LIGHTNING WEB COMPONENTS CHEAT SHEET 2ND EDITION – BY SANTANU BORAL

OVERVIEW

Lightning Web Components (LWC) is a framework for creating modern UI on web, mobile apps and digital experience on Salesforce platform. Aura Components and LWC both coexist and interoperate on the page. It leverages web standards and delivers high performance. It is treated as Lighting Components.

FEATURES AND ADVANTAGES

- LWC leverages W3C Web Standards
- Modern JavaScript (ES6 standard)
- Simplify data access through @wire
- Build resilient Apps with Shadow DOM
- Supports API versioning for custom cmps
- Is an open source

Lightning Web Components

Enhanced Security
Intelligent Caching
UI Components
Data Services
UI Services
Templates
API Versioning
Customizable Base Lightning

Web Standards leveraging

Rendering Optimization
Standard Elements
Component Model
Custom Elements
Standard Events
Core Language
Rendering

GETTING STARTED

To create and deploy LWC App follow this life cycle:

Create Salesforce DX Project
- In VS Code, press **Command + Shift P**, enter **sfdx** , and select **SFDX: Create Project**.
- Enter MyLWC as project name

Authorize an Org
- In VS Code, press **Command + Shift P**, enter **sfdx** , and select **SFDX: Authorize an Org**.
- Press Enter to accept Project Login URL option,
- Accept default alias

Create a LWC
- In VS Code, press **Command + Shift P**, enter **sfdx,** select **SFDX: Create Lightning Web Component**.
- Enter myHelloWorld as name of the component.
- Enter to accept the default force-app/main/default/lwc folder.

Deploy to Org
- In VS Code, right-click default folder under force-app/main
- Click **SFDX: Deploy Source to Org**

Add Component to App
- In VS Code, press **Command + Shift P**, enter **sfdx,** select **SFDX: Open Default Org**
- Open the page (Edit Page) where component to be added and drag myHelloWorld component and save.

COMPONENT BUNDLES AND RULES

To create a component, first create folder and it consists of following components where first three are mandatory.

End UI (HTML) | Controller (.js) | Configuration (.js-meta.xml) | Style (.css)

Resource (.svg) | Additional .js | __tests__

HTML FILE

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

UI components should have HTML file, service components don't need.

When component renders <template> tag will be replaced by name of component <namespace-component-name>, like myComponent renders as <c-my-component> where c is default namespace.

CONTROLLER

```
import { LightningElement } from 'lwc';
export default class MyComponent extends LightningElement {
//component code here
}
```

If component renders UI, then JavaScript file defines HTML element. It contains public API via public properties and methods with @api, Fields, Event handlers.

CONFIGURATION

```
<?xml version="1.0" encoding="UTF-8"?>
<LightningComponentBundle
xmlns="http://soap.sforce.com/2006/04/metadata">
    <apiVersion>54.0</apiVersion>
    <isExposed>false</isExposed>
</LightningComponentBundle>
```

Configuration file defines metadata values, including design configuration for Lightning App Builder and Community Builder. Include the configuration file in your component's project folder, and push it to your org along with the other component files.

CSS

```
.title {
    font-weight: strong;
}
```

Use standard css syntax.

SVG

Use SVG resource for custom icon in Lightning App Builder and Community Builder. To include that, 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.

COMPONENT TEST FILES

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

Jest runs JavaScript files in the __tests__ directory. Test files must have names that end in .js, and it is recommended that tests end in .test.js. We can have a single test file or multifile test file in an organized way. Test files can be placed in sub folders.

REACTIVITY

Reactive Properties: if value changes, component renders. It can be either private or public. When component renders all expressions in the template as re-evaluated.

Primitive fields—like booleans, numbers, and strings—are reactive. LWC tracks field value changes in a shallow fashion. Changes are detected when a new value is assigned to the field by comparing the value identity using ===.

@api	For exposing public property, this is reactive. If value of a public property used in template changes, the component rerenders, hence expressions are re-evaluated, renderedCallback() executes
@track	To observe changes to the properties of an object or to the elements of an array, use @track. Though framework doesn't observe mutations made to complex objects, such as objects inheriting from Object, class instances, Date, Set, or Map.
@wire	To get and bind data.
setAttribute()	For reflecting JavaScript properties to HTML attributes

Example : Below example shows how @api and @track haven been used.

```
<!-- myComponent.html -->
<template>
        <div class="view">
                <label>{title}</label>
        </div>
   <p>{fullName}</p>
   <lightning-button label="Change Name"
                onclick={handleClick}>
        </lightning-button>
</template>
```

```
// myComponent.js
import { LightningElement, track, api } from 'lwc';
export default class myComponent extends LightningElement {
   @api title = 'Sample Example';
   @track fullName = { firstName : '', lastName : ''};
    handleClick(){
        this.fullName.firstName = 'John';  //assign firstName
    }
}
```

COMPOSITION

Owner - owns template

- Set public properties on composed components
- Call methods on composed components
- Listen to any events fired by composed components

Container - contained within owner & contains other components. Less powerful than owner

- Read but no change, public properties contained in container
- Call methods on composed components
- Listen for some, but not all, events bubbled up by its components

Parent & Child - Parent can be owner or container, can contain child components

- Same principle as stated for Owner or Container

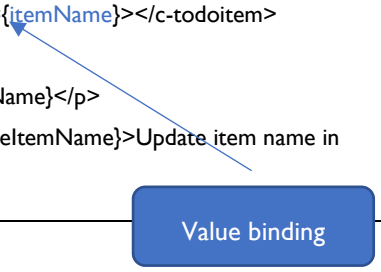
Setting property to Children

- To communicate down to component hierarchy owner can set a property.
- Set a primitive value by using @api decorator
- Set a non-primitive value like object or array that is passed to component is readonly. To mutate the data, make a swallow copy of the objects we want to mutate.

Data Flow

- Data should flow from parent to child.
- Use primitive data types for properties instead of using object data types. Standard HTML elements accept only primitive.

Parent	Child
<!-- todoapp.html --> <template> <c-todowrapper> <c-todoitem item-name={itemName}></c-todoitem> </c-todowrapper> item in todoapp: {itemName} <button onclick={updateItemName}>Update item name in todoapp</button> </template>	**<!-- c-todoitem.html -->** <template> item in todoitem: {itemName} <button onclick={updateItemName}>Update item name in todoitem</button> </template>
// c-todoapp.js import { LightningElement, track } from 'lwc'; export default class Todoapp extends LightningElement { @track itemName = "Milk"; updateItemName() { this.itemName = "updated item name in todoapp"; } }	**// c-todoitem.js** import { LightningElement, api } from 'lwc'; export default class Todoitem extends LightningElement { @api itemName; // This code won't update itemName because: // 1) You can update public properties only at component construction time. // 2) Property values passed from owner components are read-only. updateItemName() { this.itemName = "updated item name in todoitem"; } }



Call methods on Children

Owner and parent component can call JavaScript methods on Child components.

Parent	Child
<!-- methodCaller.html --> <template> <c-video-player video-url={video}></c-video-player> <button onclick={handlePlay}>Play</button> </template>	**<!-- videoPlayer.html -->** <template> <video autoplay> <source src={videoUrl} type={videoType} /> </video> </template>
// methodCaller.js import { LightningElement } from 'lwc'; export default class MethodCaller extends LightningElement { video = "https://www.w3schools.com/tags/movie.mp4"; handlePlay() { this.template.querySelector('c-video-player').play(); } } /*	**// videoPlayer.js** import { LightningElement, api } from 'lwc'; export default class VideoPlayer extends LightningElement { @api videoUrl; @api play() { const player = this.template.querySelector('video'); // the player might not be in the DOM just yet if (player) { player.play(); } }

The handlePlay() function in c-method-caller calls the play() method in the c-video-player element. this.template.querySelector('c-video-player') returns the c-video-player element in methodCaller.html. **/*	get videoType() { return 'video/' + this.videoUrl.split('.').pop(); } }

Spread Properties on Children

Pass set of properties using lwc:spread directive. It accepts one object. lwc:spread can used with event handler.

Parent	Child
<!-- methodCaller.html --> <template> <c-child lwc:spread={childProps}></c-child> </template>	**<!-- child.html -->** <template> Name: {name} Country : {country} </template>
// methodCaller.js import { LightningElement } from 'lwc'; export default class MethodCaller extends LightningElement { childProps = { name: "James Smith", country: "USA" }; }	**// child.js** import { LightningElement, api } from 'lwc'; export default class Child extends LightningElement { @api name; @api country; }

Access elements the Component owns	this.template.querySelector() - method is a standard DOM API that returns the first element that matches the selector. this.template.querySelectorAll() - method returns an array of DOM Elements. Refs – locate DOM without a selector. //assign it <template> </template> //use it export default class extends LightningElement { renderedCallback() { console.log(this.refs.myDiv); }}
Access Static Resource	import myResource from '@salesforce/resourceUrl/resourceReference';
Access Content Asset	import myContentAsset from "@salesforce/contentAssetUrl/contentAssetReference";
Access Internationalization Properties	import intPropertyName from @salesforce/i18n/internationalizationProperty
Access Labels	import labelName from '@salesforce/label/labelReference';
Access Current UserId	import Id from '@salesforce/user/Id';
Check Permissions	import hasPermission from "@salesforce/userPermission/PermissionName"; import hasPermission from "@salesforce/customPermission/PermissionName";
Access Client Form Factor	import formFactorPropertyName from "@salesforce/client/formFactor";

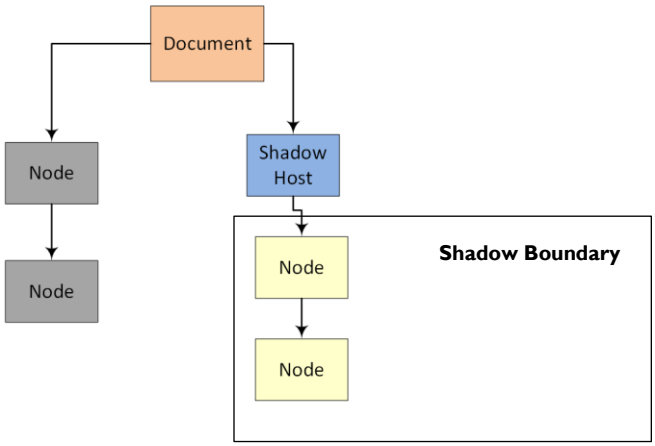
DYNAMIC COMPONENTS (DC)

- Config should include lightning__dynamicComponent
- HTML file will have <lwc:component lwc:is={componentConstructor}></lwc:component>
- Selecting dynamic component, use lwc:ref="myCmp" on above statement.
- It can include child elements, <lwc:component> first renders DC, then its children.
- Pass properties using @api
- Pass recordId like this: <lwc:component record-id={recordId} lwc:is={componentConstructor} >

WORKING WITH DOM

Shadow DOM	Light DOM	Synthetic Shadow DOM
Every element of each LWC are encapsulated in shadow tree. This part of DOM is hidden from the document it contains and hence called shadow tree. Shadow DOM is a web standard that encapsulates the elements of a component to keep styling and behavior consistent in any context.	When you use light DOM, your component lives outside the shadow DOM and avoids shadow DOM limitations. This approach eases third-party integrations and global styling.	It's a polyfill that mimics native shadow DOM behavior, Currently, Lightning Experience and Experience Builder sites use synthetic shadow by default.

Example of Shadow DOM



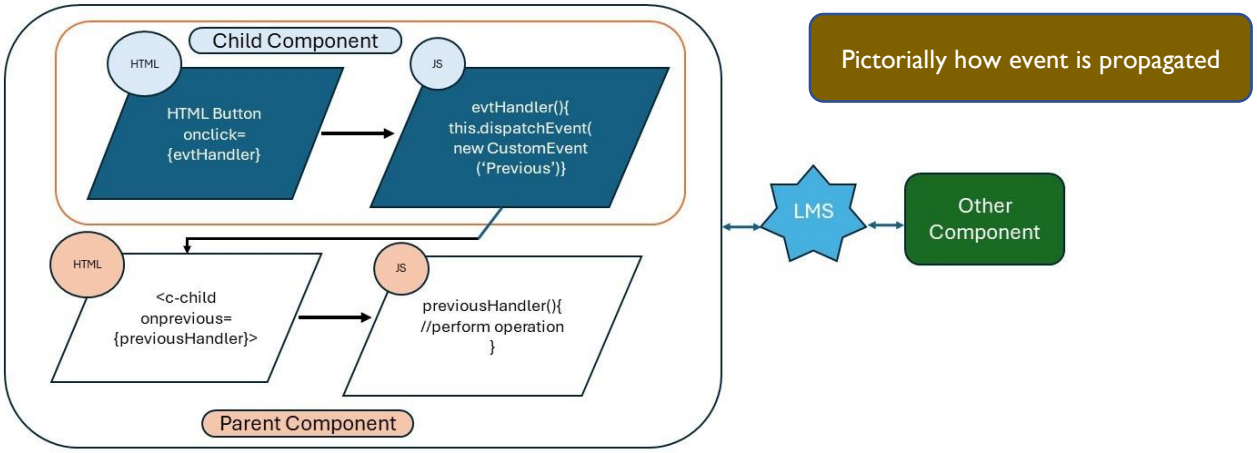

```
<c-todo-app>
  #shadow-root
    <div>
      <p>Your To Do List</p>
    </div>
    <c-todo-item>
      #shadow-root
        <div>
          <p>Go to the store</p>
        </div>
    </c-todo-item>
</c-todo-app>
```

Shadow DOM vs. Light DOM

Questions	Shadow DOM	Light DOM
How is it available?	LWC enforces shadow DOM on every component, encapsulates component's internal markup, making it inaccessible to programmatic code.	It lives outside shadow DOM and avoids shadow DOM limitations.
When to use it?	Recommended way to author components. It hides your component's internals so consumers can only use its public API.	Building a highly customizable UI, and we want complete control over web app's appearance. Third party integrations and global styling.
What about styling?	Requires CSS custom properties to override styles.	Easy to override styles
CSS	CSS styles defined in a parent component don't apply to a child component.	It enables styling from the root document to target a DOM node and style it.
How to use it?	<my-app> #shadow-root | <my-header> | Hello World | </my-header> | <my-footer> | #shadow-root | | Footer | </my-footer> </my-app> my-header uses light DOM and my-footer uses shadow DOM.	**lightDomApp.js** import { LightningElement } from "lwc"; export default class LightDomApp extends LightningElement { static renderMode = "light"; // the default is 'shadow' } **lightDomApp.html** <template lwc:render-mode="light"> </template>
How to access elements?	this.template.querySelector("div");	this.querySelector("div");

COMMUNICATE WITH EVENTS

Create an Event	CustomEvent() constructor in the js file
Dispatch an Event	EventTarget.dispatch(new CustomEvent('event name')); //in the js file
Pass data with an Event	const selectedEvent = new CustomEvent('selected', { detail: this.contact.Id }); //in the js file this.dispatchEvent(selectedEvent);
Attach Event listener declaratively	<template><c-child onnotification={handleNotification}></c-child></template>
Attach Event listener programmatically	For components within shadow boundary, use following snippet in js file: constructor() { super(); this.template.addEventListener('notification', this.handleNotification.bind(this)); } For components outside template: this.addEventListener('notification', this.handleNotification.bind(this));
Get reference to component who dispatched Event	Use Event.Target: handleChange(evt) { console.log('Current value of the input: ' + evt.target.value); }
Communicate across DOM	Using **Lightning Messaging Service (LMS)**: To communicate between components in a single Lightning page or across multiple pages, not restricted to a single page. Any component in a Lightning Experience application that listens for events on a message channel updates when it receives a message 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 and unsubscribe to a message using the subscribe() and unsubscribe() functions from the @salesforce/messageChannel module. Using **PubSub**: If containers that don't support Lightning Messaging Service, use the pubsub module.



WORKING WITH SALESFORCE DATA

Lightning Data Service (LDS)	To work with data and metadata for Salesforce records, use components, wire adapters and Javascript functions built on top of LDS. Records loaded are cached and shared across all components. Optimizes server calls by bulkifying and deduping requests.

	Base Lightning components: lightning-record-form, lightning-record-edit-form, or lightning-record-view-form. To create/update data use: lightning/uiRecordApi module, it respects CRUD access, FLS and sharing settings. Use GraphQL wire adapters which comes with client-side caching and data management capabilities
Using Base Components - Load a record:	`<template>` `<lightning-record-form` `record-id={recordId}` `object-api-name={objectApiName}` `fields={fields}>` `</lightning-record-form>` `</template>` `// myComponent.js` `import { LightningElement, api } from 'lwc';` `export default class MyComponent extends LightningElement {` `@api recordId;` `@api objectApiName;` `fields = ["AccountId", "Name", "Title"];` `}`
Using Base Components - Edit a record:	Use record-id and object-api-name and code will be almost as above. For, fields to appear: `<template>` `<lightning-record-form` `object-api-name={objectApiName}` `record-id={recordId}` `fields={fields}` `mode="edit">` `</lightning-record-form>` `</template>` `import Id from '@salesforce/user/Id';` `import { LightningElement, api } from 'lwc';` `import ACCOUNT_FIELD from '@salesforce/schema/Contact.AccountId';` `import NAME_FIELD from '@salesforce/schema/Contact.Name';` `export default class RecordFormStaticContact extends LightningElement {` `// Flexipage provides recordId and objectApiName` `@api recordId;` `@api objectApiName;` `fields = [ACCOUNT_FIELD, NAME_FIELD];` `}`
Using Base Components - Create a record:	`<template>` `<lightning-record-form` `object-api-name={accountObject}` `fields={myFields}` `onsuccess={handleAccountCreated}>` `</lightning-record-form>` `</template>` `import { LightningElement } from 'lwc';` `import ACCOUNT_OBJECT from '@salesforce/schema/Account';` `import NAME_FIELD from '@salesforce/schema/Account.Name';` `export default class AccountCreator extends LightningElement {` `accountObject = ACCOUNT_OBJECT;` `myFields = [NAME_FIELD];` `handleAccountCreated(){` `// Run code when account is created.` `}` `}`
Get Data With wire service – Get record data	This is reactive, which is built on LDS. Wire adapter is one of lightning/ui*Api modules. `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 recordId;
    @wire(getRecord, { recordId: '$recordId',
                       fields: [ACCOUNT_NAME_FIELD]})
    record;
}
```

Handle Errors

Using LDS wire adapters means it uses Async JavaScript

```
connectedCallback() {
  console.log("first");
  setTimeout(() => {
    try {
      throw new Error("some error");
      console.log("second");
    } catch(e) {
      console.log("error");
    }
  }, 1000);
  console.log("third");
}
```

```
@wire(getApexMethod)
wiredContacts({ error, data }) {

  if (data) {
    try {
      // handle result
    } catch(e) {
      // error when handling result
    }
  } else if (error) {
    // error with value provisioning
  }
}
```

Call Apex Method

```
import apexMethodName from '@salesforce/apex/Namespace.Classname.apexMethodReference';
public with sharing class ContactController {
    @AuraEnabled(cacheable=true)
    public static List<Contact> getContactList() {
        return [SELECT Id, Name FROM Contact
                WHERE Picture__c != null LIMIT 10];
    }
}
```

Wiring a Apex method:

```
import apexMethod from '@salesforce/apex/Namespace.Classname.apexMethod';
@wire(apexMethod, { apexMethodParams })
propertyOrFunction;
```

Call Apex Methods Imperatively:

-When cacheable=true omitted

-To control when invocation occurs

-Work with objects where UI API not supported

-Calling method that doesn't extend LightningElement

```
async handleLoad() {
  try {
      this.contacts = await getContactList();
      this.error = undefined;
    } catch (error) {
       this.contacts = undefined;
       this.error = error;
    } }
```

Wire a Apex method with dynamic parameter:

```
@wire(findContacts, { searchKey: '$searchKey' })
contacts;
```

Refresh cache using: notifyRecordUpdateAvailable(recordIds)

Import Objects and Fields from @salesforce/schema

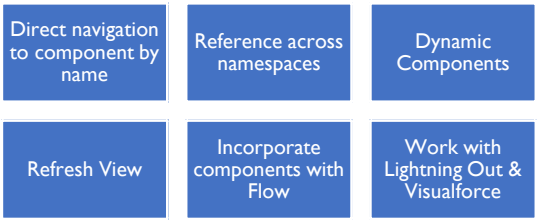
```
getSObjectValue(sobject, fieldApiName);
```

Refresh Component	Ability to sync data without refreshing entire page use RefreshView API and lightning/refresh module.

AURA COMPONENT CO-EXISTENCE

Lightning Web Component can only be child of Aura Component or other LWC, but LWC cannot be a parent of Aura Component.

Aura Component or simple wrapper is need when



LIGHTNING WEB COMPONENTS BEST PRACTICES

UI CONSIDERATIONS

Rendering:

- Declarative rending is preferable if Component is configured via Lightning App Builder
- if:true|false is preferrable to load the page faster deferring component creation based on conditions
- CSS style is preferrable when pre-loading a component and to show it based on conditions
- Use hooks for custom styling

Lifecycle hooks:

constructor() – it fires when component is created

- Don't inspect element's attributes and children, as they don't exist yet
- Don't inspect element's public properties, as they are set after component is created

connectedCallback() and disconnectedCallback() – when the component is inserted/removed into/from the DOM

- If component derives its internal state from properties, use setter than in connectedCallback()
- Use this.isConnected to check whether a component is connected to a DOM

renderedCallback() – after a component is connected and rendered

- Use Boolean to prevent renderedCallback() multiple times.
- Don't update wire adapter configuration object property and public property/field in this method.

Use Custom Datatype in Datatable

- To implement a custom cell, such as a delete row button or an image, or even a custom text or number display
- It doesn't support dispatching custom events.


```
import LightningDatatable from 'lightning/datatable';
import customNameTemplate from
'./customName.html';
import customNumberTemplate from
'./customNumber.html';

export default class MyCustomTypeDatatable extends
LightningDatatable {

        static customTypes = {
        customName: {
        template: customNameTemplate,
```

```
            standardCellLayout: true,
            typeAttributes: ['accountName'],
            }
            // Other types here
    }
}
```

DATA RELATED OPERATIONS

Lightning Data Service (LDS)

- Optimizes server calls by bulkifying and deduplicating the requests.
- When using getRecord wire adapter specify fields instead of layout whenever possible.
- Use GraphQL wire adapter which comes with client-side caching and data management capabilities.

Use of LDS Wire Adapters and Functions

- Each operation is an independent transaction. To work multiple transactions in single transaction use Apex.

Use Apex

- Unlike LDS, Apex data is not managed, we must refresh data. Use getRecordNotifyChange() to update cache.
- Loading List of records by criteria.

TIPS TO USE DECORATORS

@api

- Apply to class fields and class methods only.
- Public property which has been set by Parent component should never re-assign by Child.
- If the component needs to reflect internal value changes above rule doesn't apply.

@track

- To observe changes to the properties of an object or to the elements of an array.

@wire

- Do not track dynamic properties and use that in wire functions. It will call multiple times unnecessarily on every changes.
- Try passing object as parameter rather than passing values separately.

DESIGN CONSIDERATIONS

UI level Considerations

- Improve performance with Client-side using LDS and cache Apex methods [@AuraEnabled(cacheable=true)]
- Try leveraging pagination on front-end using JavaScript Array.slice() function rather than bringing huge data on UI.
- To import Object and field reference use static schema @salesforce/schema rather than dynamic one.

Data Retrieval Considerations

- Reference When making SELECT query, specify those fields which are necessary
- LIMIT the number of rows returned
- Lazy loading data rather than retrieving all during page load (using offset)
- To retrieve metadata, list, picklist values use UI API functions

Recommendations for server calls

- Consider making server calls when no other options are available.
- Consider reusing same data between components rather than making separate calls.
- Try using client-side filtration using JavaScript functions rather than making separate server calls

- Use continuation for long running request to external web service.
- To update records, use imperative Apex call or use UI-API methods

EVENT COMMUNICATIONS

- Use payload for EventTarget.dispatchEvent() - primitive data or copy of data to a new object
- bubbles:true and composed:true are not recommended as they bubble up entire DOM tree, across shadow boundary
- Consider using LMS over pubsub
- Don't add property into event.detail to communicate data in same shadow tree.
- Don't forget to remove event listener in disconnectedCallback() lifecycle hook

SECURITY CONSIDERATIONS

Coding Recommendations:

- Use platformResourceLoader module to import 3rd party JavaScript or CSS library
- Avoiding ClickJacking Vulnerability - Avoid using "position: fixed" or high z-index value or negative values at margins
- Never use **inline css** in html
- Restrict use of innerHTML
- Prevent event listeners from leaking memory

```
connectedCallback() {
    window.addEventListener('test', this.handleTest.bind(this));
    //                          ^ Event listener will leak.
}
```

Tools Recommendations:

- ESLint
- Evaluate JavaScript in Lightning Web Security Console
- Lookup distortion details in LWS Distortion Viewer

References

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

My Dreamforce Session LWC 20-20: 20 Tips in 20 Minutes

Author

Santanu Boral

Salesforce MVP Hall of Fame, 37x certified Professional, Dreamforce 2022 Speaker, Blogger.

Blog: https://santanuboral.blogspot.com/

Date: 29th June 2024