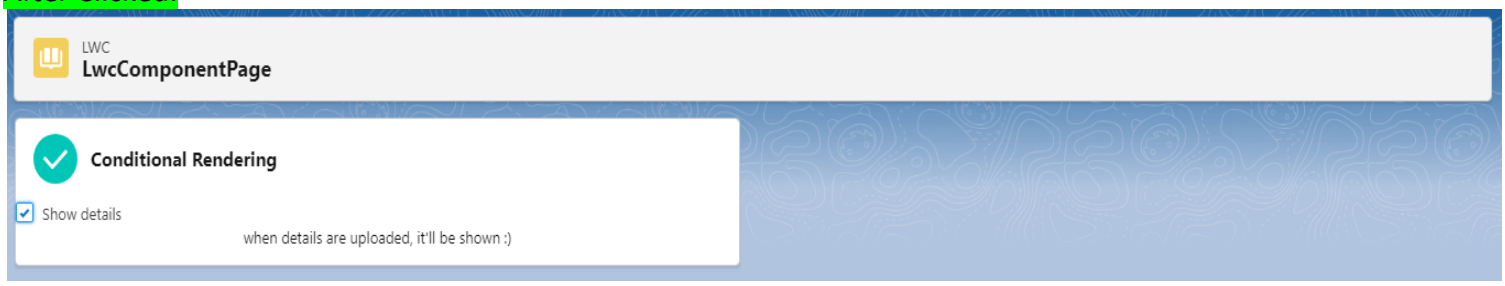
conditionalRendering2.js

```
import { LightningElement } from 'lwc';

export default class ConditionalRendering2 extends LightningElement {
  DetailsVisibility = false;

  handleChange(event) {
      this.DetailsVisibility = event.target.checked;
  }
}
```

Output:



After Clicked:



5. GETTER & SETTER IN LWC

- GET method is used in conjunction with the set method to create a property that can be read and written to by the component.
- The **Get** method is used to return the **current value** of the property, while the **Set** method is used to **update the value** of the property.
- The property defined using the get method can be accessed in the template using expressions like {propertyName}.
- It's important to note that LWC automatically tracks the property value changes and re-renders the component if it changes.

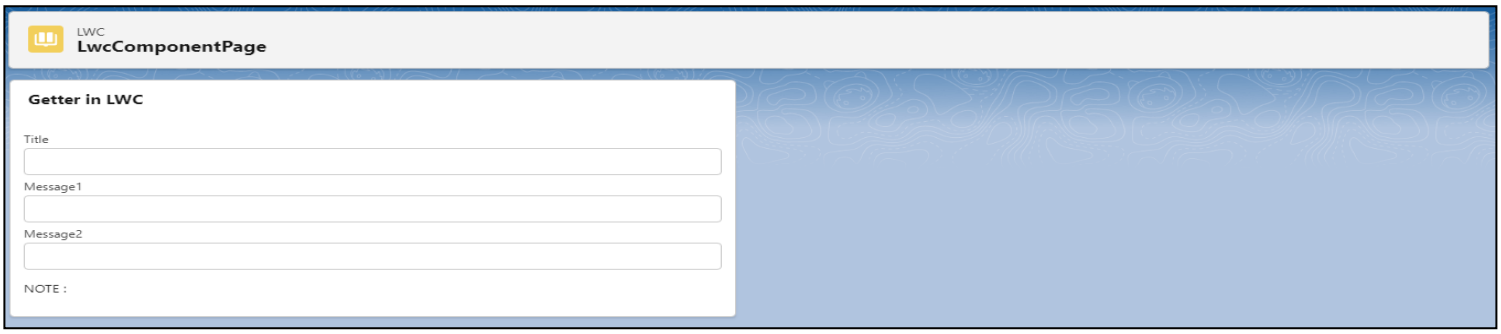
getterComponent.html

```
<template>
  <lightning-card title="Getter in LWC">
    <div class="slds-var-p-around_small">
       <div>
         <lightning-input label="Title" type="text" value={title}
onchange={titleChanged}></lightning-input>
       </div>

       <div>
<lightning-input label="Message1" type="text" value={message1}
onchange={message1Changed}></lightning-input>
       </div>

       <div>
<lightning-input label="Message2" type="text" value={message2}
onchange={message2Changed}></lightning-input>
       </div>

       <div class="slds-var-p-top_small">
           NOTE : {allMessage}
       </div>

    </div>
  </lightning-card>
</template>
```

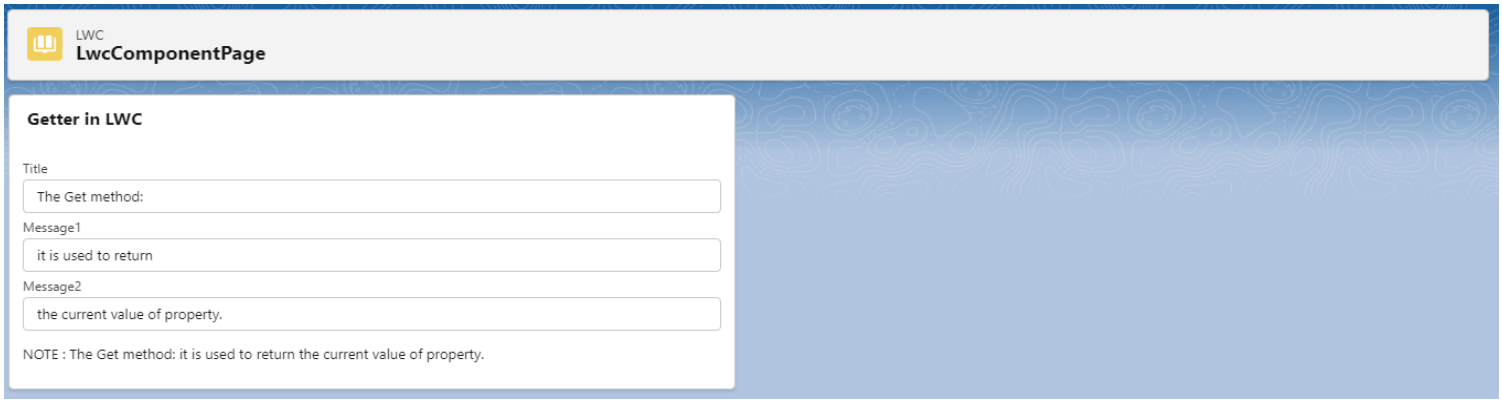
getterComponent.js

```
import { LightningElement } from 'lwc';
export default class GetterComponent extends LightningElement {
    title='';
    message1='';
    message2='';

    titleChanged(event) {
        this.title=event.target.value;   // instead of target, details can be used.
    }
    message1Changed(event) {
        this.message1=event.target.value;
    }
    message2Changed(event) {
        this.message2=event.target.value;
    }

    get allMessage() {
        return this.title + ' ' + this.message1 + ' ' + this.message2;
  // return `${this.title} ${this.message1} ${this.message2}`; (Another return type writing)
    }
}
```

Output:



After inputs has been written:



Example: Sum of written numbers into inputs.

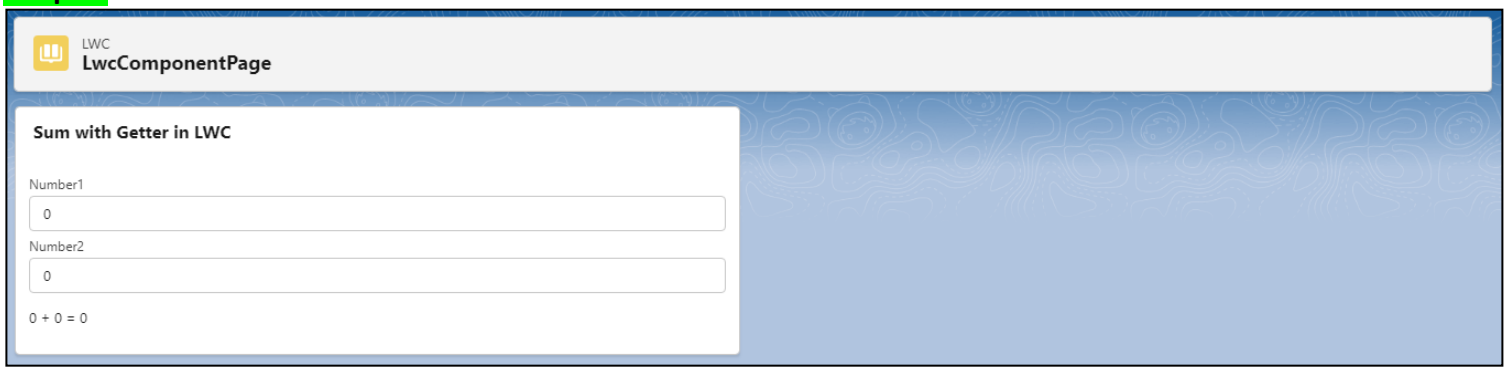
sumWithGetter.html

```
<template>
  <lightning-card title="Sum with Getter in LWC">
    <div class="slds-var-p-around_small">
      <div>
   <lightning-input label="Number1" type="text" value={number1}
onchange={number1Changed}></lightning-input>
      </div>

      <div>
   <lightning-input label="Number2" type="text" value={number2}
onchange={number2Changed}></lightning-input>
      </div>

      <div class="slds-var-p-top_small">
       {number1} + {number2} = {sumOfNumber}
      </div>
    </div>
  </lightning-card>
</template>
```

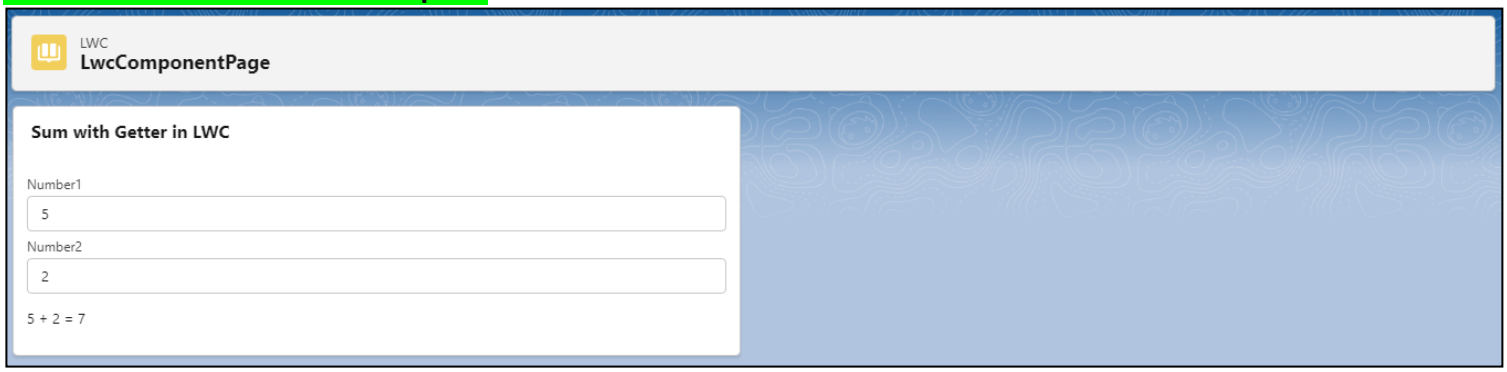
sumWithGetter.js

```
import { LightningElement } from 'lwc';

export default class SumWithGetter extends LightningElement {
  number1='0';
  number2='0';

  number1Changed(event) {
      this.number1=event.target.value;   // instead of target, details can be used.
  }
  number2Changed(event) {
      this.number2=event.target.value;
  }
  get sumOfNumber() {
      //return parseInt(this.number1)+parseInt(this.number2);

  return `${parseInt(this.number1)+parseInt(this.number2)}`;
  }
}
```

Output:



After numbers are written to inputs:



Example: (Setter Method)

- If you write a setter for a public property, you must also write a getter. Annotate either the getter or the setter with `@api`, but not both. It's a best practice to annotate the getter.
- To hold the property value inside the getter and setter, use a field. This example uses `outputMessage`.

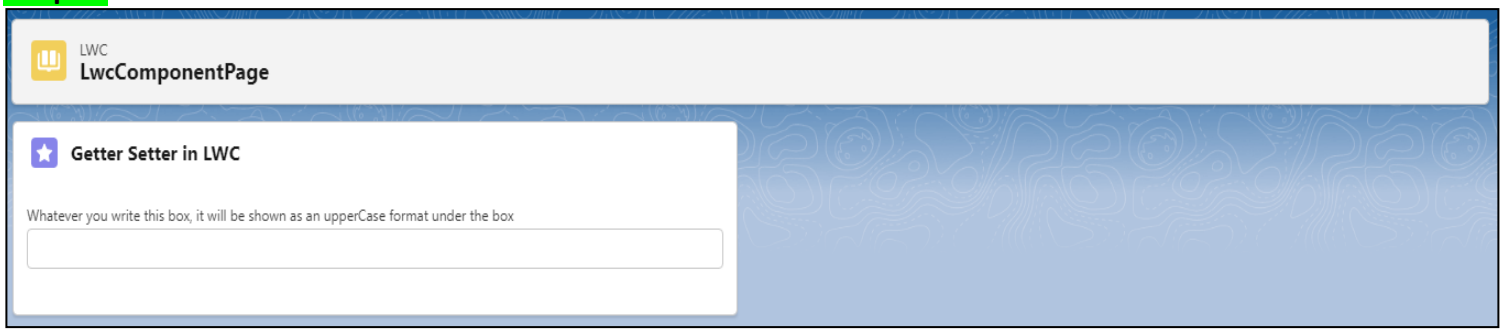
getterSetter.html

```
<template>
  <lightning-card title = "Getter Setter in LWC" icon-name="custom:custom11">
      <lightning-layout>
        <lightning-layout-item flexibility="auto" padding="around-small">

            <div class="padding-around:small">
              <lightning-input label="Whatever you write this box, it will be shown as an
upperCase format under the box" value={message} onchange={handleMessage}></lightning-input>
            </div>

            <div class="padding-around:small slds-m-top_small">
              <lightning-formatted-text value={outputMessage} ></lightning-formatted-text>
            </div>
        </lightning-layout-item>
      </lightning-layout>
  </lightning-card>
</template>
```

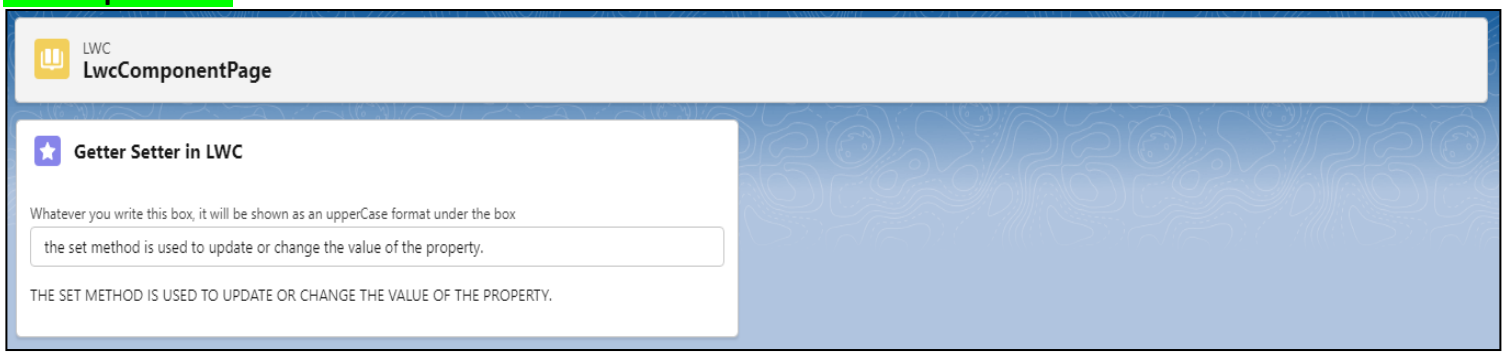
getterSetter.js

```
import { LightningElement, api } from 'lwc';

export default class GetterSetter extends LightningElement {
 message1;
  outputMessage;
  @api
  get message(){
      return this.message1;
  }

  set message(val){
     this.outputMessage = val.toUpperCase();
  }
  handleMessage(event){
      this.message = event.target.value;
  }
}
```

Output:



After input filled:



6. Rendering Dom Element

- The power of Lightning web component is templating system, which uses the virtual DOM to render components smartly and efficiently.
- To render HTML conditionally, add the `if:true|false` directive to a nested `<template>` tag that encloses the conditional content.
- The `if:true|false={property}` directive binds data to the template and removes and inserts DOM elements based on whether the data is a true or false value

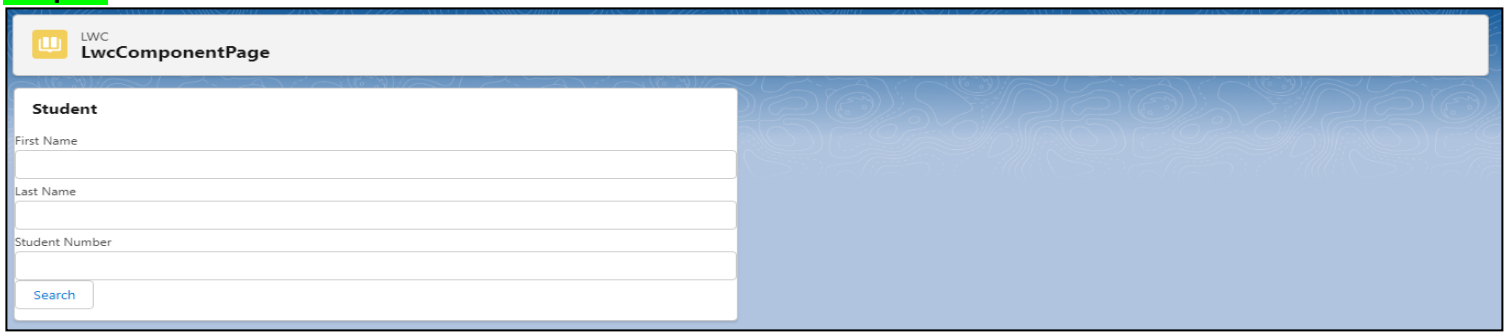
renderingDomElement.html

```
<template>
  <template if:true={display}>
    <lightning-card title="Student">
      <div>
        <lightning-input label="First Name" value={firstName}
onchange={firstNameHandler}></lightning-input>
        <lightning-input label="Last Name" value={lastName}
onchange={lastNameHandler}></lightning-input>
        <lightning-input label="Student Number" value={studentNumber}
onchange={studentNumberHandler}></lightning-input>

        <lightning-button label="Search" onclick={searchHandler}></lightning-button>
      </div>
    </lightning-card>
  </template>
  <template if:false={display}>
    <lightning-card title="Student Details">
      <div>
        <h1 style="font-weight: bolder;">{firstName} {lastName} ({studentNumber})</h1>
        <p>Phone : </p>
        <p>Email : </p>
        <p>Address : </p>
        <img
src="https://us.123rf.com/450wm/pandavector/pandavector1705/pandavector170501741/77587996-
avatar-of-a-man-with-orange-hair.avatar-and-face-single-icon-in-cartoon-style-rater,bitmap-
symbol-stock-illustration..jpg">
      </div>
    </lightning-card>
  </template>
</template>
```

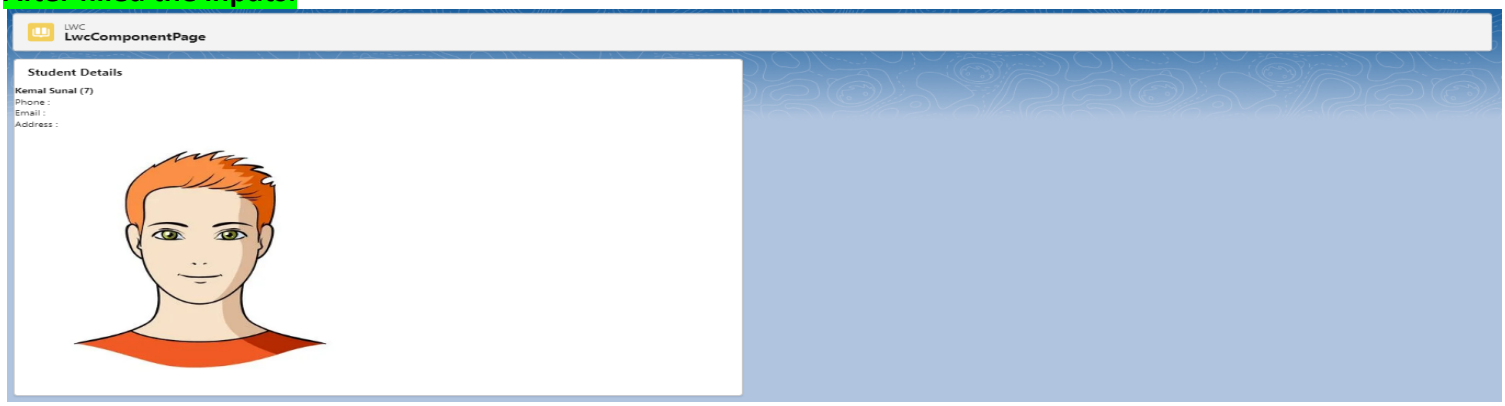
renderingDomElement.js

```
import { LightningElement } from 'lwc';

export default class RenderingDomElement extends LightningElement {
  display=true;
  firstName;
  lastName;
  studentNumber;

  firstNameHandler(event){
    this.firstName=event.target.value;
  }
  lastNameHandler(event){
    this.lastName=event.target.value;
  }
  studentNumberHandler(event){
    this.studentNumber=event.target.value;
  }
  searchHandler(){
    this.display=false;
  }
}
```

Output:



After filled the inputs:



7. List Rendering in LWC

- To render a list of items in Lightning web components(lwc), we use **`for:each`** or **`iterator`** directives.

7.1. for:each

- When using the **for:each** directive, use **for:item**="currentItem" to access the current item.
- To assign a key to the first element in the nested template, use the **key={uniqueId}** directive.
- you can use them **for:index="index"** to access the current item index, it is optional
- The **key** here is the unique value to identify your record from the list.

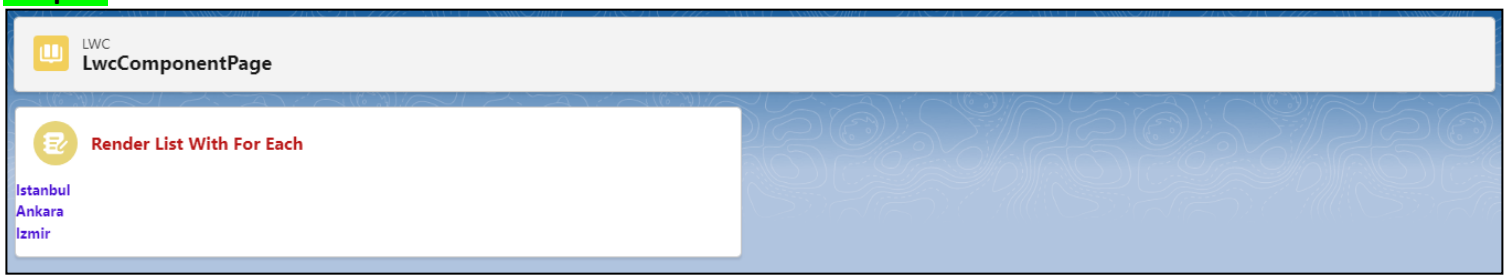
listRenderingCity.html

```
<template>
  <lightning-card title="Render List With For Each" icon-name="action:new_note"
style="color:rgb(185, 23, 23)">
      <ul>
          <template for:each={cities} for:item="citiname">
              <li key={citiname.Id}>
                  <b style="color:rgb(72, 13, 211)">{citiname.Name}</b>
              </li>
          </template>
      </ul>
  </lightning-card>
</template>
```

listRenderingCity.js

```
import { LightningElement } from 'lwc';

export default class ListRenderingCity extends LightningElement {
  cities = [
    {
        Id: 1,
        Name: 'Istanbul',
    },
    {
        Id: 2,
        Name: 'Ankara',
    },
    {
        Id: 3,
        Name: 'Izmir',
    },
  ];
}
```

Output:



Example:

listRenderingCourses.html:

```
<template>
    <lightning-card class="myfont">
      <h1 style="background-color: #5B318C;" class="myTitle">Wise Quarter Courses</h1>
          <template for:each={courses} for:item="crs">
            <div key={crs.Id} class="mydiv">
                Name : {crs.Name} <br>
                Website : {crs.Website}
            </div>
        </template>
      </lightning-card>
 </template>
```

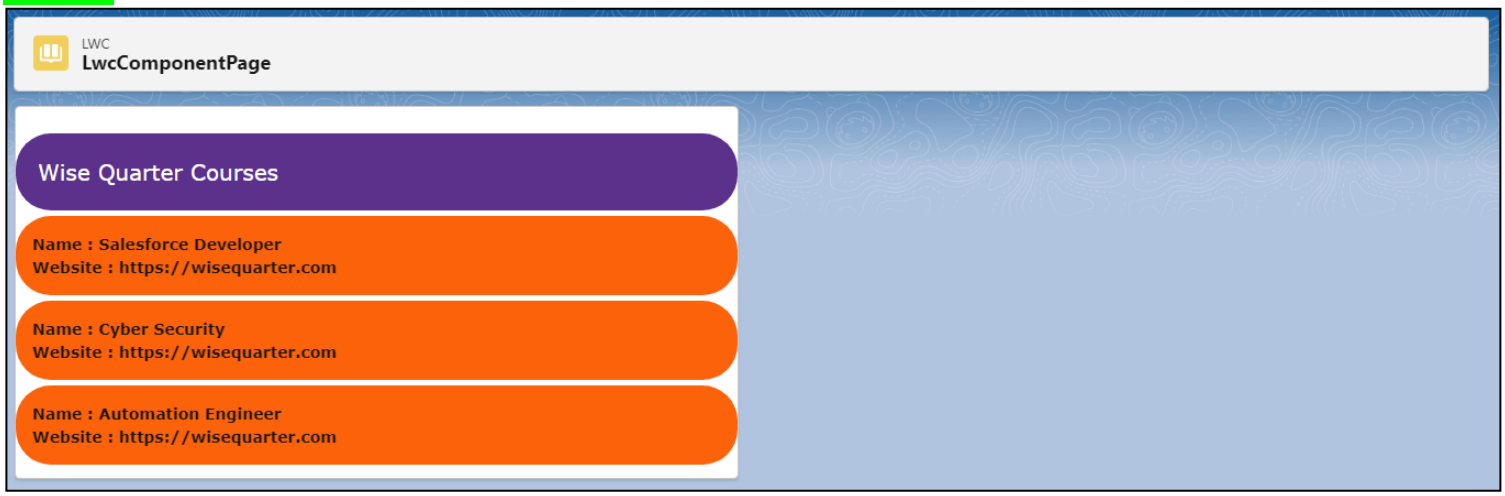
listRenderingCourses.css:

```
.myfont{
  font-family:Verdana, Geneva, Tahoma, sans-serif;
  font-weight: lighter;
  font-size: 14px;
}
.mydiv{
  font-weight: bold;
   padding: 15px;
   background-color : #FC620A;
   margin-top: 5px;
   border-radius:  2rem;
}
.myTitle{
  color: white;
  font-size: 20px;
  font-weight:bolder;
  position:relative;
  padding: 20px;
  border-radius:  2rem;
}
```

listRenderingCourses.js:

```
import { LightningElement } from 'lwc';

export default class ListRenderingCourses extends LightningElement {
  courses = [
    {
        Id: 1,
        Name: 'Salesforce Developer',
        Website: 'https://wisequarter.com',
    },
    {
        Id: 2,
        Name: 'Cyber Security',
        Website: 'https://wisequarter.com',
    },
    {
        Id: 3,
        Name: 'Autommation Engineer',
        Website: 'https://wisequarter.com',
    },
  ];
}
```

OUTPUT:



7.2. Iterator

If you want add special behavior to the **first** or **last** item in a list, use the **iterator** directive, **iterator:iteratorName={array}**. Use the iterator directive on a **template** tag.

Use **iteratorName** to access these properties:

- **value**—The value of the 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 this item is the first item in the list.
- **last**—A boolean value indicating whether this item is the last item in the list.

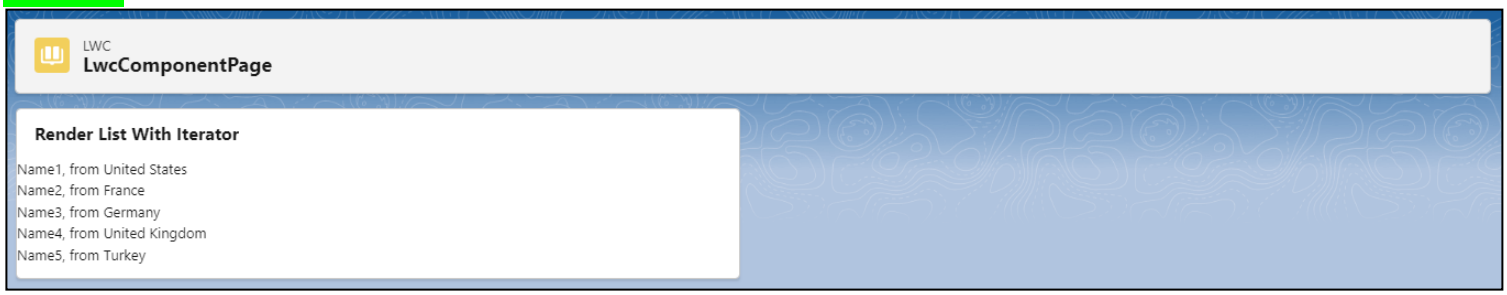
listRenderingCountry.html

```
<template>
  <lightning-card title="Render List With Iterator">
    <template iterator:item={students}>
        <div key={item.value.Id}>
          <div if:true={item.first}></div>
          {item.value.Name}, from
          {item.value.Country}
          <div if:true={item.last}></div>
        </div>
    </template>
  </lightning-card>
</template>
```

listRenderingCountry.js

```
import { LightningElement } from 'lwc';

export default class ListRenderingCountry extends LightningElement {
  students=[
    {Id:1, Name:'Name1', Country:'United States'},
    {Id:2, Name:'Name2', Country:'France'},
    {Id:3, Name:'Name3', Country:'Germany'},
    {Id:4, Name:'Name4', Country:'United Kingdom'},
    {Id:5, Name:'Name5', Country:'Turkey'}
  ]
}
```

OUTPUT:



8. Decorators in Lightning Web Component

- The Lightning Web Components programming model has three decorators that add functionality to a property or function.
- The ability to create decorators is part of ECMAScript, but these three decorators are unique to Lightning Web Components.
- They are used to dynamically alter or modify the behavior of a property or function.
- Decorators provides developers with a set of reusable components and tools for building responsive user interfaces.
- To use a decorator, import it from the lwc module and place it before the property or function.

There are three types of Decorators in Lightning web components:

1. Api (@api)
2. Track (@track)
3. Wire (@wire)

8.1. @api

- The **@api** decorator is used to expose a public property or method on a LWC component. This allows other components to access and interact with the decorated property or method.
- A developer that uses the component in its HTML markup can access the component's public properties.
- To expose a public method, decorate it with **@api**. Public methods are part of a component's API.
- To communicate with other components, owner and parent components can call JavaScript methods on child components.
- To use **@api** we have to import it first from lwc. Then add **@api** before the method, function or property.

```
import { LightningElement, api } from 'lwc';
export default class MyComponent extends LightningElement{
    @api
    message;
}
```

Note:

- while deploying components through the Vscode, child component should be deployed to org before the parent component. And no need to make 'isExposed' to true for child component.
- When we call child component inside the parent component we use **kabab case** to represent the name of it. we call the child component as **<c-child-component>** in the parent syntax. **<c-** this is a default namespace.
- In other words: Property names in JavaScript are in camel case while HTML attribute names are in kebab case (dashseparated) to match HTML standards.

Example:

apiParent.js-meta.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<LightningComponentBundle xmlns="http://soap.sforce.com/2006/04/metadata">
    <apiVersion>56.0</apiVersion>
    <isExposed>true</isExposed>
    <targets>
        <target>lightning__AppPage</target>
        <target>lightning__HomePage</target>
        <target>lightning__RecordPage</target>
    </targets>
</LightningComponentBundle>
```

Take care of that:
On parent component,
<isExposed>true</isExposed>
On child component,
<isExposed>false</isExposed>

apiParent.html

```
<template>
      <c-api-child my-message="my first message"></c-api-child>
      <c-api-child my-message="my second message"></c-api-child>
</template>
```

apiParent.js

```
import { LightningElement } from 'lwc';

export default class ApiParent extends LightningElement {

}
```

To call child component from parent component's html,
the syntax is :
<c-api-child> </c-api-child>
*** we can call more than once.

apiChild.js-meta.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<LightningComponentBundle xmlns="http://soap.sforce.com/2006/04/metadata">
    <apiVersion>56.0</apiVersion>
    <isExposed>false</isExposed>
</LightningComponentBundle>
```

apiChild.html

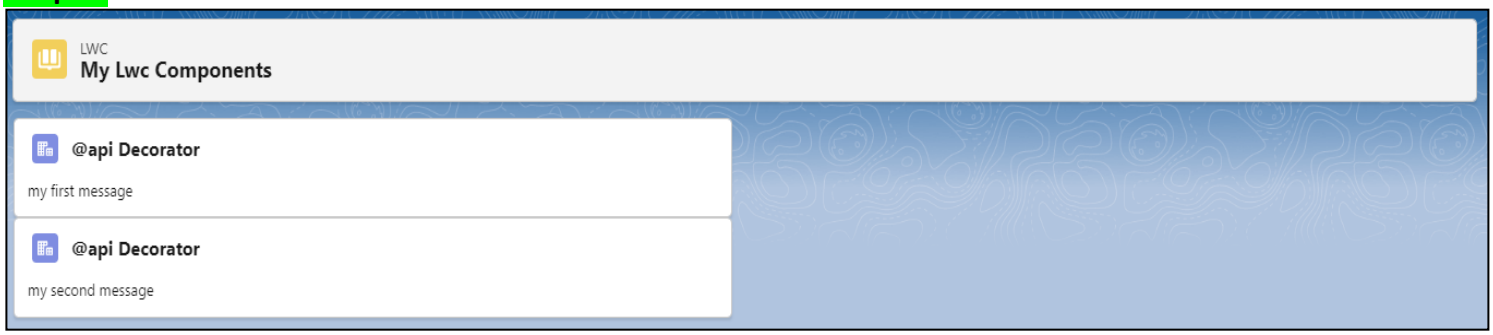
```
<template>
  <lightning-card title="@api Decorator" icon-name="standard:account">
    <p class="slds-p-horizontal_small">{myMessage}</p>
  </lightning-card>
</template>
```

apiChild.html

```
import { LightningElement,api } from 'lwc';

export default class ApiChild extends LightningElement {
  @api
  myMessage = 'Message';
}
```

Take care of the name cases
On parent component's HTML:
my-message
On child component's HTML:
{myMessage}
On child component's JS:
myMessage

Output:



Note: instead of " `myMessage = 'Message';` " in child component, " `my-message=""` " is seen on parent component.

8.2. @track

- The **@track** decorator is used to track changes to a property in a LWC component. This allows the component to automatically rerender itself whenever the value of the decorated property changes.
- If a field's value changes, and the field is used in a template or in a getter of a property that's used in a template, the component rerenders and displays the new value. If a field is assigned an object or an array, the framework observes some changes to the internals of the object or array, such as when you assign a new value.
- To tell the framework to observe changes to the properties of an object, decorate the field with **@track**.
- we can't access **@track** properties from outside as they are private and only accessible within its component only.
- **To use @track we have to import it first** from lwc. Then add **@track** before the method, function or property.
- *The **@track** Decorator Is No Longer Required for Lightning Web Components*.
- Before Spring '20, to make a field reactive, you had to decorate it with **@track**. You see this approach used in older code samples, and it's still supported.
- There is still one use case for **@track**. **When a field contains an object or an array**, there's a limit to the depth of changes that are tracked. To tell the framework to observe changes to the properties of an object or to the elements of an array, decorate the field with **@track**.
- Without using **@track**, the framework observes changes that assign a new value to a field. If the new value is not === to the previous value, the component rerenders.

```
import { LightningElement, track } from 'lwc';
export default class MyComponent extends LightningElement{
    @track
    message;
}
```

Example 1: (without @track. Because no need to use when field is not an array or object)

trackDecorator.js-meta.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<LightningComponentBundle xmlns="http://soap.sforce.com/2006/04/metadata">
    <apiVersion>56.0</apiVersion>
    <isExposed>true</isExposed>
    <targets>
        <target>lightning__AppPage</target>
        <target>lightning__RecordPage</target>
        <target>lightning__HomePage</target>
    </targets>
</LightningComponentBundle>
```

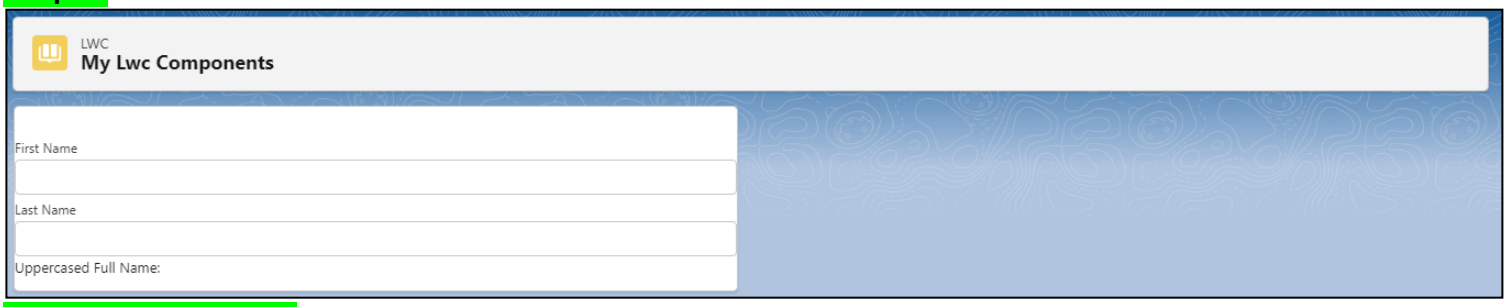
trackDecorator.html

```
<template>
  <lightning-card>
    <lightning-input
        name="firstName"
        label="First Name"
        onchange={handleChange}
    ></lightning-input>
    <lightning-input
        name="lastName"
        label="Last Name"
        onchange={handleChange}
    ></lightning-input>
    <p>
        Uppercased Full Name: {uppercasedFullName}
    </p>
  </lightning-card>
</template>
```

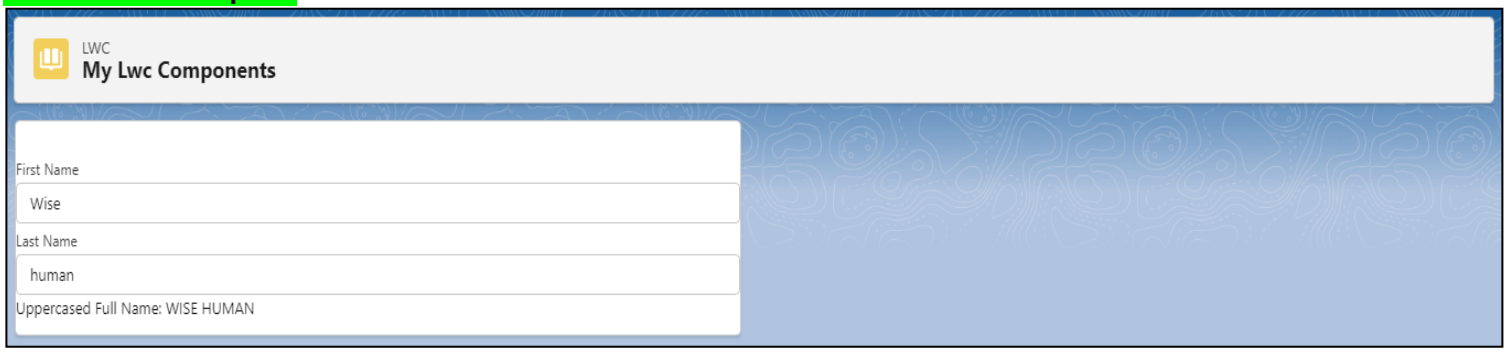
trackDecorator.js

```
import {LightningElement,track} from 'lwc';

export default class TrackDecorator extends LightningElement {
  firstName = '';
  lastName = '';

  handleChange(event) {
      const field = event.target.name;
      if (field === 'firstName') {
          this.firstName = event.target.value;
      } else if (field === 'lastName') {
          this.lastName = event.target.value;
      }
  }

  get uppercasedFullName() {
      return `${this.firstName} ${this.lastName}`.trim().toUpperCase();
  }
}
```

Output:



After filled the inputs:



Example 2: (with @track, when field is an array (list) or **object**)

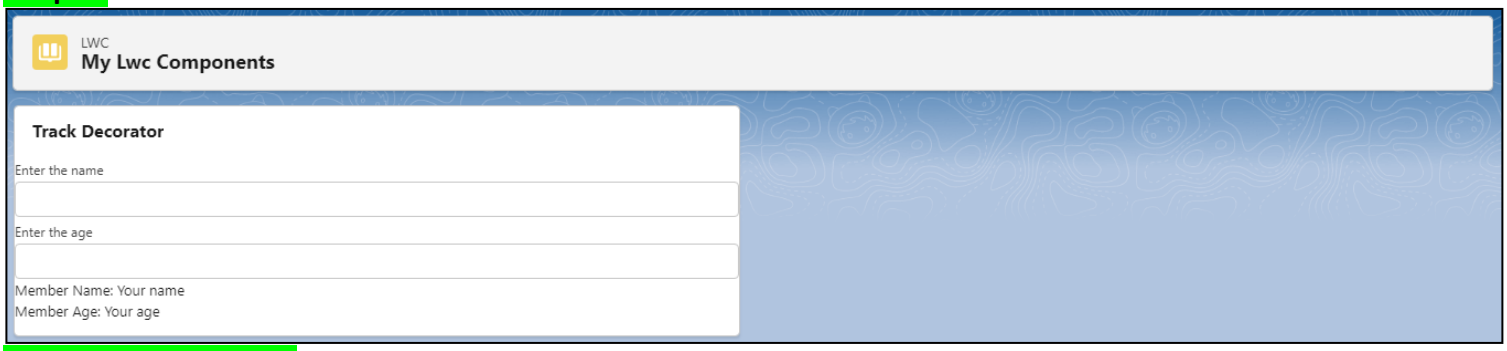
trackDecoratorr.html

```
<template>
  <lightning-card title="Track Decorator">
    <lightning-input
      type="text"
      label="Enter the name"
      onchange={handleMemberName}>
    </lightning-input>
    <lightning-input
      type="text"
      label="Enter the age"
      onchange={handleMemberAge}>
    </lightning-input>

    <h1>Member Name: {MemberInfo.name} </h1>
    <h1>Member Age: {MemberInfo.age} </h1>
  </lightning-card>

</template>
```

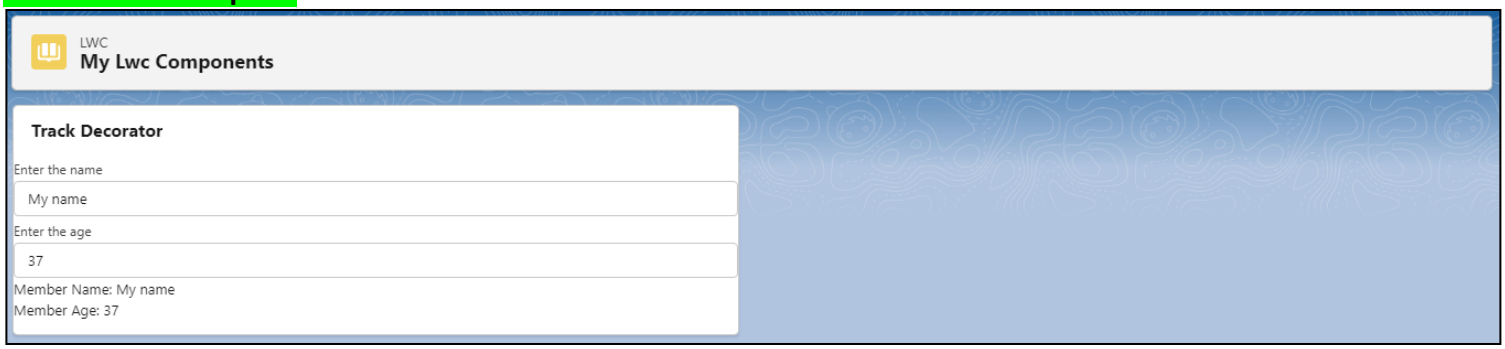
trackDecoratorr.js

```
import { LightningElement, track } from 'lwc';

export default class TrackDecoratorr extends LightningElement {
  @track
  MemberInfo={
    name:'Your name',
    age: 'Your age'
  }
  handleMemberName(event){
    this.MemberInfo.name=event.target.value;
  }
  handleMemberAge(event){
    this.MemberInfo.age=event.target.value;
  }
}
```

Output:



After filled the inputs:



8.3. @wire

- To read Salesforce data, Lightning web components use a reactive wire service. When the wire service provisions data, the component rerenders.
- Components use **@wire** in their JavaScript class to specify a wire adapter or an Apex method.
- Gives you an easy way to get and bind data from a Salesforce org.
- We need to import the **@salesforce/apex** scoped module into JavaScript controller class.

```
import apexMethodName from '@salesforce/apex/Namespace.Classname.apexMethodReference';
```

apexMethodName: An imported symbol that identifies the Apex method.
apexMethodReference: The name of the Apex method to import.
Classname: The name of the Apex class.
Namespace: The namespace of the Salesforce organization. Specify a namespace unless the organization uses the default namespace (c), in which case don't specify it.

Example 1:

MessageFromApex.apxc class

```
public with sharing class MessagefromApex {
  @AuraEnabled(cacheable=true)
  public static String getInputMessage() {
      return 'Message from Apex Class';
  }
}
```

wireDecorator.js-meta.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<LightningComponentBundle xmlns="http://soap.sforce.com/2006/04/metadata">
    <apiVersion>56.0</apiVersion>
    <isExposed>true</isExposed>
    <targets>
        <target>lightning__AppPage</target>
        <target>lightning__HomePage</target>
        <target>lightning__RecordPage</target>
    </targets>
</LightningComponentBundle>
```

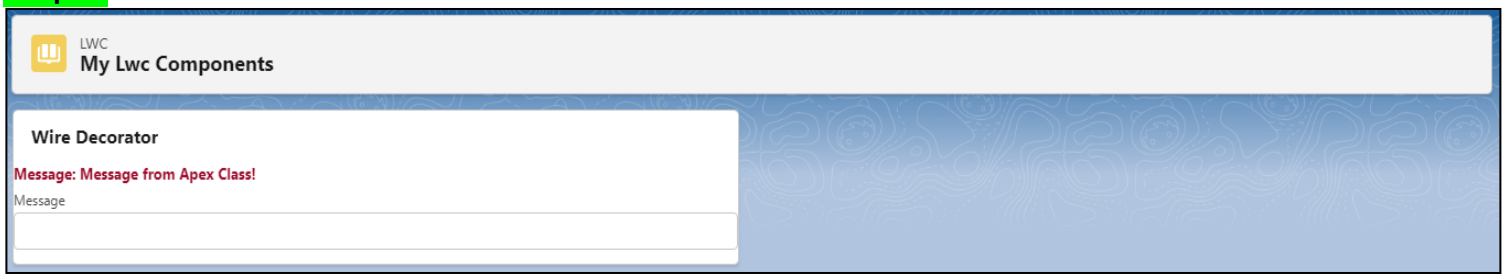
wireDecorator.html

```
<template>
  <lightning-card title="Wire Decorator">
      <div>
          <strong><p style="color:#9b0c2b;" >Message: {inputMessage.data}!</p></strong>
          <lightning-input  label="Message" value={inputMessage}
onchange={handleChange}></lightning-input>
      </div>
  </lightning-card>
</template>
```

wireDecorator.js

```
import { LightningElement, api, wire } from 'lwc';

import getInputMessage from '@salesforce/apex/MessagefromApex.getInputMessage';

export default class WireDecorator extends LightningElement {
   @api
   inputMessage = 'World';

    @wire
    (getInputMessage) inputMessage;

    handleChange(event) {
        this.inputMessage = event.target.value;
    }
}
```

Output:



Example 2:

wireDecorator2.html

```
<template>
  <lightning-card title="Account List From Apex Class" icon-name="custom:custom63">
    <div class="slds-p-around_medium" >
          <template if:true={accounts.data}>
              <template for:each={accounts.data} for:item="acc">
                  <p key={acc.Id}>{acc.Name}</p>
              </template>
        </template>
          <template if:true={accounts.error}>
              {accounts.error}
          </template>
      </div>
      </lightning-card>
</template>
```

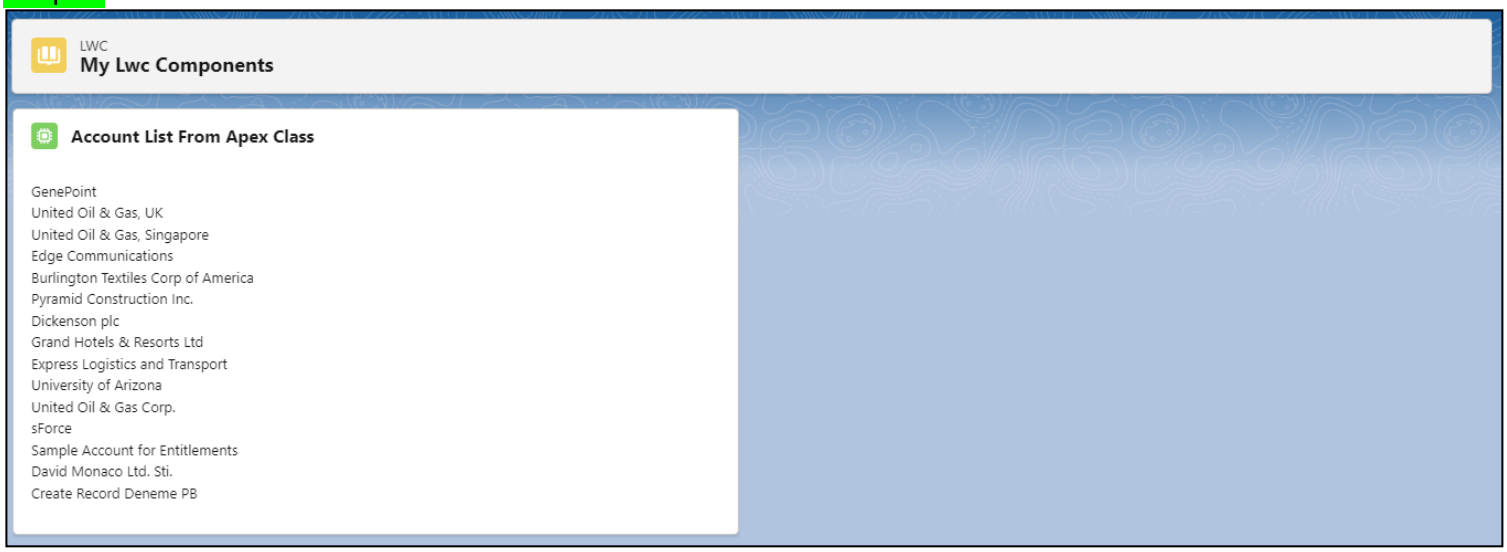
wireDecorator2.js

```
import { LightningElement, wire } from 'lwc';

import getAccountList from '@salesforce/apex/AccountSharingClass.getAccountList';

export default class WireDecorator2 extends LightningElement {

    @wire(getAccountList) accounts;
}
```

AccountSharing.apxc Class

```
public with sharing class AccountSharingClass {
  @AuraEnabled(cacheable=true)
  public static List<Account> getAccountList() {
    return [SELECT Id, Name, Type, Rating, Phone FROM Account];
  }
}
```

Output:



9. Component Composition In LWC

Before we compose more complex components and apps, it's important to understand how Lightning web components use shadow DOM to render elements.

9.1. Shadow DOM

Shadow DOM is a web standard that encapsulates the elements of a component to keep styling and behavior consistent in any context. Since not all browsers implement shadow DOM, Lightning Web Components uses a shadow DOM polyfill (@lwc/synthetic-shadow). A polyfill is code that allows a feature to work in a web browser.

Example:

```
<!-- recipe-hello -->
<template>
  <ui-card title="Hello">
      <div>Some content in child component</div>
  </ui-card>
</template>
```

In this example, This **`recipe-hello`** component contains a **`ui-card`** component with some markup.

When the components render in a browser, the elements are encapsulated in a shadow tree. A shadow tree is part of the DOM that's hidden from the document that contains it. The shadow tree affects how you work with the DOM, CSS, and events.

```
<recipe-hello>
  #shadow-root
  |    <ui-card>
  |        #shadow-root
  |        |    <div>Some content in child component</div>
  |     </ui-card>
</recipe-hello>
```

The *shadow root* (document fragment.) defines the boundary between the DOM and the shadow tree. This boundary is called the *shadow boundary (line between the shadow root and the host element)*.

9.2. Composing Components

Component Composition in LWC refers to adding of a component inside the body of another component. Composition permits you to build complex components from simple building-block components.

In other words; Component composition is concept to create a small reusable component separately and use in the other component.

There are many **advantages** of component composition:

- Composing, applications, and components from a collection of smaller components make code reusable and maintainable. No need to re-write the code again.
- Easy to call other components.
- Reduce code size and improves code readability.

To understand how we can create a composition in lightning web component, we create the main component (parent component) and **call other reusable child components inside the main component**.

parentComponent.js-meta.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<LightningComponentBundle xmlns="http://soap.sforce.com/2006/04/metadata">
    <apiVersion>56.0</apiVersion>
    <isExposed>true</isExposed>

    <targets>
        <target>lightning__AppPage</target>
        <target>lightning__RecordPage</target>
        <target>lightning__HomePage</target>
    </targets>
</LightningComponentBundle>
```

parentComponent.html

```
<template>
    <lightning-card>
        <h1>This is inside Parent Component</h1>
        <hr/>
        <!--child compoenent-->
            <c-child-component></c-child-component>
    </lightning-card>
</template>
```

When we call child component inside the parent component we use **kabab case** to represent the name of it. we called the child component as **<c-child-component>** in the above parent syntax. **<c-** this is a default namespace.

In other words: Property names in JavaScript are in camel case while HTML attribute names are in kebab case (dashseparated) to match HTML standards.

parentComponent.js

```
import { LightningElement } from 'lwc';

export default class ParentComponent extends LightningElement {
}
```

childComponent.js-meta.xml

```
<LightningComponentBundle xmlns="http://soap.sforce.com/2006/04/metadata">
    <apiVersion>56.0</apiVersion>
    <isExposed>false</isExposed>
</LightningComponentBundle>
```

To allow the component to be used in Lightning App Builder or Experience Builder, we set isExposed to **true** and we **define at least one <target>**, which is a type of Lightning page. But in child component, when setting isExposed to false, we can use it. And it can appear inside parent component.

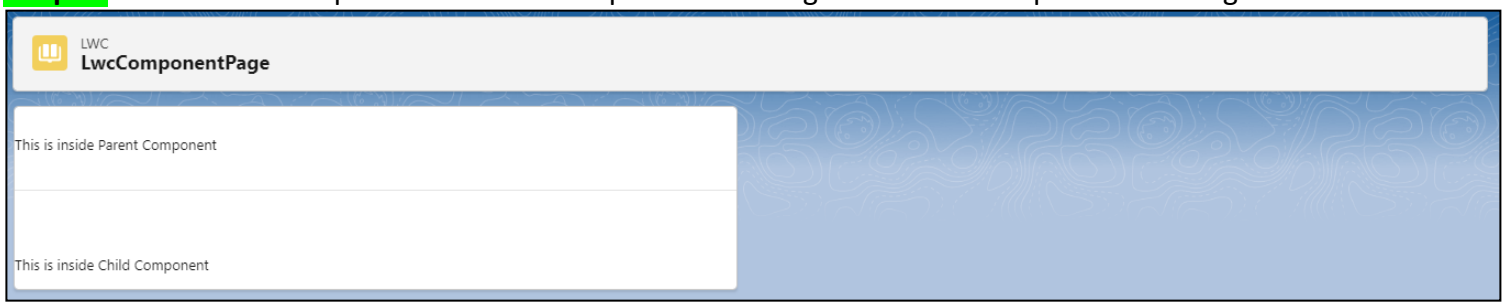
childComponent.html

```
<template>
    <lightning-card>
        <p>This is inside Child Component</p>
    </lightning-card>
</template>
```

childComponent.js

```
import { LightningElement } from 'lwc';

export default class ChildComponent extends LightningElement {
}
```

Output: we can see both parent and child component data together as we composed them together.



Container

A container contains other components but itself is contained within the owner component. In this example**, `c-child-component`** is a container. A container is less powerful than the owner.

A container can:

- Read, but not change, public properties in contained components
- Call methods on composed components
- Listen for some, but not necessarily all, events bubbled up by components that it contains.

Parent and child

When a component contains another component, which, in turn, can contain other components, we have a containment hierarchy. In the documentation, we sometimes talk about parent and child components. A parent component contains a child component. A parent component can be the owner or a container. In previous example we learnt how we can include the reusable components in a single (main) component.

Now in this example, we will learn how to pass the value from parent component to the child component by using **`@api`**.

When a component decorates a field with **@api** to expose it as public property, the value should be set only when the field is initialized. Only the owner component should set the value after the field has been initialized. Values passed to a component should be treated as read-only.

parentComponent.html

```
<template>
  <lightning-card>
    Look at the below!! This is a name from the parent component :
    <div>
     <c-child-2component name={myname}></c-child-2component>
    </div>
  </lightning-card>
</template>
```

parentComponent.js

```
import { LightningElement } from 'lwc';

export default class Parent2Component extends LightningElement {
  myname = 'Wise Quarter';
}
```

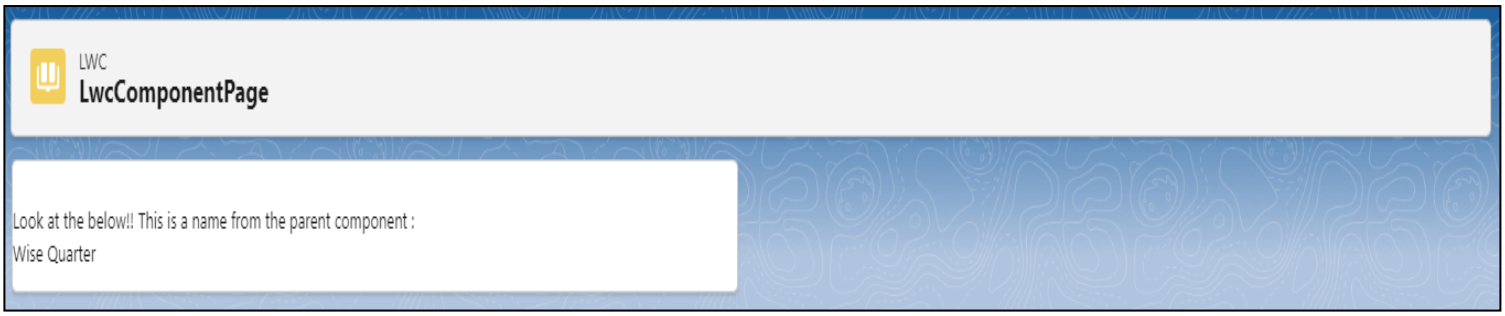
childComponent.html

```
<template>
  {name}
</template>
```

childComponent.js

```
import { LightningElement, api } from 'lwc';

export default class Child2Component extends LightningElement {
   @api name;
}
```

Output:



Example: we will create 3 components; 1. mySchool (Parent), 2. Managers (child), 3. Instructors(child).

mySchool.js-meta.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<LightningComponentBundle xmlns="http://soap.sforce.com/2006/04/metadata">
    <apiVersion>56.0</apiVersion>
    <isExposed>true</isExposed>
    <targets>
        <target>lightning__AppPage</target>
        <target>lightning__RecordPage</target>
        <target>lightning__HomePage</target>
    </targets>
</LightningComponentBundle>
```

mySchool.html

```
<template>
  <lightning-card title="My School" icon-name="custom:custom51">
      <div class="slds-var-p-around_x-large">
          <p class="head">School Staff</p><br/>
          <p><b>Manager List</b></p>
          <c-managers managers={managerlist}></c-managers>
          <p><b>Instructor List</b></p>
          <c-instructors instructors={instructorlist}></c-instructors>
      </div>
  </lightning-card>
</template>
```

mySchool.js

```
import { LightningElement } from 'lwc';

export default class MySchool extends LightningElement {
  managerlist = ['Manager 1 ', 'Manager2 ', 'Manager 3 '];
  instructorlist = ['Instructor 1', 'Instructor 2', 'Instructor 3'];
}
```

mySchool.css

```
.head{
  font-weight: bold;
  padding: 15px;
  background-color : #5B318C;
  color: white;
  margin-top: 5px;
  border-radius:  2rem;
}
```

managers.html

```
<template>
  <p style="color:#FC620A;">{managers}</p>
</template>
```

managers.js

```
import { LightningElement, api } from 'lwc';

export default class Managers extends LightningElement {
  @api managers = [];
}
```

instructor.html

```
<template>
  <p style="color: green;">{instructor}</p>
</template>
```

instructor.js

```
import { LightningElement, api } from 'lwc';

export default class Instructors extends LightningElement {
  @api instructors = [];
}
```

Output:

The component that owns the template(s) is known as the owner. In this case, the owner is the 'mySchool' component. All of the assembled components are under the control of the owner.
The owner component can:

- set public properties on composed components
- call methods on composed components
- listen for any events fired by the composed components.

Example: Student App component (parent-container-child)

student.js-meta.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<LightningComponentBundle xmlns="http://soap.sforce.com/2006/04/metadata">
    <apiVersion>56.0</apiVersion>
    <isExposed>true</isExposed>
    <targets>
        <target>lightning__AppPage</target>
        <target>lightning__HomePage</target>
        <target>lightning__RecordPage</target>
    </targets>
</LightningComponentBundle>
```

student.html

```
<template>
  <template if:true={display}>
      <lightning-card title="Student App">
            <lightning-input
                  type="text"
                  label="Student Number"
                  placeholder="Enter the student's number..."
                  onblur={handleStudentNumber}
            ></lightning-input>
            <lightning-input
                  type="text"
                  label="Student First Name"
                  placeholder="Enter the student's first name..."
                  onblur={handleFirstName}
            ></lightning-input>
            <lightning-input
                  type="text"
                  label="Student Last Name"
                  placeholder="Enter the student's last name..."
                  onblur={handleLastName}
            ></lightning-input>
    </lightning-card>
   <lightning-button label="Search" onclick={searchHandler}></lightning-button>
  </template>

  <template if:false={display}>
    <lightning-card title='Student Detail'>
      <p>Student Number:<strong>{studentNumber}</strong></p>
      <c-student-detail first-name={StudentFirstName} last-name={StudentLastName}>
</c-student-detail>
<!-- to assign java script property
from parent component we create
attribute in html element and we assign
the value -->
    </lightning-card>
  </template>
</template>
```

student.js

```
import { LightningElement } from 'lwc';
export default class student extends
LightningElement {
  display=true;
  studentNumber='';
  StudentFirstName='';
  StudentLastName='';

  handleStudentNumber(event){
    this.studentNumber=event.target.value;
  }
  handleFirstName(event){
    this.StudentFirstName=event.target.value;
  }
  handleLastName(event){
    this.StudentLastName=event.target.value;
  }
  searchHandler(){
    this.display=false;
  }
}
```

studentDetail.html

```
<template>

  <p>Student First Name:<strong>{firstName}</strong></p>
  <p>Student Last Name:<strong>{lastName}</strong></p>

  <c-course phone="123456789" course="Salesforce Developer" ></c-course>

 </template>
```

studentDetail.js

```
import { LightningElement, api } from 'lwc';

export default class StudentDetail extends LightningElement {
  @api
  firstName='';
  @api
  lastName='';
}
```

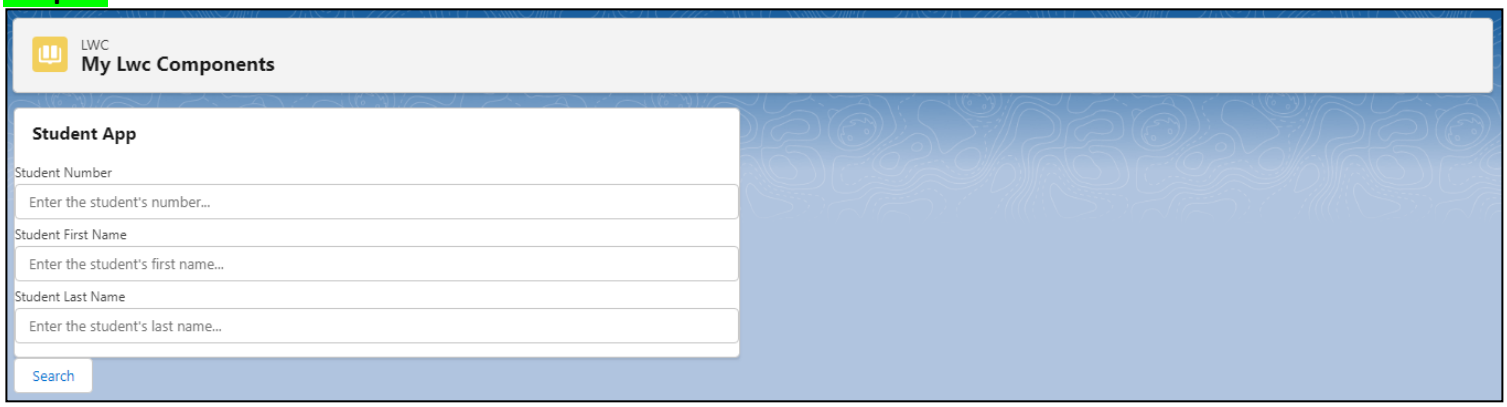
course.html

```
<template>
  <strong><p style="background-color: chartreuse;"> Phone: {phone}</p></strong>
  <strong><p style="color:blueviolet">Courses Taken: {course}</p></strong>
 </template>
```

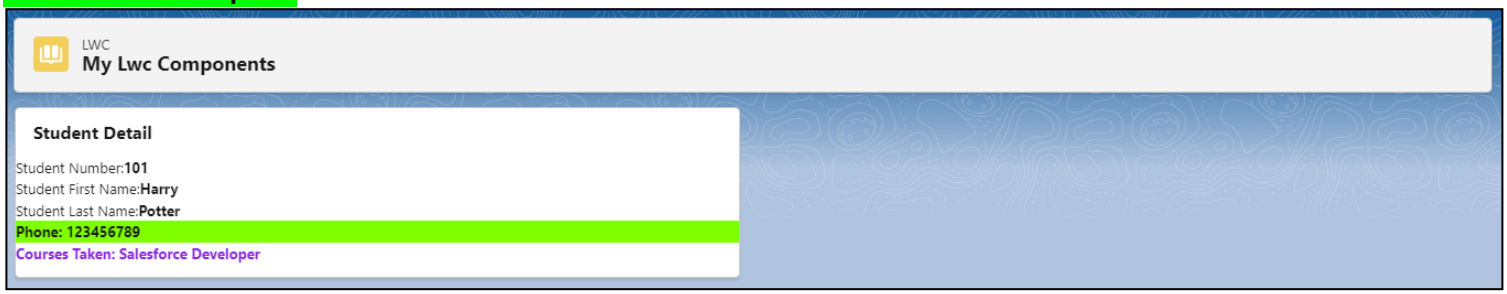
course.js

```
import { LightningElement, api } from 'lwc';

export default class Course extends LightningElement {
  @api
  phone='';
  @api
  course='';
}
```

Output:



After filled the inputs:

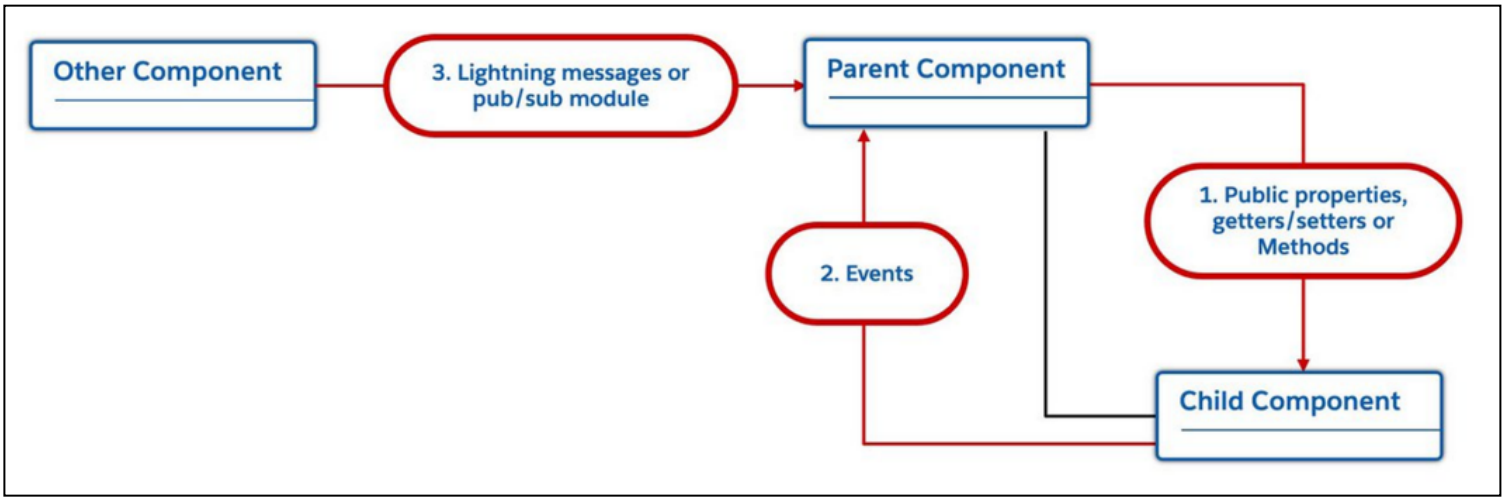


- we created 3 components which are student, studentDetail, and course.
- student is the parent component that includes the other two component,
- studentDetail is a container which contains course.
- In our example, we exposed the parent component (student) and called the children's info to reflect the user interface.
- Using **@api** decorator, we made child properties public, so we could manipulate child properties in parent components.

10. Events in LWC

- Events are used in LWC for components communication. There are typically 3 approaches for communication between the components using events.

Component Hierarchy



https://developer.salesforce.com/blogs/2021/05/inter-component-communication-patterns-for-lightning-web-components?_ga=2.19151480.1115298334.1681331614-1480446641.1673265161&_gac=1.53362010.1680637924.Cj0KCQjwla-hBhD7ARIsAM9tQKuNfwZ2pHCG0hz8wlFJRjAKYGtrQH85RmE-fWX_6NUwb2CW73w4kQMaAhyYEALw_wcB

1. Standard Event communication in Lightning web component (Parent to Child).
   (Passing data *down* the component hierarchy.)
2. Custom Event Communication in Lightning Web Component (Child to Parent).
   (Passing data *up* the component hierarchy.)
3. Publish Subscriber model in Lightning Web Component (*Two components which doesn't have a direct relation*)
   (Passing data to components that share no common ancestry.)

10.1. Standard Event communication in Lightning web component (Parent to Child)

- There are three techniques for passing data from a parent to a child component; you may either use a **public properties** or **public methods**.
- For reference, a child component is a component that's included in the HTML template of a parent component at design time. At runtime, the child component will be embedded in a sub-DOM tree of the parent component.

10.1.1. Public Properties

A public property is the simplest way to receive data in a child component and requires the least amount of code. A public property is exposed by applying the **`@api`** decorator on a class property of a child component. This lets the parent component assign a value to it in two different ways. Either via a custom HTML attribute or dynamically with JavaScript.

Public properties are reactive so the component re-renders whenever one of its public properties is updated. On the downside, you cannot run custom logic when the value changes and the child component should not assign values to its public properties.

Example: (@api decorator used to define public property, which is reactive.)

parentComponent.js-meta.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<LightningComponentBundle xmlns="http://soap.sforce.com/2006/04/metadata">
    <apiVersion>56.0</apiVersion>
    <isExposed>true</isExposed>
    <targets>
        <target>lightning__AppPage</target>
        <target>lightning__RecordPage</target>
        <target>lightning__HomePage</target>
    </targets>
</LightningComponentBundle>
```

parentComponent.html

```
<template>
    <lightning-card>
        <strong><h1>Parent Component</h1></strong>
        <h2>This is inside Parent Component</h2>
        <hr/>
        <!--child compoenent-->
            <c-child-component get-value-from-parent={value}></c-child-component>
    </lightning-card>
</template>
```

parentComponent.js

```
import { LightningElement } from 'lwc';

export default class ParentComponent extends LightningElement {
  value = "This value is from parent component";
}
```

childComponent.js-meta.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<LightningComponentBundle xmlns="http://soap.sforce.com/2006/04/metadata">
    <apiVersion>56.0</apiVersion>
    <isExposed>false</isExposed>
</LightningComponentBundle>
```

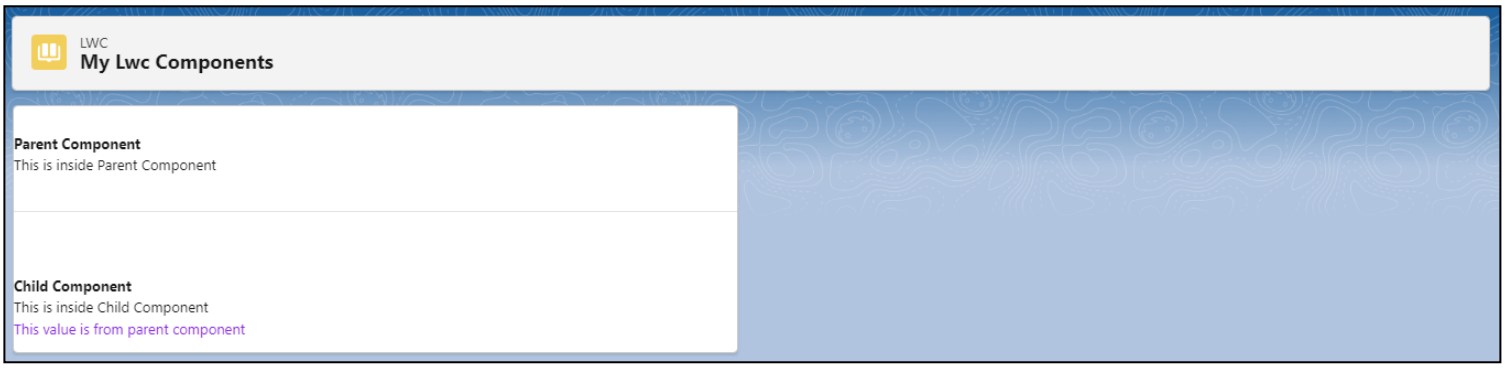
childComponent.html

```
<template>
    <lightning-card>
        <strong><h1>Child Component</h1></strong>
        <p>This is inside Child Component</p>
        <p style="color: blueviolet;" >{getValueFromParent}</p>
    </lightning-card>
</template>
```

childComponent.js

```
import { LightningElement, api } from 'lwc';

export default class ChildComponent extends LightningElement {
  @api getValueFromParent;
}
```

Output:



In this example;

- we defined a variable **getValueFromParent** using the **@api** decorator in child component and used it in the HTML template to display its value in the child component.
- we created a **value** variable and assigned a string literal to it.
- To pass the value to the child component, we added the child component's tag to the parent HTML template and assigned the value to the child's property as an attribute.
- Note how the **getValueFromParent** property name is automatically converted to kebab case (lowercase, dash-separated) when set with an HTML attribute.

10.1.2. Public Methods

We can use the **@api decorator to make the method public available so parent** can be able to call it directly using JavaScript API.

You may call public methods to pass several values to a child component in order to perform an action. As opposed to setters, methods allow you to enforce consistency by passing multiple parameters at once.

Example:

parentComponent.js-meta.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<LightningComponentBundle xmlns="http://soap.sforce.com/2006/04/metadata">
    <apiVersion>56.0</apiVersion>
    <isExposed>true</isExposed>
    <targets>
        <target>lightning__AppPage</target>
        <target>lightning__RecordPage</target>
        <target>lightning__HomePage</target>
    </targets>
</LightningComponentBundle>
```

parentComponent.html

```
<template>
  <lightning-card title="Parent to Child Component">
     <lightning-input
     label="Enter the Message"
     onchange={handleChangeEvent}>
    </lightning-input>
     <br>
    <c-child-component></c-child-component>
  </lightning-card>
</template>
```

parentComponent.js

```
import { LightningElement } from 'lwc';

export default class ParentComponent extends LightningElement {
  handleChangeEvent(event){
    this.template.querySelector('c-child-component').changeMessage(event.target.value);
  }
}
```

childComponent.js-meta.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<LightningComponentBundle xmlns="http://soap.sforce.com/2006/04/metadata">
    <apiVersion>56.0</apiVersion>
    <isExposed>false</isExposed>
</LightningComponentBundle>
```

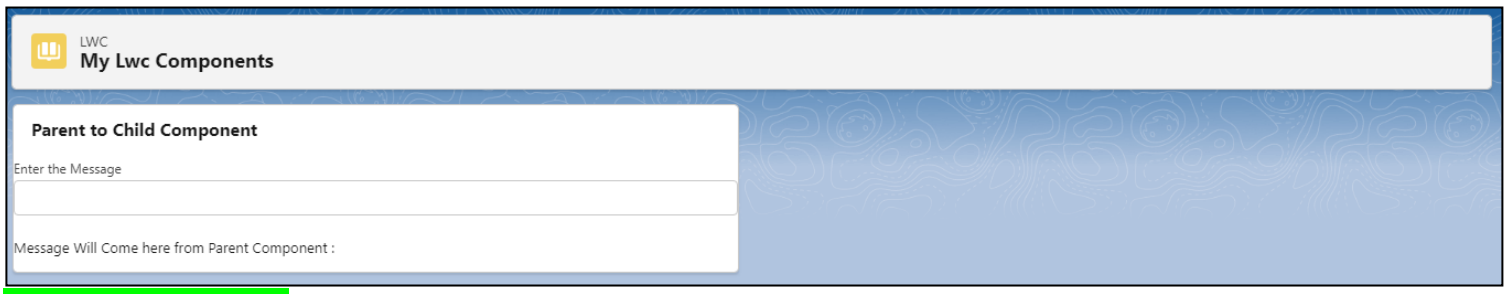
childComponent.html

```
<template>
  Message Will Come here from Parent Component : {Message}
</template>
```

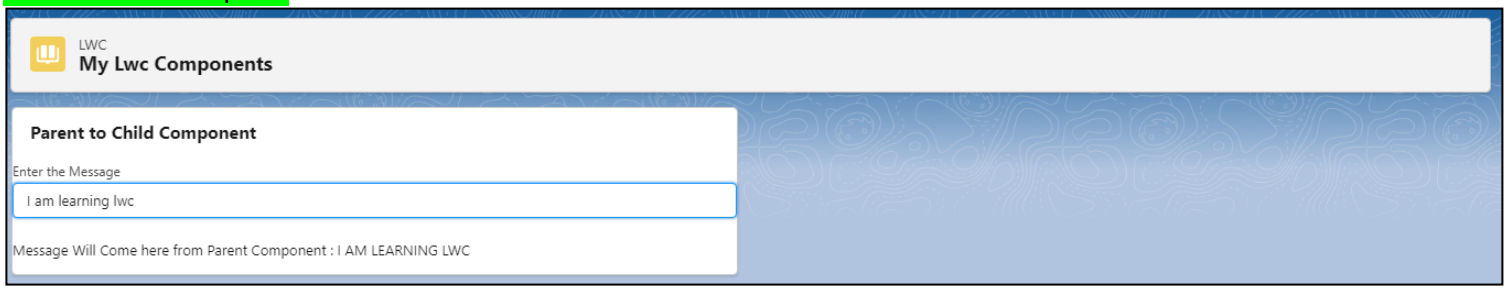
childComponent.js

```
import { LightningElement, track, api  } from 'lwc';

export default class ChildComponent extends LightningElement {
  @track Message;
  @api
  changeMessage(strString) {
       this.Message = strString.toUpperCase();
  }
}
```

Output:



After filled the inputs:



10.2. Custom Event Communication in Lightning Web Component (Child to Parent)

As we have seen, passing a public property from a parent and receiving it in the child is the easiest way to achieve Parent to Child communication. In the case of Child to Parent communication, it is a bit more complicated. From the child component, we will pass the value to the parent component using **CustomEvent**.

Custom Event is used to make the communication from Child Component to Parent Component. With LWC we can create and dispatch the custom event.

You can pass data from a child component to a parent or ancestor with events. Lightning Web Components relies on DOM events (as defined in the standard DOM specification) to propagate data up the component hierarchy.

Note: The CustomEvent constructor has one required parameter: a string, which refers to the event type.

Example: (Declarative via html markup)

when the user enters something in the lightning input field, the component creates and dispatches the CustomEvent called **getsearchvalue** event. The event includes the data in **detail** property.

parentComponent4.js-meta.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<LightningComponentBundle xmlns="http://soap.sforce.com/2006/04/metadata">
    <apiVersion>56.0</apiVersion>
    <isExposed>true</isExposed>
    <targets>
        <target>lightning__AppPage</target>
        <target>lightning__RecordPage</target>
        <target>lightning__HomePage</target>
    </targets>
</LightningComponentBundle>
```

parentComponent4.html

```
<template>
  <lightning-card>

   <strong><h1 style="text-align: center;">Parent Component</h1></strong>

    <c-child-component4 ongetsearchvalue={handleSearchValue}></c-child-component4>

    <p>{searchValue}</p>
  </lightning-card>
</template>
```

parentComponent4.js

```
import { LightningElement, track } from 'lwc';

export default class ParentComponent4 extends LightningElement {
  @track searchValue;
  handleSearchValue(event){
    this.searchValue= event.detail;
  }
}
```

The parent component listens for the getsearchvalue event in the **ongetsearchvalue** attribute and handles it in the **handleSearchValue** event handler. In handleSearchValue, we are assigning the value from the event. detail to searchValue variable.

In this way, we can perform Parent-Child communication effectively.

childComponent4.js-meta.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<LightningComponentBundle xmlns="http://soap.sforce.com/2006/04/metadata">
    <apiVersion>56.0</apiVersion>
    <isExposed>false</isExposed>
</LightningComponentBundle>
```

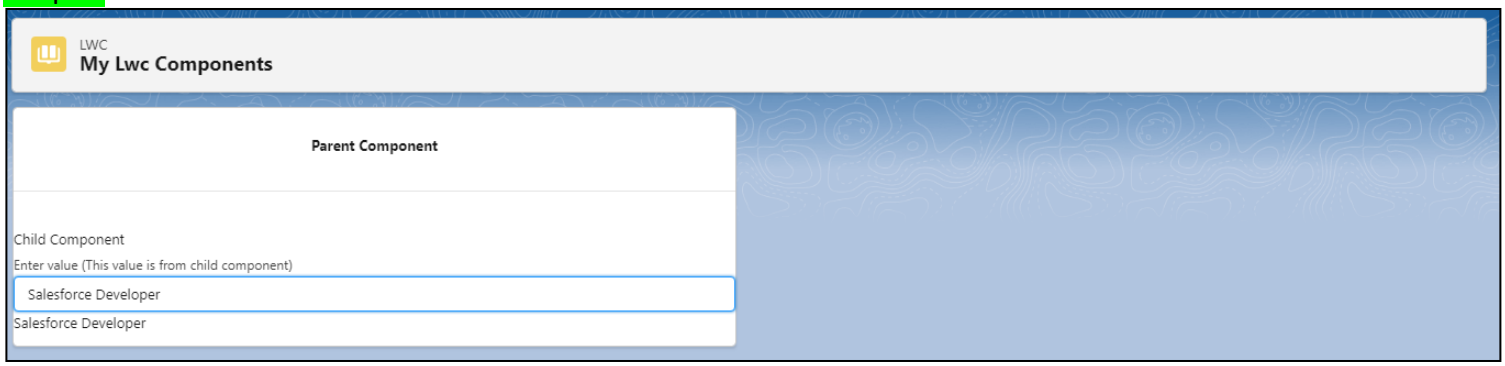
childComponent4.html

```
<template>
  <h2>Child Component</h2>
  <lightning-input label="Enter value (This value is from child component)"
onchange={handleChange} ></lightning-input>
</template>
```

childComponent4.js

```
import { LightningElement } from 'lwc';

export default class ChildComponent4 extends LightningElement {
  searchKey;
  handleChange(event){
    this.searchKey=event.target.value;

    //Create Event
    const searchEvent= new CustomEvent("getsearchvalue", {detail: this.searchKey});

    // Dispatches the event
    this.dispatchEvent(searchEvent);
  }

}
```

Output:



Example: (Declarative via html markup)

childComponent5.js-meta.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<LightningComponentBundle xmlns="http://soap.sforce.com/2006/04/metadata">
    <apiVersion>56.0</apiVersion>
    <isExposed>false</isExposed>
</LightningComponentBundle>
```

childComponent5.html

```
<template>
  <lightning-card title="Child Component">
      <div class="slds-m-around_medium">
          <lightning-input name="textVal" label="Enter Text" onchange={handleChange}>
</lightning-input>
      </div>
  </lightning-card>
</template>
```

***We created child html file to get value from user.

childComponent5.js

```
import { LightningElement } from 'lwc';

export default class ChildComponent5 extends LightningElement {
  handleChange(event) {
    event.preventDefault();
    const name = event.target.value;
    const selectEvent = new CustomEvent('mycustomevent', {
        detail: name
    });
   this.dispatchEvent(selectEvent);
  }
}
```

***we updated Child Comp javaScript file to raise a CustomEvent with text value.

parentComponent5.js-meta.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<LightningComponentBundle xmlns="http://soap.sforce.com/2006/04/metadata">
    <apiVersion>56.0</apiVersion>
    <isExposed>true</isExposed>
    <targets>
        <target>lightning__AppPage</target>
        <target>lightning__RecordPage</target>
        <target>lightning__HomePage</target>
    </targets>
</LightningComponentBundle>
```

parentComponent5.html

```
<template>
  <lightning-card title="Parent Component">
    <div>
      <h1>Value From Child : <strong style="color: brown;">{msg}</strong> </h1>
      <c-child-component5 onmycustomevent={handleCustomEvent}></c-child-component5>
    </div>
  </lightning-card>
</template>
```

***we created Parent component where we will handle the event.

We need to add prefix as “on” before the custom event name and in parent component we need to invoke the event listener as handleCustomEvent using onmycustomevent attribute. It is recommend to conform with the DOM event standard.

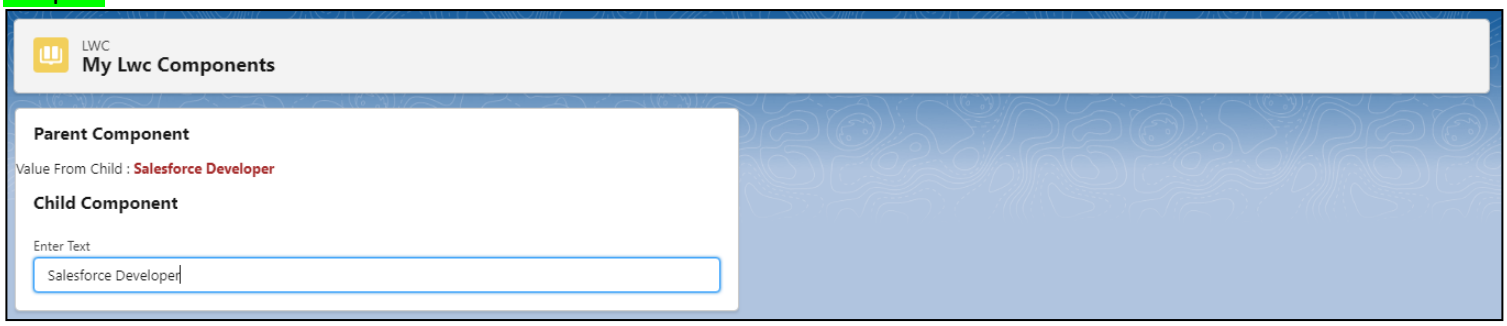
- No uppercase letters
- No spaces
- Use underscores to separate words

parentComponent5.js

```
import { LightningElement, track } from 'lwc';

export default class ParentComponent5 extends LightningElement {
  @track msg;
  handleCustomEvent(event) {
      const textVal = event.detail;
      this.msg = textVal;
  }
}
```

*** Finally, we updated parent component javaScript file and added handleCustomEvent method

Output:



Example: (with bubbles)
previous example using "**JavaScript using addEventListener method | Attaching event Listener Programmatically"**

childComponent6.js-meta.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<LightningComponentBundle xmlns="http://soap.sforce.com/2006/04/metadata">
    <apiVersion>56.0</apiVersion>
    <isExposed>false</isExposed>
</LightningComponentBundle>
```

childComponent6.html

```
<template>
  <lightning-card title="Child Component">
      <div class="slds-m-around_medium">
          <lightning-input name="textVal" label="Enter Text"
onchange={handleChange}></lightning-input>
      </div>
  </lightning-card>
</template>
```

childComponent6.js

```
import { LightningElement } from 'lwc';

export default class ChildComponent6 extends LightningElement {
  handleChange(event) {
    event.preventDefault();
    const name = event.target.value;
    const selectEvent = new CustomEvent('mycustomevent', {
        detail: name ,bubbles: true
    });
  this.dispatchEvent(selectEvent);
  }
}
```

***

- When an event is fired the event is propagated up to the DOM. Event propagation typically involves two phases event bubbling and event capturing. The most commonly used event propagation phase for handling events is **event bubbling**. In this case the event is triggered at the child level and propagates up to the DOM. Where as event capturing phases moves from top to bottom of the DOM. This phase is rarely used for event handling.
- You can also stop the event propagation at any time with `Event.stopPropagation()`.
- In LWC we have two flags which determines the behavior of event in event bubbling phase.
  1. **Bubbles:** A Boolean value indicating whether the event bubbles up through the DOM or not. Defaults to false.
  2. **Composed:** A Boolean value indicating whether the event can pass through the shadow boundary. Defaults to false.

parentComponent6.js-meta.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<LightningComponentBundle xmlns="http://soap.sforce.com/2006/04/metadata">
    <apiVersion>56.0</apiVersion>
    <isExposed>true</isExposed>
    <targets>
        <target>lightning__AppPage</target>
        <target>lightning__RecordPage</target>
        <target>lightning__HomePage</target>
    </targets>
</LightningComponentBundle>
```

parentComponent6.html

```
<template>
  <lightning-card title="Parent Component">
    <div>
      <h1>Value From Child : <strong style="color: brown;">{msg}</strong> </h1>
      <c-child-component6></c-child-component6>
    </div>
  </lightning-card>
</template>
```

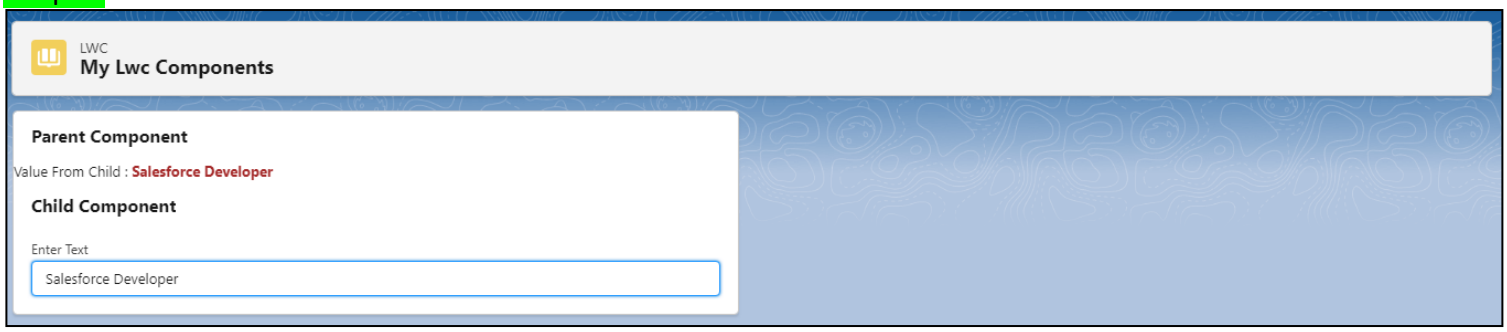
*** We removed **onmycustomevent** attribute from child component tag.

parentComponent6.js

```
import { LightningElement, track } from 'lwc';

export default class ParentComponent6 extends LightningElement {
  @track msg;

  constructor() {
      super();
      this.template.addEventListener('mycustomevent', this.handleCustomEvent.bind(this));
  }

  handleCustomEvent(event) {
      const textVal = event.detail;
      this.msg = textVal;
  }
}
```

***We changed parent component JavaScript like above

Output:



Example:

In this example, the child component will be sending a message to the parent component.

childComponent6.js-meta.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<LightningComponentBundle xmlns="http://soap.sforce.com/2006/04/metadata">
    <apiVersion>56.0</apiVersion>
    <isExposed>false</isExposed>
</LightningComponentBundle>
```

childComponent6.html

```
<template>
  <lightning-card title="Child Component">
      <lightning-button variant="brand" label="Send Message" title="Send Message" slot =
"actions" onclick={childHandler}></lightning-button>
  </lightning-card>
</template>
```

childComponent6.js

```
import { LightningElement } from 'lwc';

export default class ChildComponent7 extends LightningElement {
      childHandler() {
        const evt = new CustomEvent('sendmessage', {detail: "This message is sent from child
Component"});
        this.dispatchEvent(evt);
    }
}
```

in the above child Javascript code, we created a custom event by providing the name along with passing data in the event.

```
const evt = new CustomEvent('sendmessage', {detail: "This message is sent from child
Component"});
```

then we dispatched the event

```
this.dispatchEvent(evt);
```

parentComponent6.js-meta.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<LightningComponentBundle xmlns="http://soap.sforce.com/2006/04/metadata">
    <apiVersion>56.0</apiVersion>
    <isExposed>true</isExposed>
    <targets>
        <target>lightning__AppPage</target>
        <target>lightning__RecordPage</target>
        <target>lightning__HomePage</target>
    </targets>
</LightningComponentBundle>
```

parentComponent6.html

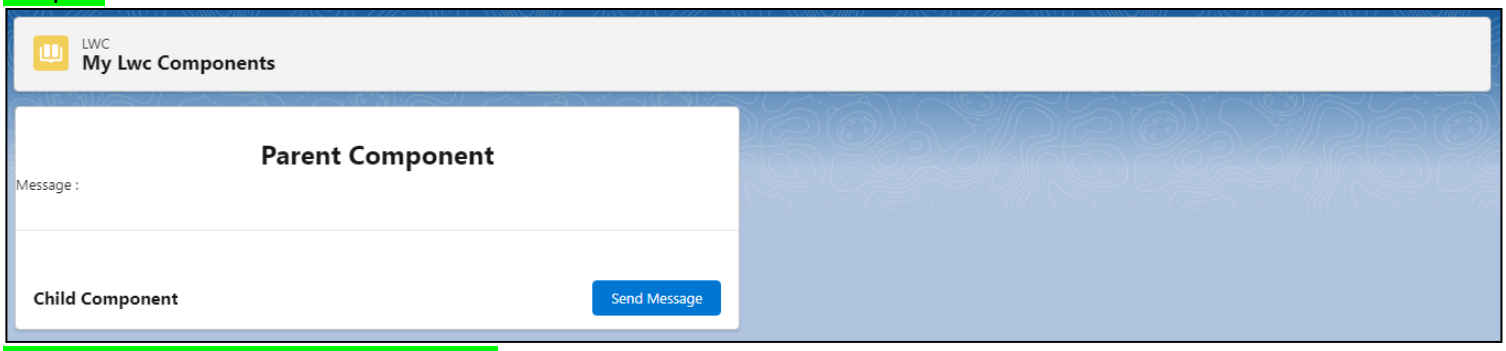
```
<template>
  <lightning-card>
    <strong><h1 style="text-align: center; font-size: x-large">Parent
Component</h1></strong>
      <p>Message : {message}</p>
      <hr>
      <c-child-component7 onsendmessage={parentHandler}></c-child-component7>
  </lightning-card>
</template>
```

we have added the handler – **onsendmessage={parentHandler}**. Whenever the child component dispatches the custom event **sendmessage**, the parent would be handling it.

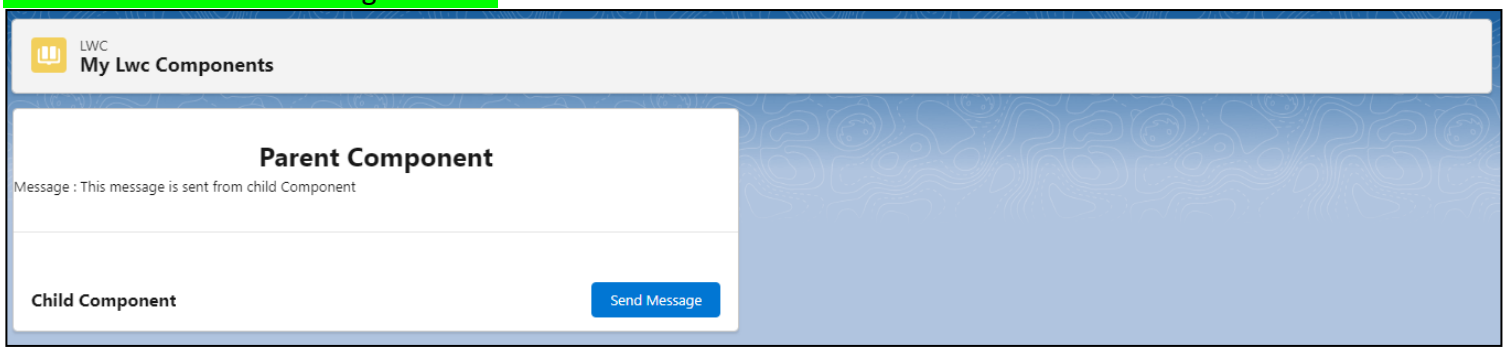
parentComponent6.js

```
import { LightningElement } from 'lwc';

export default class ParentComponent7 extends LightningElement {
  message;

  parentHandler(event) {
      this.message = event.detail;
  }
}
```

Output:



After clicked on SendMessage Button:



10.3. Publish Subscriber model in Lightning Web Component (*Two components which doesn't have a direct relation*)

Communication between components which aren't in the same DOM tree (unrelated components) can be achieved. This type of communication can be achieved with the **Lightning Message Service** or with the **publish-subscribe model**. Essentially, one component subscribes to an event, whereas another component publishes the event and handles it in the same scope.

As of the Summer '20 major release, the Lightning message service (LMS) is now Generally Available (GA). LMS is a client-side cross-DOM pub/sub eventing mechanism built and supported by Salesforce. It supersedes the open-source non-supported c/pubsub component that was made available early in the release of Lightning web components as a temporary hold over for the lack of features analogous to <aura:event> at the time.

With LMS being now Generally Available, pubsub model has been retired all use in any sample apps and removed its core code as well.

Here's a recap of the different features that you can use to share data across components:

Comm. type	Feature	Pros	Cons	Use Case
Parent to child	**Public property**	• Simplest option (smallest amount of code) • Exposed to parent in HTML and JS • Reactive	• No custom logic on update • Value should not be set by child component	Preferred solution for passing data to a child component if you don't need to: • run custom logic on update • set a value in the child component
	Public getter/setter	• Allows custom logic on get/set • Exposed to parent in HTML and JS • Reactive • Setter can be called by child component	Persistence not built-in	• Pass data to a child component and execute custom logic • Expose data to a parent and update it in a child component
	Public method	• Accepts multiple parameters (enforces consistency) • Supports asynchronous processing	• Persistence not built-in • Not exposed in parent's HTML	Trigger an action on a child component
Child to parent	**Event**	Standard DOM event behavior		Pass data up to a parent or an ancestor component
Across components that share no common ancestry	**Lightning message service (LMS)**	Supports LWC, Aura and Visualforce	• Requires metadata • Some containers are not supported (see documentation)	Preferred solution for exchanging data across components that share no common ancestry
	pubsub module	• No metadata dependency • Runs everywhere	• Not officially supported or maintained • pubsub code needs to be duplicated in your project	Fallback solution for containers that do not support LMS

https://developer.salesforce.com/blogs/2021/05/inter-component-communication-patterns-for-lightning-web-components?_ga=2.19151480.1115298334.1681331614-1480446641.1673265161&_gac=1.53362010.1680637924.Cj0KCQjwIa-hBhD7ARIsAM9tQKuNfwZ2pHCG0hz8wlFJRjAKYGtrQH85RmE-fWX_6NUwb2CW73w4kQMaAhyYEALw_wcB

10.3.1. Lightning message service

The Lightning Message Service (LMS) is the preferred solution for communicating between components that aren't in the same DOM tree. LMS also allows communication between the three Salesforce UI technologies: Lightning Web Components, Aura and Visualforce.

Lightning Message Service doesn't work with Salesforce Tabs + Visualforce sites or with Visualforce pages in Experience Builder sites.

In containers that don't support the Lightning messaging service, use the pubsub module.

LMS provides a publish/subscribe mechanism that allows for the exchange of messages between components. For the sake of brevity and in order not to duplicate documentation, we won't dive into the technical details of how to publish and subscribe to Lightning messages – but it requires three key steps:

1. **Declare a message channel** using the LightningMessageChannel metadata type.
2. **Publish a message** using the **`publish()`** function from the @salesforce/messageChannel module.
3. **Subscribe to a message** using the **`subscribe()`** function from the **`@salesforce/messageChannel`** module.

10.3.1.1. Uses Of Lightning Message Service

1. To enable communication between Visualforce page, Lightning web components, and Aura components,
2. To access Lightning Message Service API for publishing messages throughout Lightning Experience. Also, it helps in subscribing to messages that originated from anywhere within Lightning Experience via Lightning Message Channel.

3. A specific namespace is associated with Lightning Message Channel. Developers can choose whether they want their message channel to be available to other namespaces or not, ensuring the security of communication on Lightning Message Service.

10.3.1.2. Benefits Of Lightning Message Service

1. One of the significant benefits is increased development time. For instance, if a Visualforce Page or Lightning component tries to reference a message channel that is non-available and that message channel is not exposed, then the code would not compile.
2. This messaging service offers referential integrity between the code that references them and the message channel. It restricts the deletion of message channels, which are referenced by other codes. Further, Lightning Message Service supports auto-adding message channels to packages.
3. As the metadata type is packageable, you can associate a message channel to a particular namespace and make it available/unavailable to other namespaces.

10.3.1.3. Lightning Message Service Limitations

Keep the following in mind when working with Lightning message service.
The Lightning message service supports only the following experiences:

- Lightning Experience standard navigation
- Lightning Experience console navigation
- Salesforce mobile app for Aura and Lightning Web Components, but not for Visualforce pages
- Lightning components used in Experience Builder sites. Support for Experience Builder sites is beta.

In containers that don't support Lightning messaging service, use the pubsub module. Download the module from github.com/developerforce/pubsub.

Example:

Step by step:

1. Create a folder name as **messageChannels** under the directory **force-app > main > default**. Inside the folder, messageChannels create a file with a name as **myMessageChannelName.messageChannel-meta.xml**.
2. Add the following XML to the newly created file messageChannelName.messagChannel-meta.xml. In this post, we have created a message channel file with the name as myMessageChannel.messageChannel-meta.xml.
3. Deploy this message channel file myMessageChannel.messageChannel-meta.xml into your org.

myMessageChannelName.messageChannel-meta.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<LightningMessageChannel xmlns="http://soap.sforce.com/2006/04/metadata">
    <description>Lightning Message Channel For Communicating Across DOM.</description>
    <isExposed>true</isExposed>
    <lightningMessageFields>
        <description>This is a payload field</description>
        <fieldName>greetingMessage</fieldName>
    </lightningMessageFields>
    <masterLabel>myMessageChannel</masterLabel>
</LightningMessageChannel>
```

- The **masterLabel** tag is followed by the name for the message channel
- **The lightningMessageField** tag defines the fields that will carry the event payload. Use multiple lightningMessageField tags if you need more than one field in the event payload.

- The isExposed tags allow the channel to be used across namespaces.

4. In your component's JavaScript file which will act as a publisher component, import the message channel and the Lightning message service functions necessary for working with a message channel.
5. Use **@wire(MessageContext)** to create a MessageContext object, which provides information about the Lightning web component that is using the Lightning message service.

Publisher.js-meta.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<LightningComponentBundle xmlns="http://soap.sforce.com/2006/04/metadata">
    <apiVersion>56.0</apiVersion>
    <isExposed>true</isExposed>
    <targets>
        <target>lightning__AppPage</target>
        <target>lightning__RecordPage</target>
        <target>lightning__HomePage</target>
    </targets>
</LightningComponentBundle>
```

publisher.js

```
import { LightningElement, wire } from 'lwc';
import { publish, MessageContext } from 'lightning/messageService';
import myMessageChannel from '@salesforce/messageChannel/myMessageChannel__c';

export default class Publisher extends LightningElement {
  @wire(MessageContext)
  messageContext;

  buttonHandler(event) {
      const payload = {greetingMessage: "This message is send from the publisher component"};
      publish(this.messageContext, myMessageChannel, payload);
  }
}
```

- we imported the message service features required for publishing and the respective message channel
- we created a MessageContext object, which provides information about the Lightning web component that is using the Lightning message service.
- We need publish and MessageContext from the messageService. To publish the message on the message channel we called the Lightning message service's **publish()** function.
- After the imports, we created the message context.

```
@wire(MessageContext)
  messageContext;
```

- Now, set up is complete to publish to the Message Channel. All that is needed now is to make a call to "publish."

```
publish(this.context, POVMC, payload);
```

- The publish method takes three arguments: the message context, the message channel, and the message itself. The message is an object that contains the fields defined in the Message Channel file.
  `let message = {messageText: 'This is a test'};`

6. Publisher.html file

Publisher.html

```
<template>
  <lightning-card title="Publisher Component">
      <p>
          <lightning-button variant="brand" label="Send Message" title="Send Message"
onclick={buttonHandler}></lightning-button>
      </p>
      <h2>After clicked on the button, look at the Subscriber component</h2>
  </lightning-card>
</template>
```

7. Subscribe.js-meta.xml file

Subscribe.js-meta.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<LightningComponentBundle xmlns="http://soap.sforce.com/2006/04/metadata">
    <apiVersion>56.0</apiVersion>
    <isExposed>true</isExposed>
    <targets>
        <target>lightning__AppPage</target>
        <target>lightning__RecordPage</target>
        <target>lightning__HomePage</target>
    </targets>
</LightningComponentBundle>
```

8. Subscribe.js file

Subscribe.js

```
import { LightningElement, wire } from 'lwc';
import { subscribe, unsubscribe, APPLICATION_SCOPE, MessageContext} from 'lightning/messageService';
import myMessageChannel from '@salesforce/messageChannel/myMessageChannel__c';

export default class Subscriber extends LightningElement {
  message;
  subscription = null;

  @wire(MessageContext)
   messageContext;

    connectedCallback() {
        this.subscribeToMessageChannel();
    }
    disconnectedCallback() {
        this.unsubscribeToMessageChannel();
    }
    subscribeToMessageChannel() {
```

```
        if (!this.subscription) {
            this.subscription = subscribe(
                this.messageContext,
                myMessageChannel,
                (message) => this.handleMessage(message),
                { scope: APPLICATION_SCOPE }
            );
        }
    }
    unsubscribeToMessageChannel() {
        unsubscribe(this.subscription);
        this.subscription = null;
    }
    handleMessage(message) {
        this.message = message.greetingMessage;
    }
}
```

- We imported message service features required for subscribing and the message channel.

```
import { subscribe, unsubscribe, APPLICATION_SCOPE, MessageContext} from
'lightning/messageService';
import myMessageChannel from '@salesforce/messageChannel/myMessageChannel__c';
```

- We used **@wire(MessageContext)** to create a Context object, which provides information about the Lightning web components that are using the Lightning message service.

```
  @wire(MessageContext)
   messageContext;
```

- We used Standard lifecycle hooks to subscribe and unsubscribe to the message channel.

```
    connectedCallback() {
       this.subscribeToMessageChannel();
    }
    disconnectedCallback() {
        this.unsubscribeToMessageChannel();
    }
```

- We encapsulated logic for Lightning message service subscribe and unsubscribe. Here we call the Lightning message service's subscribe() and unsubscribe() functions respectively.
- Subscribe method takes three arguments: the message context, the message channel, and callback to be executed when an event is received on the message channel.

```
    subscribeToMessageChannel() {
        if (!this.subscription) {
            this.subscription = subscribe(
                this.messageContext,
                myMessageChannel,
                (message) => this.handleMessage(message),
                { scope: APPLICATION_SCOPE }
            );
        }
    }
    unsubscribeToMessageChannel() {
```

```
        unsubscribe(this.subscription);
        this.subscription = null;
    }
```

- to receive messages on a message channel from anywhere in the application we are passing **{ scope: APPLICATION_SCOPE }** as the **subscribe()** method's optional fourth parameter.

```
{ scope: APPLICATION_SCOPE }
```

- Finally, we defined the handler function for the message received by component.

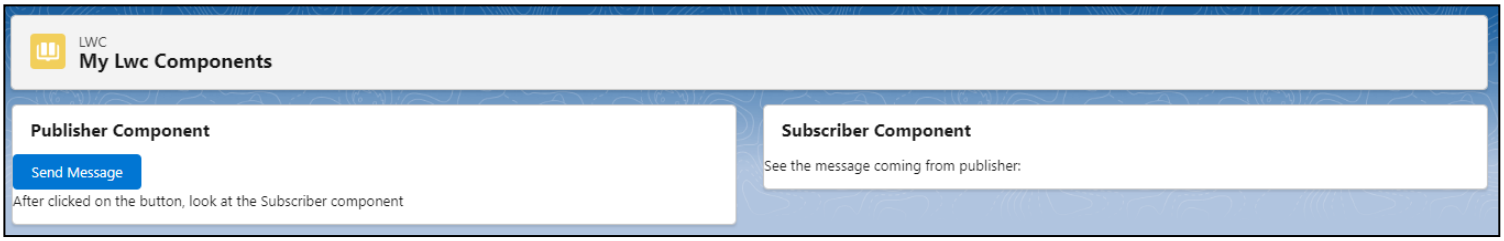
```
    handleMessage(message) {
        this.message = message.greetingMessage;
    }
```

9. Subscriber.html file

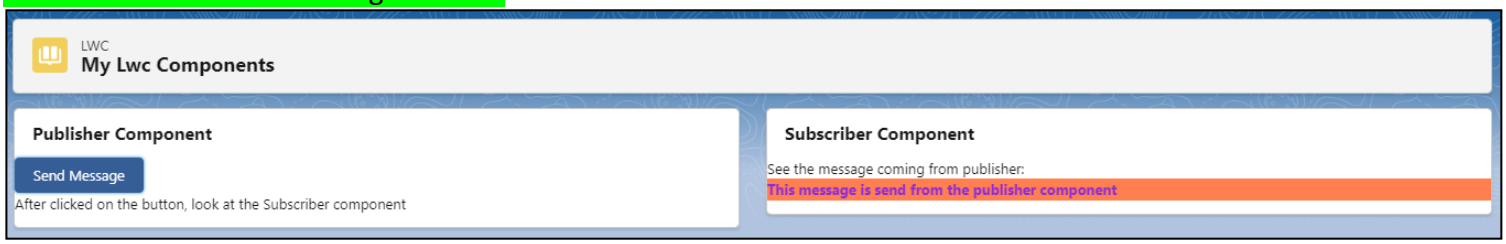
Subscriber.html

```
<template>
  <lightning-card title="Subscriber Component">
    <h2>See the message coming from publisher:</h2>
      <strong>
        <p style="color: blueviolet; background-color:coral " >{message}</p>
      </strong>
  </lightning-card>
</template>
```

Output:



After clicked on Send Message button:



Example:

Inside the messageChannels folder, create a file with a name as **POVMessageChannel.messageChannel-meta.xml.** and deploy this metadata before the lwc component folders.

POVMessageChannel.messageChannel-meta.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<LightningMessageChannel xmlns="http://soap.sforce.com/2006/04/metadata">
  <description>This is for POV implementation</description>
  <isExposed>true</isExposed>
  <masterLabel>POVMessageChannel</masterLabel>
</LightningMessageChannel>
```

publisher2.js-meta.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<LightningComponentBundle xmlns="http://soap.sforce.com/2006/04/metadata">
    <apiVersion>56.0</apiVersion>
    <isExposed>true</isExposed>
    <targets>
        <target>lightning__AppPage</target>
        <target>lightning__RecordPage</target>
        <target>lightning__HomePage</target>
    </targets>
</LightningComponentBundle>
```

publisher2.html

```
<template>
  <lightning-card title="LWC Publisher" icon-name="custom:custom9">
      <div class="slds-m-around_medium">
          <lightning-input label="Message To Send" type="text" value={msg}
onchange={handleChange}></lightning-input>
          <lightning-button label="Publish" variant="brand"
onclick={handlePublish}></lightning-button>
      </div>
  </lightning-card>
</template>
```

publisher2.js

```
import {LightningElement, track, wire} from 'lwc';
import {MessageContext, APPLICATION_SCOPE, publish} from 'lightning/messageService';
import POVMC from "@salesforce/messageChannel/POVMessageChannel__c";
export default class Publisher2 extends LightningElement {
  @track msg = '';

  // Wired message Context
  @wire(MessageContext)
  context;
  handleChange(event) {
      this.msg = event.detail.value;
  }
  handlePublish() {
          let payload = {
              source: "LWC",
              messageBody: this.msg
          };
          publish(this.context, POVMC, payload);
  }
}
```

listener.js-meta.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<LightningComponentBundle xmlns="http://soap.sforce.com/2006/04/metadata">
    <apiVersion>56.0</apiVersion>
    <isExposed>true</isExposed>
    <targets>
        <target>lightning__AppPage</target>
        <target>lightning__RecordPage</target>
        <target>lightning__HomePage</target>
    </targets>
</LightningComponentBundle>
```

listener.html

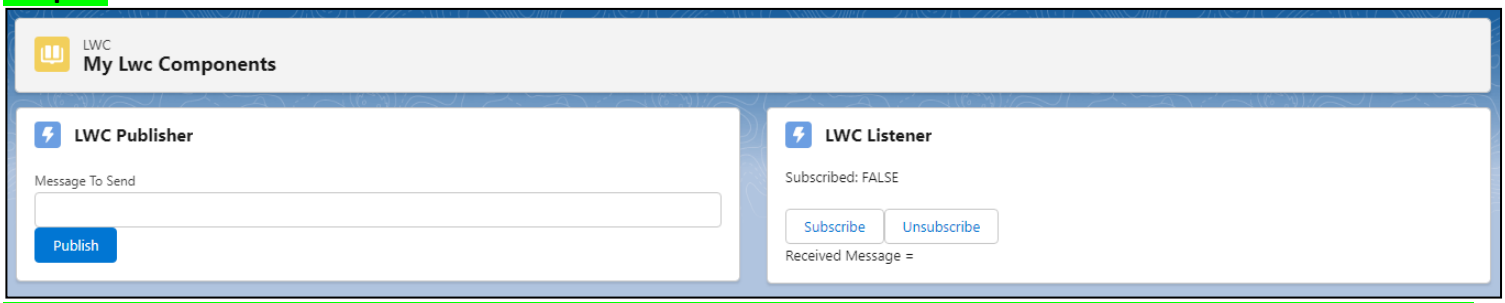
```
<template>
  <lightning-card title="LWC Listener" icon-name="custom:custom9">
      <div class="slds-m-around_medium">
          <p>Subscribed: {subscribeStatus}</p>
          <br>
          <lightning-button label="Subscribe" onclick={handleSubscribe}></lightning-button>
          <lightning-button label="Unsubscribe" onclick={handleUnsubscribe}></lightning-
button>
          <p>Received Message = </p>
          <div if:true={receivedMessage} class="slds-box">
              <lightning-formatted-text value={receivedMessage}></lightning-formatted-text>
          </div>
      </div>
  </lightning-card>
</template>
```

listener.js

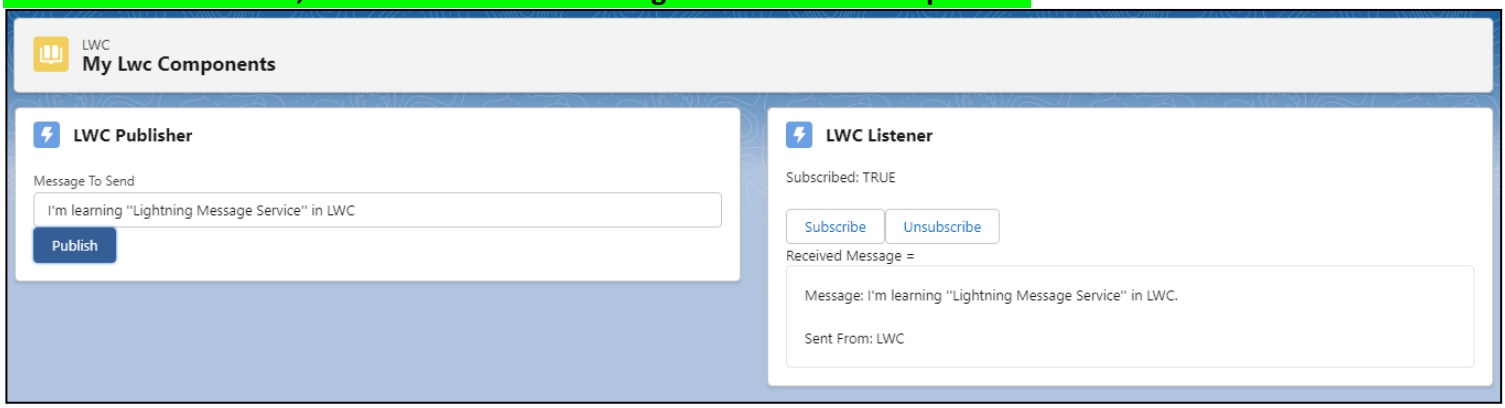
```
import {LightningElement, track} from 'lwc';
import {createMessageContext, releaseMessageContext, APPLICATION_SCOPE, subscribe,
  unsubscribe} from 'lightning/messageService';
import POVMC from "@salesforce/messageChannel/POVMessageChannel__c";
export default class pov_lwc_listener extends LightningElement {
  @track receivedMessage = '';
  @track subscription = null;
  // NOT Using Wired MessageContext.
  // Instead using createMessageContext,releaseMessageContext to subscribe unsubscribe
  // @wire(MessageContext)
  // context;
  context = createMessageContext();
  handleSubscribe() {

      if (this.subscription) {
          return;
      }
```

```
        this.context = createMessageContext();
        this.subscription = subscribe(this.context, POVMC, (message) => {
            this.handleMessage(message);
        }, {
            scope: APPLICATION_SCOPE
        });
    }
    handleMessage(event) {
        if (event) {
            let message = event.messageBody;
            let source = event.source;
            this.receivedMessage = 'Message: ' + message + '.\n \n Sent From: ' + source;
        }
    }
    handleUnsubscribe() {
        unsubscribe(this.subscription);
        this.subscription = undefined;
        releaseMessageContext(this.context);
    }
    get subscribeStatus() {
        return this.subscription ? 'TRUE' : 'FALSE';
    }
}
```

Output:



After clicked Subscribe button on Listener Component; when we write anything to input on Publisher and then clicked Publish button, we can receive the message from listener component.



10.3.2. Publish-Subscribe Model

The Publish-Subscribe pattern/model is same as Application Event in Lighting Component. If you want to communicate between components those are not bounding in parent child relationship, but available in same page (say, in same app-builder page), then we will use publish-subscribe pattern.

In a publish-subscribe pattern, one component publishes an event. Other components subscribe to receive and handle the event. Every component that subscribes to the event receives the event.

In containers that don't support Lightning messaging service, use the pubsub module. Download the module from github.com/developerforce/pubsub.

Pubsub module support below three method

1. Register
2. UnRegister
3. Fire

As of the Summer '20 major release, the Lightning message service (LMS) is now Generally Available (GA).
all use of c/pubsub in any sample apps have been retired and removed its core code as well.

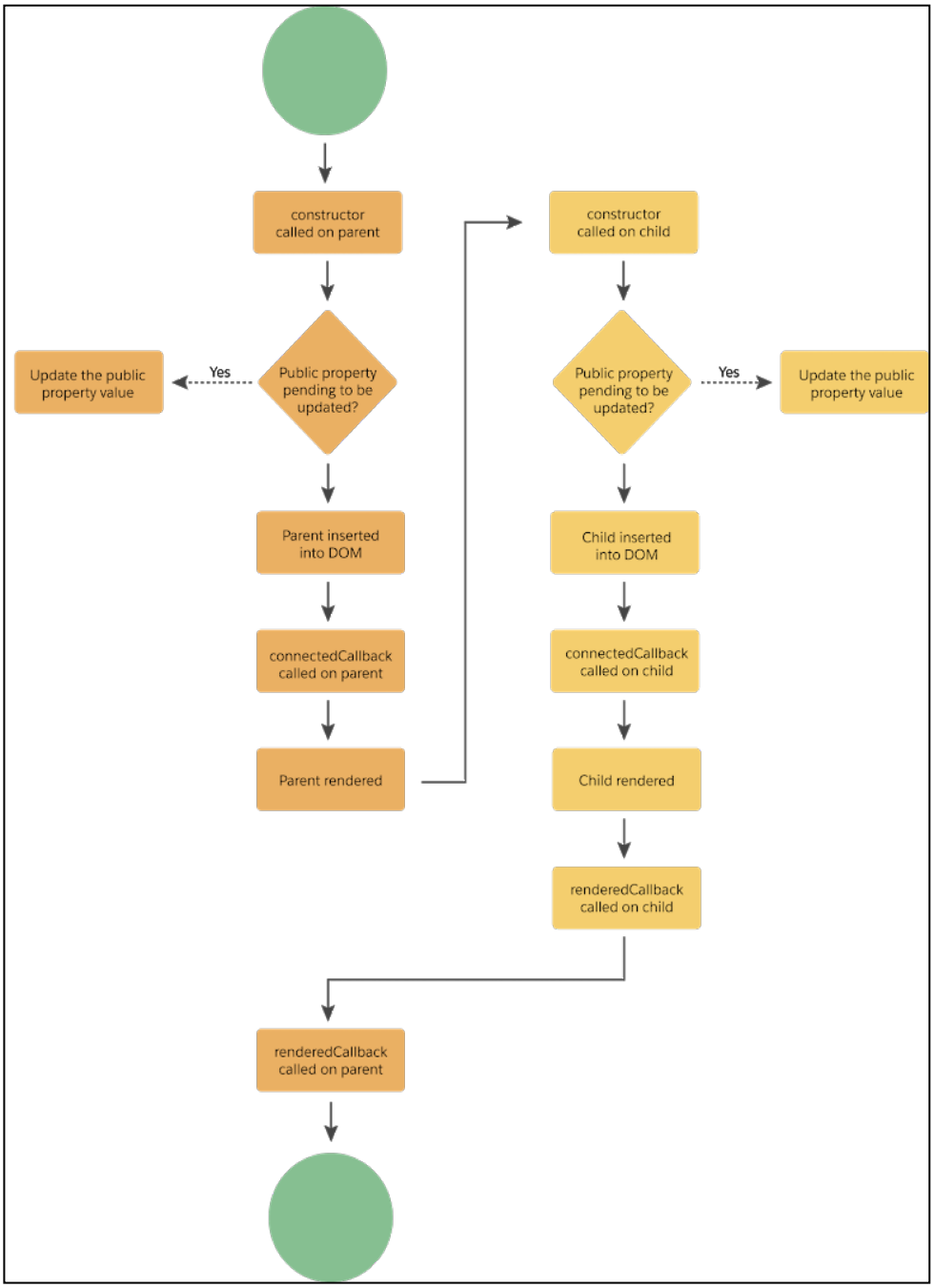
11. LWC Lifecycle Hook

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

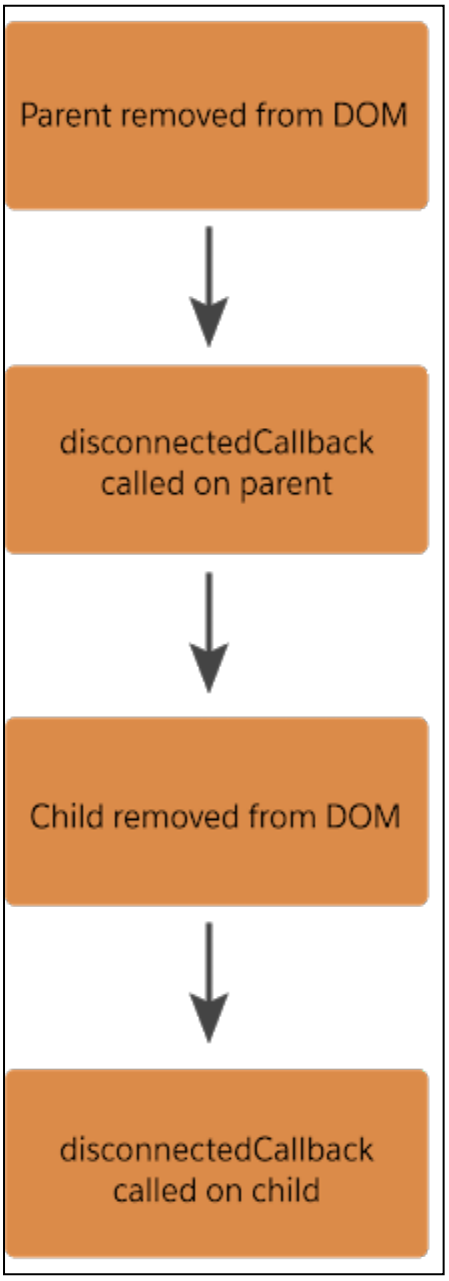
Lightning web components have a lifecycle managed by the framework. The framework creates components, inserts them into the DOM, renders them, and removes them from the DOM. It also monitors components for property changes. It Manages the flow of the related component (Parent to child and child to parent).

Lifecycle Flow:

This diagram shows what happens when a component instance is removed from the DOM.



This diagram shows what happens when a component instance is removed from the DOM.



https://developer.salesforce.com/docs/component-library/documentation/en/lwc/lwc.create_lifecycle_hooks

Lifecycle hooks are methods that are automatically called by the framework at specific points in a component's lifecycle. There are several lifecycle hooks available in LWC, including:

1. constructor()
2. connectedCallback()
3. disconnectedCallback()
4. renderedCallback()
5. errorCallback()

11.1. Constructor()

The constructor() method fires when a component instance is created. Don't add attributes to the host element during construction. You can add attributes to the host element in any other lifecycle hook.

The constructor flows from parent to child.

These requirements from the **HTML: Custom elements spec** apply to the constructor().

- The first statement must be super() with no parameters. This call establishes the correct prototype chain and value for this. Always call super() before touching this.
- Don't use a return statement inside the constructor body, unless it is a simple early-return (return or return this).
- Don't use the document.write() or document.open() methods.
- Don't inspect the element's attributes and children, because they don't exist yet.
- Don't inspect the element's public properties, because they're set after the component is created.

Syntax:

```
import { LightningElement } from 'lwc'

export default class MyComponent extends LightningElement{
  constructor(){
      super();
  this.greeting = 'Welcome';
    }
}
```

constructor.html

```
<template>
  <div>Lifecycle hooks – Constructor in LWC</div>
</template>
```

constructorExp.js

```
import { LightningElement } from 'lwc';
export default class ConstructorExp extends LightningElement {
    constructor() {
        super();
        this.classList.add('new-class');
        console.log('In Constructor');
    }
}
```

11.2. connectedCallback()

The moment the component is connected to the DOM this life cycle hook is invoked. By the time this method is invoked, all the @api properties would have received the data from the parent and we can use that data to make a call to Apex method.

You can't access child elements from the callbacks because they don't exist yet. To access the host element, use **this**.

Use connectedCallback() to interact with a component's environment. For example, use it to:

- Establish communication with the current document or container and coordinate behavior with the environment.
- Perform initialization tasks, such as fetch data, set up caches, or listen for events
- Subscribe and Unsubscribe from a Message Channel.

11.3. disconnectedCallback()

The disconnectedCallback() lifecycle hook fires when a component is removed from the DOM.

Use disconnectedCallback() to clean up work done in the connectedCallback(), like purging caches or removing event listeners.

callback.html

```
<template>
  <div>Lifecycle hooks – Callback- LWC</div>
</template>
```

callback.js

```
import { LightningElement } from 'lwc';
export default class Callback extends LightningElement {
    constructor(){
        super();
      console.log('Inside constructor');
  }
  connectedCallback() {
      console.log('Inside connected callback');
  }
  disconnectedCallback(){
        console.log('Inside disconnected callback');
  }
}
```

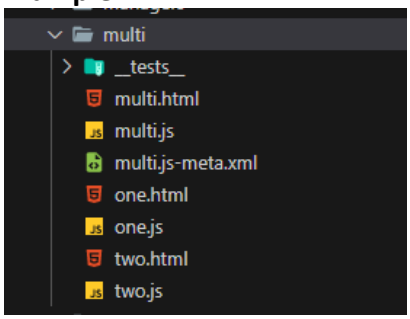
11.4. render ()

For complex tasks like conditionally rendering a template, use render() to override the standard rendering functionality. This function may be invoked before or after connectedCallback().

This method must return a valid HTML template. Import a reference to a template and return the reference in the render() method.

You may want to render a component with more than one look and feel, but not want to mix the HTML in one file. For example, one version of the component is plain, and another version displays an image and extra text. In this case, you can import multiple HTML templates and write business logic that renders them conditionally. This pattern is similar to the code splitting used in some JavaScript frameworks.

Example:



Note: It's rare to define render() in a component. It's more common to use an `if:true|false` directive to render nested templates conditionally.

multi.js-meta.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<LightningComponentBundle xmlns="http://soap.sforce.com/2006/04/metadata">
    <apiVersion>56.0</apiVersion>
    <isExposed>true</isExposed>
    <targets>
        <target>lightning__AppPage</target>
        <target>lightning__RecordPage</target>
        <target>lightning__HomePage</target>
    </targets>
</LightningComponentBundle>
```

multi.html

```
<template>
  <!-- unused -->
</template>
```

multi.js

```
import { LightningElement } from 'lwc';
import templateOne from './one.html';
import templateTwo from './two.html';

export default class Multi extends LightningElement {
    showTemplateTwo = false;

    render() {
        return this.showTemplateTwo ? templateTwo : templateOne;
    }

    switchTemplate() {
        this.showTemplateTwo = !this.showTemplateTwo;
    }
}
```

One.html

```
<template>
  <lightning-card title="-1-">

    <p style="color: rgb(223, 114, 13);">
       <strong> This is template one. </strong>
    </p>
  </lightning-card>
  <p>
    <lightning-button onclick={switchTemplate} label="Switch Templates" ></lightning-button>
  </p>
</template>
```

one.js

```
import { LightningElement } from 'lwc';

export default class One extends LightningElement {}
```

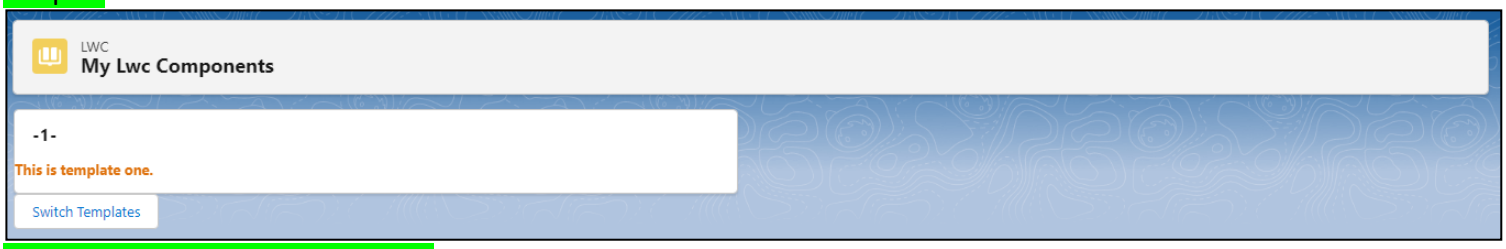
two.html

```
<template>
  <lightning-card title="-2-">
  <p style="color: blueviolet;">
     <strong> This is template two. </strong>
  </p>
</lightning-card>
  <p>
    <lightning-button onclick={switchTemplate} label="Switch Templates" ></lightning-button>
  </p>
</template>
```

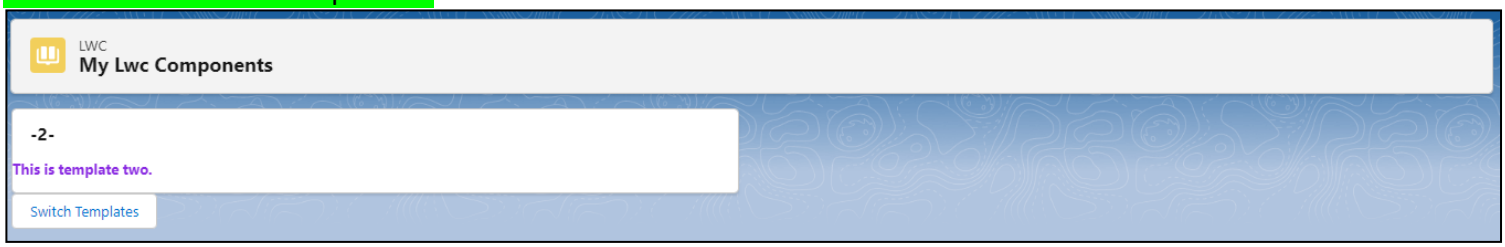
two.js

```
import { LightningElement } from 'lwc';

export default class Two extends LightningElement {}
```

Output:



After clicked 'Switch Templates' :



11.5. renderedCallback()

The renderedCallback() is unique to Lightning Web Components. Use it to perform logic after a component has finished the rendering phase.

Once the component is completely rendered that's when this Life Cycle Hook in LWC gets invoked. Since the complete component is rendered we can have some business logic that involves the DOM.

This hook flows from child to parent.

A component is usually rendered many times during the lifespan of an application. To use this hook to perform a one-time operation, use a boolean field like hasRendered to track whether `renderedCallback()` has been executed. The first time renderedCallback() executes, perform the one-time operation and set `hasRendered = true`. If `hasRendered = true`, don't perform the operation.

It's best to attach event listeners declaratively in the HTML template. However, if you want to attach an event listener to a template element programatically in JavaScript, use renderedCallback(). If a listener is added to the same element repeatedly, the browser removes the duplicates if the event type, event listener, and options are the same.

When a template is rerendered, the LWC engine attempts to reuse the existing elements. In the following cases, the engine uses a diffing algorithm to decide whether to discard an element.

- Elements created using the for:each directive. The decision to reuse these iteration elements depends on the key attribute. If key changes, the element may be rerendered. If key doesn't change, the element isn't rerendered, because the engine assumes that it didn't change.
- Elements received as slot content. The engine attempts to reuse an element in a <slot>, but the diffing algorithm determines whether to evict an element and recreate it.

```
import { LightningElement } from 'lwc'

export default class HelloWorld extends LightningElement{

  renderedCallback(){

      console.log('RENDERED CALLBACK');

    }

}
```

11.6. errorCallback()

Called when a descendant component throws an error in one of its lifecycle hooks. The **`error`** argument is a JavaScript native error object, and the **`stack`** argument is a string.

Implement this hook to create an error boundary component that captures errors in all the descendant components in its tree. The error boundary component can log stack information and render an alternative view to tell users what happened and what to do next. The method works like a JavaScript catch{} block for components that throw errors in their lifecycle hooks. It's important to note that an error boundary component catches errors only from its children, and not from itself.

You can create an error boundary component and reuse it throughout an app. It's up to you where to define those error boundaries. You can wrap the entire app, or every individual component. Most likely, your architecture falls somewhere in between. A good rule of thumb is to think about where you'd like to tell users that something went wrong.

```
import { LightningElement } from 'lwc'
```

```
export default class HelloWorld extends LightningElement{
  errorCallback(){
      console.log('ERROR CALLBACK');
    }
}
```

Example:

```
<!-- boundary.html -->
<template>
  <template lwc:if={error}>
      <error-view error={error} info={stack}></error-view>
  </template>
  <template lwc:elseif={error}>
      <healthy-view></healthy-view>
  </template>
</template>
```

```
// boundary.js
import { LightningElement } from 'lwc';
export default class Boundary extends LightningElement {
    error;
    stack;
    errorCallback(error, stack) {
        this.error = error;
    }
}
```

You don't have to use if:[true|false] in a template. For example, let's say you define a single component template. If this component throws an error, the framework calls errorCallback and unmounts the component during re-render.

```
<!-- boundary.html -->
<template>
  <my-one-and-only-view></my-one-and-only-view>
</template>
```

Note:

The **`errorCallback()`** hook doesn't catch errors like click callbacks and async operations that fall outside of the component lifecycle (`constructor()`, `connectedCallback()`, `render()`, `renderedCallback()`). It's also important to note that `errorCallback()` isn't a JavaScript `try/catch`, it's a component lifecycle guard.

12. LWC Testing with Jest Framework

Testing for LWC components can be done by using **JEST**, using jest for unit testing in LWC can provide us easy mocking of test data and great exception handling.

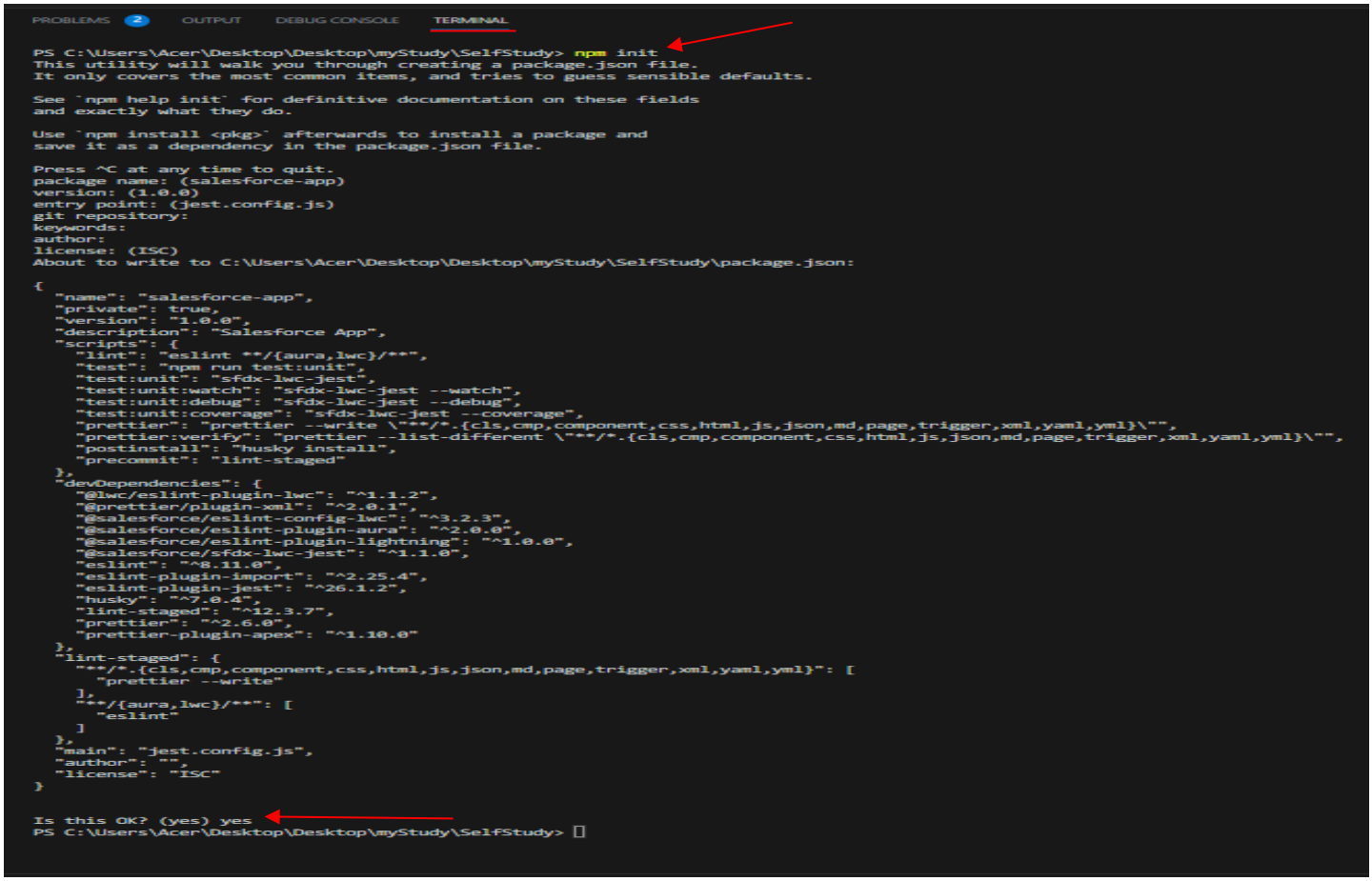
Jest is a JavaScript testing Framework which focuses on simplicity. It works with Babel, TypeScript, Node, React, Angular, Vue, and more projects. Jest tests are only local and are saved and run independently of Salesforce. Jest tests are fast as they don't run in a browser or connect to an org.

When approaching the testing of your components, it's important to remember the two types of testing, functional and unit. Salesforce.com briefly mentions in their documentation here that functional (end-to-end) testing is not recommended for Lightning Web Components; however, Jest unit testing is strongly encouraged. Jest is a JavaScript testing framework developed with an emphasis on simplicity. These tests run locally and will not be pushed to Salesforce.com as part of the component bundle. Each test lives in a separate folder inside of the component bundle and contain any number of tests inside test suites.

Set up Jest Test Framework:

To test Lightning Web Components, we need to do the following prerequisites to set up the environment.

1. Open Visual Studio Code Project.
2. Use below link to install **Node.js** and **npm**. When Node.js gets installed, npm also installs.
   https://nodejs.org/en
3. Generally, Salesforce DX Project does not have a **package.json** file. Execute the following command from the terminal of the Visual Studio Code. It will create package.json file in the project directory. After execution, system will be asking the project information; keep the defaults and the press *Enter* key until package.json file has been created.
   - **`npm init`**



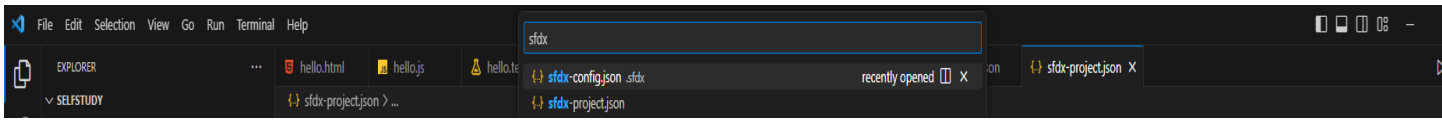
4. Execute the following **npm** commands at the top-level directory of your *Salesforce DX* project:
   - `npm install`
5. Then run below command to install sfdx-lwc-jest and its dependencies into each Salesforce DX project. sfdx-lwc-jest works in *Salesforce DX* projects only.
   - `npm install @salesforce/sfdx-lwc-jest`

Note:

If you get an "Invalid sourceApiVersion" error it is due to an updated VS Code Extension with the latest Salesforce release.

error Invalid sourceApiVersion found in sfdx-project.json. Expected 51.0, found 52.0

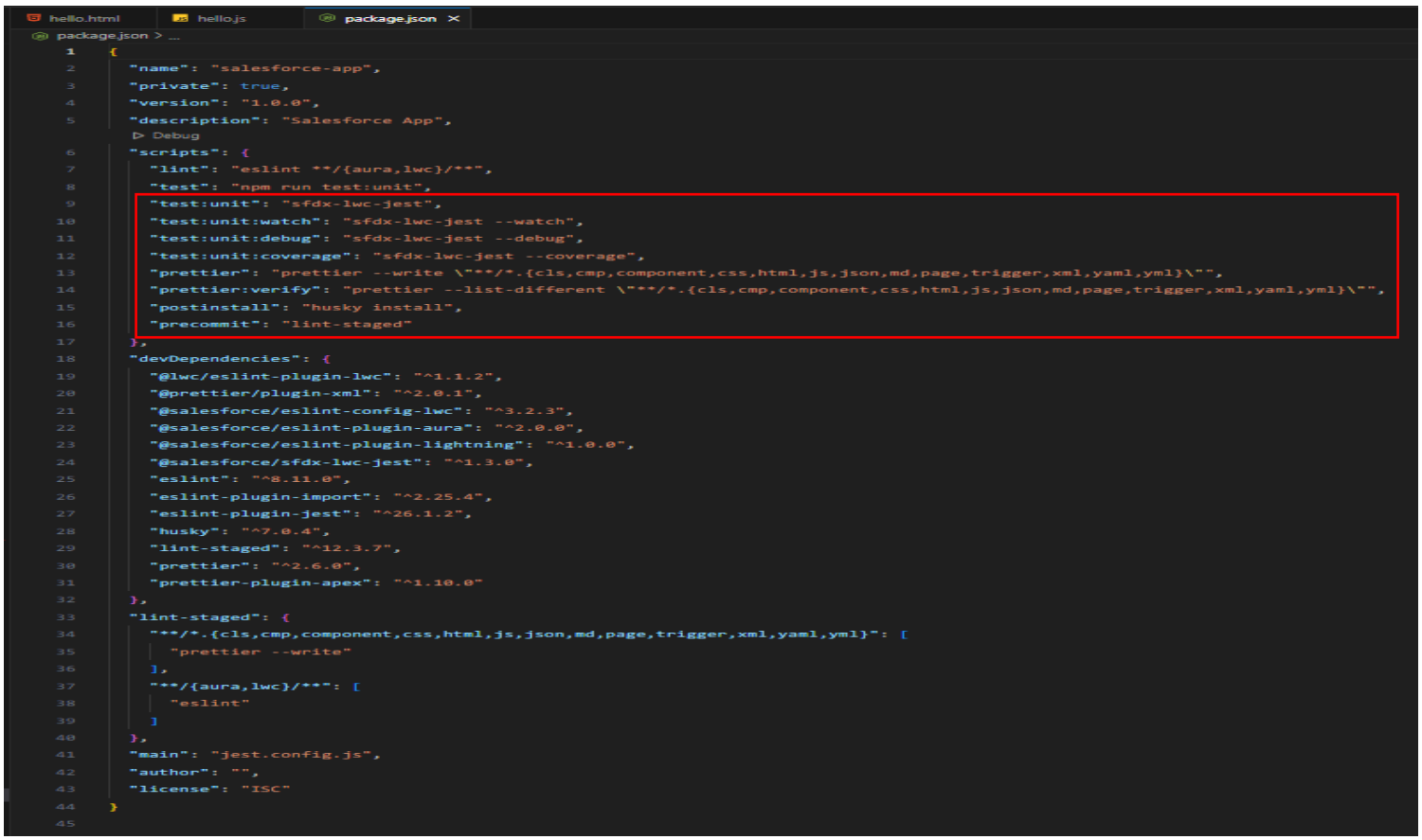
1. In Visual Studio Code, in the top-level directory, open `sfdx-project.json`.



2. Update the line of code with "sourceApiVersion" to the Expected version from the error message you received.
   `"sourceApiVersion": "55.0"` Copy (or write latest version)
3. Save the file.

6. By default, an SFDX project includes these script entries in the scripts block of its **package.json** file. If the project's file doesn't include, add them.

```
    "test:unit": "sfdx-lwc-jest",
    "test:unit:watch": "sfdx-lwc-jest --watch",
    "test:unit:debug": "sfdx-lwc-jest --debug",
  "test:unit:coverage": "sfdx-lwc-jest --coverage",
```



Writing a Basic Test: (test file is usually automatically created)

1. In Visual Studio Code, right click on the **LWC** directory and select **SFDX: Create Lightning Web Component**. And create **hello** lwc component.

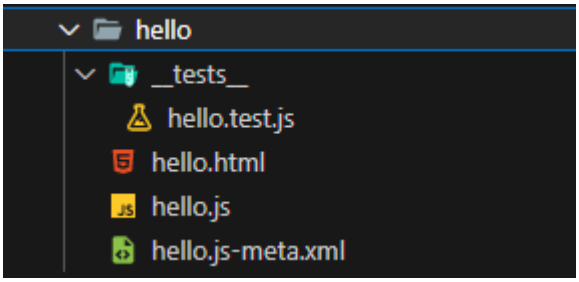
hello.html

```
<template>
  <div>{greeting}</div>
</template>
```

hello.js

```
import { LightningElement } from 'lwc';

export default class Hello extends LightningElement {
  greeting = 'world';
}
```

2. In Visual Studio Code, right click on the **hello** directory and select New Folder.
3. Name the folder as "__tests__" and press *Enter* key to proceed.
4. After that right-click on the **__tests__** directory, select New File.
5. Name the file as "hello.test.js" and then press *Enter* key.
6. **Hello** LWC component looks like below,



hello.test.js

```
import { createElement } from 'lwc';
import Hello from 'c/hello';

describe('c-hello', () => {
    afterEach(() => {
        // The jsdom instance is shared across test cases in a single file so reset the DOM
        while (document.body.firstChild) {
            document.body.removeChild(document.body.firstChild);
        }
    });

    it('TODO: test case generated by CLI command, please fill in test logic', () => {
        // Arrange
        const element = createElement('c-hello', {
            is: Hello
        });

        // Act
        document.body.appendChild(element);

        // Assert
        // const div = element.shadowRoot.querySelector('div');
        expect(1).toBe(1);
    });
});
```

All tests must use below structure:

1. **Imports:**

   First, test the imports with createElement method and is available only in tests. The code should import the component to test, which in this case is c/hello. Using these imports, the component under test is created later.

```
import { createElement } from 'lwc';
import Hello from 'c/hello';
```

2. **Cleanup:**

   The Jest afterEach() method is used to reset the DOM at the end of the test. Every test file has a single instance of jsdom, and changes aren't reset between tests inside the file. As a best practice, it is important to clean up between tests, so that one test's output doesn't affect another test.

```
    it('TODO: test case generated by CLI command, please fill in test logic', () => {
        // Arrange
        const element = createElement('c-hello', {
            is: Hello
        });

        // Act
        document.body.appendChild(element);

        // Assert
        // const div = element.shadowRoot.querySelector('div');
        expect(1).toBe(1);
    });
```

3. **Describe() block:**

   A describe block() determines a test suite. A test suite has one or more tests that belong together from a functional point of view. For hello.test.js, a single describe is sufficient.

```
describe('c-hello', () => {
    afterEach(() => {
        // The jsdom instance is shared across test cases in a single file so reset the DOM
        while (document.body.firstChild) {
            document.body.removeChild(document.body.firstChild);
        }
    });
```

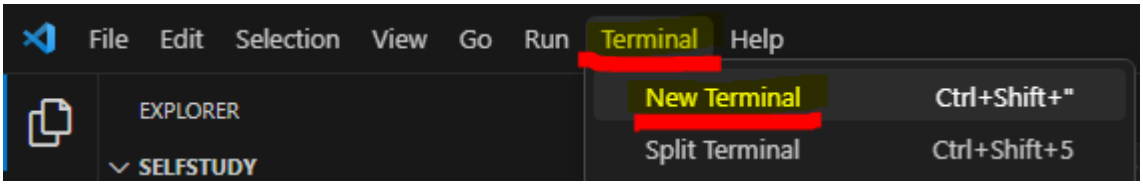
4. **it() block:**

   A single it() block describes a single test.

```
    it('TODO: test case generated by CLI command, please fill in test logic', () => {
        // Arrange
        const element = createElement('c-hello', {
            is: Hello
        });

        // Act
        document.body.appendChild(element);

        // Assert
        // const div = element.shadowRoot.querySelector('div');
        expect(1).toBe(1);
    });
```

Running the Test:

1. In Visual Studio Code, click on Terminal > New Terminal. This opens a terminal in Visual Studio Code. The terminal defaults to the current project top-level directory.



2. Execute the following command in the terminal,
   - To run all tests for your project. `npm run test:unit`
   - To run the specific test class. `npm run test:unit hello.test.js`
3. The test Passes.

```
PROBLEMS 2   OUTPUT   DEBUG CONSOLE   TERMINAL

PS C:\Users\Acer\Desktop\Desktop\myStudy\SelfStudy> npm run test:unit hello.test.js

> salesforce-app@1.0.0 test:unit
> sfdx-lwc-jest hello.test.js

info `sfdx-lwc-jest [options]` runs Jest unit tests in SFDX workspace

Options:
      --version               Show version number                          [boolean]
      --coverage              Collect coverage and display in output
                                                   [boolean] [default: false]
  -u, --updateSnapshot        Re-record every snapshot that fails during a test
                              run                  [boolean] [default: false]
      --verbose               Display individual test results with the test suite
                              hierarchy            [boolean] [default: false]
      --watch                 Watch files for changes and rerun tests related to
                              changed files        [boolean] [default: false]
      --debug                 Run tests in debug mode
                              (https://jestjs.io/docs/en/troubleshooting)
                                                   [boolean] [default: false]
      --skipApiVersionCheck   Disable the "sourceApiVersion" field check before
                              running tests. **Warning** By disabling this check
                              you risk running tests against stale versions of
                              the framework. See details here:
                              https://github.com/salesforce/sfdx-lwc-jest#disabli
                              ng-the-sourceApiVersion-check
                                                   [boolean] [default: false]
      --help                  Show help                                    [boolean]

Examples:
  sfdx-lwc-jest --coverage  Collect coverage and display in output
  sfdx-lwc-jest -- --json   All params after `--` are directly passed to Jest
error The following argument(s) are not recognized by lwc-jest: hello.test.js
If you wish to pass these arguments along to Jest, please add the '--' flag
 PASS  force-app/main/default/lwc/hello/__tests__/hello.test.js
  c-hello
    √ TODO: test case generated by CLI command, please fill in test logic (23 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        3.438 s
Ran all test suites matching /hello.test.js/i.
PS C:\Users\Acer\Desktop\Desktop\myStudy\SelfStudy>
```

Run Tests Continuously During Development:

For a single component each time you save changes, to run all tests, change directories to the component directory and run the sfdx-lwc-jest command with the watch parameter.

- `npm run test:unit:watch`

For this option, Node relies on Git to "watch" the code. To use this option, be sure to have Git initialized for your project. To run all tests for a single component every time you save changes, change directories to the component directory and run the npm command below that utilizes sfdx-lwc-jest with the --watch parameter. As mentioned above you could also run this from the base of the project and have all tests in the project run for every change.

Jest now watches all component files for updates and runs all relevant tests every time it detects a change.

Run Tests in Jest Debug Mode:

To run the project's Jest tests in debug mode, run the sfdx-lwc-jest command with the –debug parameter.

- `npm run test:unit:debug`

Conclusion:

Jest test Framework focuses on the Black-Box testing and can be written without application knowledge. However, public methods, templates, and events must be tested first, and most importantly it doesn't require a 100% code coverage. Write Jest tests to do the following,

1. Test a component's public API (`@api` properties and methods, events)
2. Test basic user interaction (clicks)
3. Verify the DOM output of a component
4. Verify that events fire when expected

13. LWC Best Practices

With Lightning Web Components (LWC), Salesforce developers can now work with open web standards like standard JavaScript and HTML, even if they have never worked with Salesforce before. This means existing developers can now use standard techniques and tools previously unavailable to them.

1. **LWC Component Bundle Naming Convention**
   - **Html file** : Use camel case to name your component and use **kebab-case** to **reference a component** in the markup
   - **JavaScript File** : Java Script **Class name** should be in **PascalCase**
   - **Bundle Component :** use camelCase.
2. **Calling Apex From LWC**

   There are two ways to call Apex class in the Lightning web component.

   1. Imperatively
   2. Wire
      1. Wire a property
      2. Wire a function

Wire Vs Imperatively:

Use `@wire` over imperative method invocation as per Lightning component best practices. The @wire component fits nicely into the overall reactive architecture of Lightning Web Components. In order to improve performance, Salesforce is developing features that are only available with @wire. Nevertheless, there are a few use cases that require you to use imperative Apex.

Wire Property Vs Wire Function:

Prefer wiring to a property. This best practice applies to @wire in general (not just to wiring Apex methods).

3. **Lightning Data Service (LDS)**

   As per LWC best practice use Lightning Data Service functions to create,Record, and delete a record over invoking Apex methods. Yes there are some use cases where you need to multiple records then we can use Apex methods.

   Lightning Data Service is built on top of User Interface API. UI API is a public Salesforce API that Salesforce uses to build Lightning Experience. Like its name suggests, UI API is designed to make it easy to build Salesforce UI. UI API gives you data and metadata in a single response

   Give preference to user interface form-type in below order.

   1. lightning-record-form : It is the fastest/most productive way to build a form.
   2. lightning-record-view-form : If you need more control over the layout, want to handle events on individual input fields, or need to execute pre-submission
   3. @wire(getRecord) : If you need even more control over the UI, or if you need to access data without a UI

4. **Event In LWC**

   There are typically 3 approaches for communication between the components using events.

   1. Communication using Method in LWC (Parent to Child)
   2. Custom Event Communication in Lightning Web Component (Child to Parent)
   3. Publish Subscriber model in Lightning Web Component Or Lightning Message Service (Two components which don't have a direct relation)

   Here is some recommendation for DOM Event.

   1. No uppercase letters
   2. No Spaces
   3. use underscores to separate words
   4. Don't prefix your event name with the string "on".

   Learn more about Events in lightning web components here. Avoid using Lightning Message Service or pubsub when not needed.

5. **Streaming API, Platform Event, Change Data Capture**

   In the lightning/empApi module, subscribers and listeners can subscribe to streaming channels. All streaming channels are supported, such as platform events, PushTopic events, generic events, and Change Data Capture events. This component requires API version 44.0 or later. There is a shared connection between lightning/empApi and CometD.

6. **How To Debug LWC**

   Use Chrome's pretty JS setting to see unminified JavaScript and Debug Proxy Values for Data. Here is example.

   - Enable Debug mode
     - It gives unminified Javascript
     - Console warnings
     - Pretty data structure
   - Caution – it reduces the performance of Salesforce, make sure it's disabled in production

7. **Use Storable Action/ Cache Data**

   Use Storable Action, It will reduces call to Server. Syntax:

```
@AuraEnabled(cacheable=true)
```

   Caution: A storable action might result in no call to the server. Never mark as storable an action that updates or deletes data.

   For storable actions in the cache, the framework returns the cached response immediately and also refreshes the data if it's stale. Therefore, storable actions might have their callbacks invoked more than once: first with cached data, then with updated data from the server.

8. **Build Reusable Lightning Web Components**

   Modularity is key to creating a scalable application by reusing components. We rarely write code that is abstract enough to be reusable everywhere. You should write components not to be reused out of the box, but to be abstract and composable enough to serve that purpose in a variety of implementations.

9. **Styling Reusable Components**

   LWC encapsulates and isolates a component's stylesheet. Does it mean it is hard to style a child component from a parent?

   - Spend some time at design time to think about possible variants of your component.
   - Use CSS Custom Properties for custom styles
   - Use Styling Hooks to override styling of Base Components.
   - Favor Styling Hooks and CSS Custom Properties over component attributes.

10. **Other Best Practices**

    Here are some other LWC best practices which we can follow while building LWC components.

    - Use UI API Wire Adapters and Lightning Base Components
      - Gives the admin the opportunity to configure your components without modifying code by configuring lists views, record layouts.
      - If required don't hesitate to restrict your components visibility to certain objects
    - Think whether the component needs to be tied to an object, or can it be object agnostic
      - Don't use static references
      - If needed, create object agnostic classes
    - Extract utility methods outside your LWC components
      - LWC components should only deal with UI-related logic.
    - Favor public properties over public methods
      - Properties can be set directly in the template, while a method requires the consumer to render the component first, retrieve it back from the DOM, and invoke the method

Resources

1. https://developer.salesforce.com/docs/component-library/
2. https://www.lightningdesignsystem.com/
3. https://trailhead.salesforce.com/
4. https://lwc.dev/guide
5. https://blogs.perficient.com/
6. https://www.apexhours.com/
7. https://developer.salesforce.com/blogs
8. https://salesforceblue.com/
9. https://lwc-redux.com/basic-tutorial
10. https://www.salesforceben.com/
11. https://sfdclesson.com/lightning-web-component-decorators/
12. https://www.sfdckid.com/
13. https://www.salesforcecodecrack.com/2019/06/render-list-of-items-in-lwc.html
14. https://www.mstsolutions.com/
15. https://blog.webnersolutions.com/create-component-in-lwc-and-display-in-salesforce-page/
16. https://salesforceblue.com/
17. https://debarunsengupta.medium.com/apex-code-to-display-image-from-a-rich-text-area-field-in-salesforce-63272dc4cce
18. https://help.salesforce.com/
19. https://recipes.lwc.dev/
20. https://studysalesforce.com/Developer/Lightning-Web-Component
21. https://salesforce.stackexchange.com/
22. http://bobbuzzard.blogspot.com/2022/02/lightning-web-component-getters.html
23. https://github.com/sfwiseguys/LWCComms/tree/master/force-app/main/default
24. https://blog.webnersolutions.com/getter-in-lwc/
25. https://dineshyadav.com/
26. https://salesforcebinge.com/2021/02/06/search-accounts-using-lwc-map-view/
27. https://developer.salesforce.com/blogs/2021/05/inter-component-communication-patterns-for-lightning-web-components?_ga=2.19151480.1115298334.1681331614-1480446641.1673265161&_gac=1.53362010.1680637924.Cj0KCQjwla-hBhD7ARIsAM9tQKuNfwZ2pHCG0hz8wlFJRjAKYGtrQH85RmE-fWX_6NUwb2CW73w4kQMaAhyYEALw_wcB
28. https://pashtek.com/

29. https://blog.salesforcecasts.com/
30. https://jayakrishnasfdc.wordpress.com/
31. https://salesforcediaries.com/
32. https://salesforceblue.com/communicate-across-dom-in-lightning-web-components/
33. https://pashtek.com/lightning-message-service-in-lwc/
34. https://www.mstsolutions.com/technical/events-in-lwc/
35. https://salesforceblue.com/communicate-across-dom-in-lightning-web-components/
36. https://trailhead.salesforce.com/content/learn/modules/test-lightning-web-components/set-up-jest-testing-framework
37. https://www.mirketa.com/hands-on-lwc-jest-unit-testing/
38. https://www.mstsolutions.com/technical/lwc-testing-with-jest-framework/
39. https://www.mirketa.com/hands-on-lwc-jest-unit-testing/
40. https://techkasetti.com/blog/index.php/2021/02/18/jest-unit-testing-with-lwc/
41. https://tigerfacesystems.com/blog/lwc-nested-component-testing
42. https://www.jamessimone.net/blog/joys-of-apex/advanced-lwc-jest-testing/
43. https://sudipta-deb.in/2023/01/best-practices-for-lightning-web-components.html
44. https://climbtheladder.com/10-lwc-best-practices/