LWC DAILY CLASS DOCUMENT

Lightning Web Component Topics:

Introduction to Lightning Web Component
Setup Visual Studio Code for Salesforce
Understand Lightning Web Component Bundle
Create First Lightning Web Component & Deploy to Salesforce Org
Migrating Markup, CSS, JS | Aura to Lightning Web Component
Decorators - @api, @track @wire
Use @api Decorator | Private Vs Public Property in Lightning Web Component
Use @wire decorator | Lightning Data Service Wire Adapter
Import Object & Fields Reference Vs Use Object & Fields as String
Use of @wire decorator | Wire Apex Class Method in Lightning Web Component
Decorate Property or Function with @wire Decorator
Use of Property | HTML Template Bind Data
Use of Getter | HTML Template Data Binding | Lightning Web Component
Use querySelector to Fetch Data | HTML Template Data Binding
Use querySelectorAll to fetch data | HTML Template Data Binding
Use of Getter & Setter | Lightning Web Component
Message Passing through Parent to Child Component

Introduction to Lightning Web Component

Introduction:

Lightning web component is an implementation of the of W3'C Web component Standards It supports the part of web component that works in browser and add parts supported by salesforce as well

Benefits of Lwc

- ★ Uses Modern js with ES6+
- ★ Uses web Standards to lightning component Framework
- ★ Quick Component Development Because developer has to use only HTML ,CSS and Javascript
- ★ Code performance using web standards boosts performance because more Features are executed natively by the browser instead by a js framework

More On LWC
Most of the code we write is standard Js and HTML So:

- ★ We can find solution in common place on the web
- ★ LWC is lightweight & delivers Exceptional performance
- ★ We can utilize full encapsulations so component become more versatile

Files to create Component

- ★ **You just have to create three simple files**

- ★ **HTML**
  - Provide the structure of component
- ★ **JS**
  - Define the core business logic & event handling
- ★ **CSS**
  - Provide look & feel to component

Aura Vs LWC Component bundles

Resource	**Aura File**	**LWC File**
Markup	test.cmp	test.html
controller	testController.js	test.js
Helper	testHelper.js	test.js
Renderer	testRenderer.js	test.js
Css	test.css	test.css

Documentation	test.auradoc	NA
Design	test.design	test.js-meta.xml
Svg	test.Svg	Include or HTML or a static Resource

HTML File :

```
<template>

    <input value = {message}></input>

    {message}

</template>
```

JS File:

```
Import {LightningElement} From 'LWC'

Export default Class Digital Extends LightningElement
{
   message='Hello World';
 }
```

CSS.File:

```
 Input
 {
 color:green;
 }
```

Component Configuration File

```
<?xml version="1.0" encoding="UTF-8"?>
<Lightning ComponentBundle xmlns="http://soap.sforce.com/2006/04/metadata">
<apiVersion>55.0</apiVersion>
<isExposed>true</isExposed>
<targets>
<target>lightning_HomePage</target>
<target>lightning_Record Page</target>
```

<target>lightning_AppPage</target>
</targets>
</Lightning ComponentBundle>

The LWC Module:

- ★ Lightning Web Components uses modules (built-in modules
- ★ were introduced in ECMAScript 6) to bundle core
- ★ functionality and make it accessible to the JavaScript in your
- ★ component file.
- ★ The core module for Lightning web components is lwc.
- ★ Begin the module with the import statement and specify the
- ★ functionality of the module that your component uses.
- ★ The import statement indicates the JavaScript uses the
- ★ Lightning Element functionality from the lwc module.

Important to Remember

- ★ Aura Components can contain Lightning Web Components.
- ★ Lightning Web Components cannot contain Aura Components.

Understand Lightning Web Component Bundle

LWC 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

LWC Folder and File Rules

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)

Camelcase & Kebab case:

- ★ 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>.

Camelcase & Kebab case:

- ★ However, the Salesforce platform doesn't allow hyphens in the component folder or file names. What if a component's Does the name have 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>.

Lightning Web Component & HTML File :

HTML File

- ★ 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.**

```
<!-- myComponent.html -->
<template>
<!-- Replace comment with component HTML -->
</template>
```

HTML File:

★ When a component renders, the <template> tag is replaced with the name of the component, <namespace-component-name>.

★ For example, in the browser console, myComponent renders as <c-my-component>, where c is the default namespace.

★ 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.

Lightning Web Component & JavaScript File:

JavaScript File:

★ 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.

JavaScript File

- ★ 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.

JavaScript File

```
import { Lightning Element } from 'lwc';
export default class MyComponent extends Lightning Element
 {
//Write your code here
}
```

- ★ The core module in Lightning Web Components is lwc. The import statement imports Lightning Element from the lwc module.
- ★ Lightning Element is a custom wrapper of the standard HTML element.
- ★ Extend Lightning Element to create a JavaScript class for a Lightning web component. You can't extend any other class to create a Lightning web Component.

The JavaScript file can contain:

- ★ The component's public API via public properties and methods
- ★ annotated with @api.
- ★ Fields
- ★ Event handlers

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.

- ★ Every component must have a configuration file.
  The configuration file follows the naming convention <component> js-meta.xml, such as myComponent.js-meta.xml.

- ★ Include the configuration file in your component's project folder, and push it to your org along with the other component files.

Configuration File - Example1

```
<?xml version="1.0" encoding="UTF-8"7>
<LightningComponentBundle xmlns="http://soap.sforce.com/2006/04/metadata">
<apiVersion51.0</apiVersioncisExposed>true<lisExposed>
<targets>
<target>lightning_HomePage</target>
<target>lightning_RecodPage</target>
<target>lightning_AppPage</target>
</targets>
</LightningComponentBundle>
```

Configuration File Example 2

```
<?xml version="1.0" encoding="UTF-8"7>
```

```
<LightningComponentBundle xmlns="http://soap.sforce.com/2006/04/metadata">
<apiVersion>51.0</apiVersion>
<isExposed>true</isExposed>
<targets>
<target>lightning HomePage<target>
<target>lightning Record Page</target>
<target>lightning AppPage</target>
</targets>
<targetConfigs>
<targetConfig targets="lightning RecordPage>
<objects>
<object>Account</object>
</objects>
</targetConfig>
</targetConfigs>
</Lightning ComponentBundle>
```

CSS File:

- ★ 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.

```
/*myComponent.css*/
    h1{
    font-size: x-large;
    }
```

SVG Icon

- ★ A component can include an SVG resource to use as a custom icon in Lightning App Builder and Experience Builder.

★ 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.

Additional JavaScript Files:

★ A component's folder can contain other JavaScript files. Use these JavaScript files to structure code in Ul components, and share code from service components.

★ These additional JavaScript files must be ES6 modules and must have names that are unique within the component's folder.

myComponent:

- ★ EmyComponent.html
- ★ myGomponentjs
- ★ EmyComponentjs-meta.xml
- ★ myComponent.css
- ★ EsomeUtils.js
- ★ LsomeMore Utils.js

★ Export functions and variables in the JavaScript files so they can be imported by thecomponent's main JavaScript file.

Import JavaScript Functions from current Folder:

```
import{ LightningElement} from "wc;
import {Some Function} from '/someUtils';
export default class MyComponent extends LightningElement {
//Write your code here
}
```

Lightning Web Component Namespace:

Component Namespace:

★ To reference your own components, always code with the default namespace, c. Use c regardless of where the code is running: in an org with or without namespace, in a managed or unmanaged package.

```
<template>
<lightning-card title="CompositionBasics" icon-name="custom:custom57">
<div class="slds-m-around_medium”>
<c-contact-tile contact={contact}><lc-contact-tile>
</div>
<lightning-card>
</template>
```

Component Namespace:

```
import {LightningElement } from "Lwc';
import {SomeFunction } from 'c/commonUtils';
export default class Example extends LightningElement {
//Write your code here
}
```

Create First Lightning Web Component & Deploy to Salesforce Org

FirstLWC.html File:

```
<template>
    <div>
        <div>Name: {name}</div>
        <div>Company: {company}</div>
        <div>Designation : {designation} </div>
        <div> Salary: {salařy}</div>
    </div>
</template>
```

FirstLWC.js File:

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

export default class FirstLWC extends LightningElement
    {
    name = ‘Digital';
    Company = 'DigitalLync';
    designation = 'Developer’;
    salary= '$400000’;
    }
```

FirstLWC.js-meta.xml File:

```
<?xml version="1.0" encoding="UTF-8"?>
<LightningComponent Bundle xmlns="http: //soap. sforce.com/2006/04/metadata">
        <apiVersion>55.0</apVersion>
        <isExposed>true</isExposed>
    <targets>
        <target>lightning_ Home Page</target>
        <target>lightning_Record Page</target>
    </targets>
</LightningComponentBundle>
```

Deploy Lightning Web Component :

Click the SFDC: Deploy to Source org

Migrating Markup, CSS, JS | Aura to Lightning Web Component

Migrate Markup:

Aura

- ★ Aura component contains markup in .cmp file.
- ★ File starts with <aura:component>
- ★ Above tag can contain HTML and aura-specific tags.

LWC

- ★ It contains markup in .html file
- ★ File starts with <template>
- ★ Tag can contain HTML and directives for dynamic Content.

Attributes vs Properties

- ★ Aura contains attributes in the markup file. To create attribute in aura we use <aura:attribute> tag.
  In LWC properties are created in JavaScript files. In To create properties we use @api or@track.

Attributes in Aura:

- ★ <aura:attribute name="message" type="String"/ >
- ★ <aura:attribute name="employee" type="employee_c"/>

Properties in LWC:

```
Import{ Lightning Element, api } from 'lwc';

export default class MyCmp extends LightningElement

{

@api message;

Employee;

}
```

Aura Expression Vs HIML Expression:

- ★ **In Aura Expressions are written like:**

  {!v.total} Items

- ★ **.In LWC Expressions aře written like:**

  {total items}Items

Aura Expression:

```
<aura:component>
<aura:attribute name="pages" type="integer"/>
<aura:attribute name="total" type="integer"/>
<aura:handler name="init" value="{!this}" action="{lc.dolnity”/>
{lv.total} items
Total Pages: {!v.pages}
</aura:component>
```

Aura JSController:

```
({
dolnit: function(component, event, helper) {
component,set("v.total", 20);
component.set("v.pages", 100);
}
})
```

Aura Expression:

```
<aura:component>
```

```
<aura:attribute name="pages" type="integer"/>
<aura:attribute name="total" type="integer"/>
<aura:handler name="init" value="{!this}" action="{lc.doInit}"/>
{!v.total} items
Total Pages: {!v.pages}
</aura:component>
```

Aura JSController:

```
({
doInit: function(component, event, helper) {
component.set("v.total", 20);
component.set("V.pages", 100);
}
})
```

LWC HTML Expression:

```
<template>
{total} items
Total Pages: {totalPages}
</template>
```

LWC JavaScript:

```
import { LightningElement, api } from 'lwc';
export default class Paginator extends LightningElement
{
```

```
@api page Size = 10;
@api total = 200;
get totalPages() {
return Math.ceil(this.total / this.pageSize);
}
}
```

Aura Conditionals Vs HTML Conditionals:

★ Aura

<aura:if>

★ LWC

If:true or if:false

Aura Conditionals:

```
<aura:if isTrue="{!v.employee.Office_cl">
<lightning:.recordForm recordId="{!v.employee.Office_c">
objectApiName="Office_c"
fields="{!v.officeFields}" columns="2"/>
</aura:if>
```

HTML Conditionals:

```
<template if:true={employee.data}>
```

```
<lightning-record-form object-api-name="Office_c"
record-id={officeId) fields={officeFields}
columns="2">
</lightning-record-form>
</template>
```

Aura Iterations Vs HTML Iterations

- ★ Aura

  <aura:iteration>

- ★ LWC

  for:each

Aura Iterations:

```
<aura:iteration items="{!v.employees}" var="employee">
<c:EmployeeTile employee="{#employee}"/>
</aura:iteration>
```

HTML Iterations:

```
<template for:each={employees.data.records}
for:item="employee">
<c-employee-tile employee=(employee) key={employee.ld}
onselected={handleEmployeeSelected)></c-employee-tile
</template>
```

Aura Init Vs Lifecycle Hooks:

- ★ Aura

init event handler

★ LWC

connected Callback( ) method

Aura Init

```
<aura:handler name="init" value="{!this}"
action="{!c.onInit"} />
```

connected Callback():

```
export default class MyCmp extends LightningElement
{
connectedCallback()
{
// initialize component
}
}
```

Aura Base Component:

```
<aura:component>
<lightning:formatted Number value="5000" style="currency"
currencyCode="USD" />
</aura:component>
```

LWC Base Component:

```
<template>
```

```
<lightning-formatted-number value="5000" style="currency"
currency-code="USD">
</lightning-formatted-number>
</template>
```

Aura CSS:

```
THIS.div{
padding: 0;
margin: 0;
}
```

Aura JavaScript:

```
({
previousPage: function(component) {
var pageChangeEvent = component.getEvent("pagePrevious")
pageChangeEvent.fire()
},
nextPage: function(component) {
var pageChange Event = component.getEvent("page Next");
pageChangeEvent.fire();
} })
```

LWC JavaScript:

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

```
previousHandler(){
this.dispatchEvent(new CustomEvent (previous'));
}
get isFirstPage()
{
return this.pageNumber === 1;
}
}
```

Use Third-party Java Script Libraries:

★ Aura

```
<ltng:require scripts="{!$Resource.resourceName}"
afterScriptsLoaded="{lc.afterScriptsLoaded}" />
```

★ LWC

```
import resourceName from
@salesforce/resourceUrl/resourceName';
```

Decorators - @api, @track @wire

Decorators:

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

@api

@track

@wire

@api:

- ★ Marks a field as public.
- ★ HTML markup can access the component's public properties.
- ★ All public properties are reactive.
- ★ Reactive means the framework observe the property for change.
- ★ When property changes value then the framework reacts and renders the component.
- ★ Fields and property are interchangeable terms.

Example:

```
import { LightningElement, api } from 'lwc';
export default class MyCmp extends LightningElement{
@api message; //public property
Value;              //private property
```

@track:

- ★ Observe changes to the properties of an object or to the elements of an array.
- ★ Framework renders the component when changes occur.

@wire:

- ★ It provides a way to get and bind data from a Salesforce org.
- ★ The wire service provisions an immutable stream of data to the component.
- ★ Each value in the stream is a newer version of the value that precedes it.
- ★ Objects passed to a component are read-only.
- ★ To mutate the data, a component should make a shallow copy of the objects it wants to mutate.

Wire Service Syntax :

import(adapterId) from 'adapterModule';

@wire(adapterId, adapterConfig)

propertyOrFunction;

- ★ **adapterId (identifier)--**The identifier of the wire adapter.
- ★ **adapterModule (String)**-The identifier of the module that contains the wire adapter function, in the format namespace/moduleName.
- ★ **adapterConfig (Object)-A** configuration object specific to the wire adapterConfiguration object property values can be either strings or references to objects and fields imported from@salesforce/schema.
- ★ **propertyOrFunction-A** private property or function that receives the stream of data from the wire service. Ifa property is decorated with@wire, the results are returned to

the property's data property or error property. Ifa function is decorated with @wire, the results are returned in an object with a data property and an error property.

Example - 1 :

```
import { LightningElement, api, wire ) from 'lwc;
import {getRecord} from lightning/uiRecordApi;
import ACCOUNT_NAME_FIELD from
'@salesforce/schema/Account.Name;
export default class Record extends LightningElement {
@api recorded;
@wire(getRecord, { recordld: '$recordid', fields:
[ACCOUNT_NAME_FIELD] })
Record;
}
```

Example - 2:

```
public with sharing class ContactConțroller
{
@AuraEnabled(cacheable=true)
```

```
public static List<Contact> getContacts(String accId)
{
return [
SELECT Account, Id, FirstName, LastName, Title, Phone, Email
FROM Contact
WHERE Accountid :accId
WITH SECURITY_ENFORCED
];
}
}
```

```
import {LightningElement, wire, api } from 'lwc';
import getContacts from '@salesforce/apex/ContactController.getContacts';
import { refreshApex} from '@salestorce/apex;
import{ updateRecord } from "lightning/uiRecordApi';
export default class Example extends LightningElement {
@api recorded;
@wire(getContacts, { accId: '$recordId' })
Contacts;
}
```

Use @api Decorator | Private Vs Public Property in Lightning Web Component

Private

(Vs)

Public Property

Private

JavaScript.file:

```
import {LightningElement } from "lwc';
export default class PriPubDemo extends LightningElement
{
message = "Private Property';
recordId;
}
```

Html.file:

```
<template>
    <div> Message : {message}</div>
    <div> Record Id : {recordId}</div>
</template>
```

Js-meta.xml file:

```
<?xml version="1.0" encoding ="UTF-8"?>
<LightningComponentBundle xmls='http://soap.sforce.com/2006/04/metadata">
 <apiVersion>55.0</apiVersion>
<isExposed>true</isExposed>
```

```
    <targets>
        <target>Lightning_RecordPage</target>
    </targets>
</LightningComponentBundle>
```

Public Property

JavaScript.file:

```
import {LightningElement, api } from "lwc";
export default class PriPubDemo extends LightningElement
{
message = "public  Property";
@api recordId;
}
```

Html.file:

```
<template>
    <div> Message : {message}</div>
    <div> Record Id : {recordId}</div>
</template>
```

Js-meta.xml file:

```
<?xml version="1.0" encoding ="UTF-8"?>

<LightningComponentBundle xmls='http://soap.sforce.com/2006/04/metadata">

 <apiVersion>55.0</apiVersion>

<isExposed>true</isExposed>

        <targets>

                <target>Lightning_RecordPage</target>

        </targets>

</LightningComponentBundle>
```

Use @wire decorator | Lightning Data Service Wire Adapter

Import references to Fields:

WireDemo.htmlfile:

```
        <template>
         <div class="slds-grid">
           <div class="slds-col slds-size_1-of-2">
                 <lightning-formatted-text value={name} class="slds-text-heading_large">
           </lightning-formatted-text>
         </div>
         <div class="slds-col slds-size_2-of-2">
                 <lightning-formatted-text  value={phone} class="slds-text-heading_large">
          </lightning-formatted-text>
         </div>
         </div>
         </template>
```

Js.File:

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

```
import{ getRecord,getFieldValue} from 'lightning/uiRecordApi';

import Name_FIELD From '@salesforce/schema/Account.Name';
Import Phone-FIELD From '@salesforce/schema/Account.Phone';
const FIELDS = [‘Account.Name’,’Account.Phone’,]

export default class WireADapterDemo extends LightningElement
{
 @api recordId;

 @wire(getRecord,{recordId:'$recordId', fields: })
 record;
get name(){
     return this.record.data ? getFieldValue(this.record.data,’Account.Name’): ' ';

  }

 get Phone(){
         return this.record.data ? getFieldValue(this.record.data,‘Account.Phone’) : ' ';

    }
}
```

Use Object’s Field Through String:

```
import {LightningElement,api,wire} from 'lwc';
import{ getRecord,getFieldValue} from 'lightning/uiRecordApi';

//import Name_FIELD From '@salesforce/schema/Account.Name';
//Import Phone-FIELD From '@salesforce/schema/Account.Phone';
const FIELDS = [‘Account.Name’,’Account.Phone’,]

export default class WireADapterDemo extends LightningElement
{
 @api recordId;
```

```
 @wire(getRecord,{recordId:'$recordId', fields:})
 record;
get name(){
    // return this.record.data ? getFieldValue(this.record.data,'Account.Name'): ' ';
        return  this.record.data.field.Name.value;
  }

 get Phone(){
        // return this.record.data ? getFieldValue(this.record.data,'Account.Phone') : ' ';
        return  this.record.data.data.field.Name.value;
   }
}
```

Import Object & Fields Reference Vs Use Object & Fields as String

Import References to Fields

Vs

Use Field Through String

```
import {LightningElement,api,wire} from 'lwc';
import{ getRecord,getFieldValue} from 'lightning/uiRecordApi';

import Name_FIELD From '@salesforce/schema/Account.Name';
Import Phone-FIELD From '@salesforce/schema/Account.Phone';
const FIELDS = ['Account.Name','Account.Phone',]

export default class WireADapterDemo extends LightningElement
{
 @api recordId;
```

```
 @wire(getRecord,{recordId:'$recordId', fields:['Account.Name','Account.Phone']})
 record;

get name(){
     return this.record.data ? getFieldValue(this.record.data,'Account.Name'): ' ';

  }

 get Phone(){
         return this.record.data ? getFieldValue(this.record.data,'Account.Phone') : ' ';

    }
}
```

Why to Import References to Salesforce Objects and Fields?

- ★ When you use a wire adapter in a lightning/ui"Api module, we strongly recommend importing references to objects and Fields
- ★ Salesforce verifies that the objects and fields exist, prevents objects and fields from being deleted, and cascades any renamed objects and fields into your component's source Code.
- ★ It also ensures that dependent objects and fields are included in change sets and packages.
- ★ Importing references to objects and fields ensures that your code works, even when object and field names Change
- ★ If a component isn't aware of which object it's using, use strings instead of imported references. Use getObjectinfo to return the object's fields.
- ★ All wire adapters in the lightning/ui"Api modules respect

object CRUD rules, field-level security, and sharing.

★ If a user doesn't have access to a field, it isn't included in the response.

★ import POSITION_OBJECT from '@salesforce/schema/Position_c;
★ import ACcOUNT_OBJECT from '@salesforce/schema/Account';
★ import POSITION_LEVEL_FIELD from '@salesforce/schema/Position_c.Level_c;
★ import ACCOUNT_NAME_FIELD from '@salesforce/schemalAccount.Name';
★ import POSITION_HIRINGMANAGER_NAME_FIELD from @salesforce/schema/Position_c.HiringManager _r.Name_c;
★ import ACCoUNT_OWNER_NAME FIELD from @salesforce/schema/Account.Owner.Name';

Use of @wire decorator | Wire Apex Class Method in Lightning WebComponent

Apex Class:

```
  public with sharing class ContactController
{
 @AuraEnabled(cacheable=true)
public static List<Contact> getContacts(String accId)
{
 try{
    return[SELECT AccountId,Id,FirstName FROM Contact Where AccountId =: accId
With SECURITY_ENFORCED];
}
 catch (Exception e)
```

```
{
  throw new AuraHandledException(e.getMessage());
}
}
}
```

Apex.jsFile

```
import { LightningElement,api,wire } from 'LWC';
import getContacts from '@salesforce/apex/ContactController.getContacts';
export default class wireApexDemo extends LightningElement
{
@api recordId;
@wire(getContacts, {accId: '$recordId'})
contacts;
}
```

Apex.htmlfile:

```
<template>
<lightning-card title="Related Contacts" icon-name="custom:custom14">
 <ul class="slds-m-around_medium">
  <template for:each = {contacts.data} for:item="contact">

      <li key={contact.Id}>
          {contact.FirstName}
       </li>
   </template>
 </ul>
</Lightning-card>
</template>
```

Apex.js-meta.xml file:

```
<?xml version="1.0" encoding="UTF-8"?>
        LightningComponentBundlexmlns="http://soap.sforce.com/2006/04/metadata">
            <apiversion>55.0</apiversion>
                <isExposed>true</isExposed>
                    <targets>
                        <target>Lightning_RecordPage</target>
                    </targets>
                </LightningComponentBundle>
```

Decorate Property or Function with @wire Decorator

Wire Service Syntax:

```
import { adapterId ) from 'adapterModule';
@wire(adapterId, adapterConfig9)
```

propertyOrFunction;

- ★ **adapterId (identifier)**-The identifier of the wire adapter.
- ★ **adapterModule (String)**-The identifier of the module that contains the wire adapter
  function, in the format namespace/moduleName.
- ★ **adapterConfig (Object)-A** configuration object specific to the wire adapter. Configuration object property values can be either strings or references to objects and
  fields imported from@salesforce/schema.
- ★ **propertyOrFunction–A** private property or function that receives the stream of data
  from the wire service. If a property is decorated with@wire, the results are returned to

the property's data property or error property. If a function is decorated with @wire, the
results are returned in an object with a data property and an error property.

Apex Class:

```
  public with sharing class ContactController
{
 @AuraEnabled(cacheable=true)
public static List<Contact> getContacts(String accId)
{
 try{
    return[SELECT AccountId,Id,FirstName FROM Contact Where AccountId =: accId
With SECURITY_ENFORCED];
}
 catch (Exception e)
{
  throw new AuraHandledException(e.getMessage());
}
}
}
```

Apex Js.file:

```
import {LightningElement,api,wire} from 'LWC';
import getContacts from '@salesforce/apex/ContactController.getContacts';
import{getRecord} from 'lightning/uiRecordApi';

 export default class wireApexDemo extends LightningElement
{
@api recordId;

@wire{getRecord, {recordId: '$recordId', fields:'Account.Name'})
record;
@wire{getContacts,{accId: '$recordId'})
wiredContact({error,Data})
{
if(data){
this.Contacts = data;
this.error = undefined;
}
else if(error)
```

```
{
 this.error = error;
this.contacts = undefined;
}
}
get name(){ return this.record.data.field.Name.value;}}
```

Apex.htmlfile:

```
  <template>
<lightning-card title="Related Contacts" icon-name="custom:custom14">
 <ul class="slds-m-around_medium'>
  <template if:true={record.data}">
    <lightning-formatted-text value={name} class="slds-text-heading_large">
 </lightning-formatted-text>
</template>
 <template for:each={contacts} for:item="contact">
  <li key={contact.Id}>
     {contact.FirstName} - {contact.LastName}
</li>
</template>
<template if:true={error}>
</template>
</ul></lightning-card>
```

Use of Property | HTML Template Bind Data

Html.file:

```
<template>
   <p>Hello,{greeting}!</p>
     <lightning-input label="Name" value = {greeting}
           onchange={handleChange}>
```

```
        </lightning - input>
    </template>
```

Js.file:

```
import { LightningElement } from 'LWC';
export default class DataBinding extends LightningElement
{
greeting = 'Digital';
handleChange(event)
{
 this.greeting = event.target.value;
}
}
```

Use of Getter | HTML Template Data Binding | Lightning Web Component

Html.file:

```
<template>
  <div class="slds-m-around_medium">
    <lightning-input name='fname' label="First Name"
     onchange={handleChange}></lightning-input>
      <lightning-input name='lname' label="Last Name"
     onchange={handleChange}></lightning-input>
    <p class="slds-m-top_medium"> Uppercased Full Name : {uppercase Full
     Name}</p>
  </div>
</template>
```

Js.file:

```
import {LightningElement} from 'lwc'
export default class DataBinding extends LightningElement
{
firstName = '';
lastName = '';
handleChange(event){
const field = event.target.name;
 if(field === 'fname')
{
  this.firstName = event.target.value;
} else if(field ==='lname')
{
 this.lastname = event.target.value;
}
}
get uppercase Full Name()
{
  return ${this.firstName} ${this.lastName}.toUpperCase();
}
}
```

js-meta.xml file:

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

</LightningComponentBundle>

Use querySelector to Fetch Data | HTML Template Data Binding

Html.file:

```
<template>
<p> Hello, {greeting}!</p>
<lightning-input label="Name" value={greeting}></lightning-input>
<br/>
<lightning-button label="click ME" onclick={handleClick}
variant="brand"></lightning-button>
</template>
```

js.file:

```
import {LightningElement} from 'lwc';
export default class DataBinding extends LightningElement
{
greeting = 'digital';
handleClick(event)
{
   this.greeting = this.template.querySelector("Lightning-input").value;
}
}
```

Use querySelectorAll to fetch data | HTML Template Data Binding

★ **HTML TemplatesData Binding**

★ **Fetch Data UsingquerySelectorAll**

DataBinding.html:

```
<template>
```

```
 <lightning-card>
   <p>Hello,{firstName} {lastName}!</p>
     <lightning-input name="fname" label="First Name"
value={firstName}></lightning-input>
     <br/>
       <lightning-input name="lname" label="Last Name"
value={lastName}></lightning-input>
     <br/>
<lightning-button label="click Me" onclick={handeClick}
variant="brand"></lightning-button>
</lightning-card>
</template>
```

DataBinding.js:

```
import { LightningElement} from 'lwc';
export default class DataBinding extends LightningElement
{
 firstName='';
 lastName='';

     handleClick(event)
{
   var input = this.tempalte.querySelectorAll("lightning-input");
  input.forEach(function(element)
{
 if(element.name =='fname')
   this.firstName = element.value;
else if(element.name =='lname')
this.lastName = element.value;
}, this);
}
```

DataBinding.js-meta.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<LightningComponentBundlexmls="http://soap.sforce.com/2006/04/metdata">
 <apiVersion>55.0</apiVersion>
 <isExposed>true</isExposed>
 <targets>
   <target>lightning_HomePage</target>
 </targets>
 </LightningComponentBundle>
```

Use of Getter & Setter | Lightning Web Component

Child.html:

```
<template>
    {uppercaseItem}
</template>
```

Child.jsfile:

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

export default class Child extends LightningElement
{
  uppercaseItemName = 'default value';

@api

  get itemName()
{
 return this.uppercaseItemName;
}
set itemName(value)
{
this.uppercaseItemName = value.toUpperCase();
}
}
```

Parent.html:

```
<template>
    <lightning-card>
          <div class="slds-p-top_large">
             <c-child class="slds-text-heading_medium" item-name="Digital">
              </c-child>
         </div>
         <div class="slds-p-top_large">
              <c-child class="slds-text-heading_medium" item-name="Kona">
              </c-child>
        </div>
  </lightning-card>
</template>
```

Parent.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>
                    <targetlightning_HomePage</target
                </targets.
   </LightningComponentBundle>
```

Message Passing through Parent to Child Component

Message Passing
Between
Components

Child.html:

```
<template>

        Hello, {firstName}!

</template>
```

Child.js:

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

export default class Child extends LightningElement
{
  @api

 firstName = 'Digital';
}
```

Parent.html:

```
  <template>
        <lightning-card>
              <div class="slds-p-top_large">
                    <c-child class="slds-text-heading_medium" first-name="Arjun">
                    </c-child>
              </div>
  <div class="slds-p-top_large">
        <c-child class="slds-text-heading_medium" first-name="kalyan">
        </c-child>
  </div>

<div class="slds-p-top_large">
      <c-child class="slds-text-heading_medium" first-name="Kona">
      </c-child>
</div>
</lightning-card>
</template>
```

Parent.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>
                    <targetlightning_HomePage</target
                </targets.
</LightningComponentBundle>
```

Conditional Rendering in HTML | Apply if-else in LWC