LIGHTNING WEB COMPONENTS CHEAT SHEET (UNOFFICIAL) – BY SANTANU BORAL

OVERVIEW

Lightning Web Components (LWC) are custom HTML elements using HTML and modern Javascript (ES7 standard & above), supported all browsers supported by Salesforce. Aura Components and LWC both coexists and interoperate on the page. It leverages web standards and delivers high performance. It is treated as Lighting Components.

FEATURES AND ADVANTAGES

- LWC leverages Web Standards
- Modern Javascript (ES7 standard)
- Simplify data access through @wire
- Build resilient Apps with Shadow DOM

Lightning Web Components

Enhanced Security
Intelligent Caching
UI Components
Data Services
UI Services
Templates

+

Web Standards leveraging

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

GETTING STARTED

1. Enable Lightning Components at Developer Edition, Setup → Develop → Lightning Components. Select Enable Lightning Component checkbox.
2. To create LWC App follow this life cycle

Create Salesforce DX Project

- In VS Code, press Command + Shift P, enter sfdx , and select **SFDX: Create Project**.
- From Command Line:
  - cd path/to/your/sfdx/projects
  - sfdx force:project:create --projectname MyLWC
  - cd MyLWC

Authorize an Org

- In VS Code, press Command + Shift P, enter sfdx , and select **SFDX: Authorize a Dev Hub**.
- From Command Line:
  - sfdx force:auth:web:login -d -a LWC-Hub

Create a Lightninng Web component

- Create component under folder force-app/main/default/lwc folder.
- From Command Line:
  - sfdx force:lightning:component:create --type lwc -n myComponent -d force-app/main/default/lwc

Deploy to Org

- In VS Code, press Command + Shift P, enter , and select **SFDX: Push Source to Default Scratch Org**.
- From Command Line:
  - sfdx force:source:push

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)

Folder Rules:

- Must - begin with lower case, only alpha numeric or underscore characters
- Can't - include whitespace, cant ends with underscore, cant contain two consecutive underscores, hyphen

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 @api, Private properties, Event handlers.

CONFIGURATION

```
<?xml version="1.0" encoding="UTF-8"?>
<LightningComponentBundle
xmlns="http://soap.sforce.com/2006/04/metadata">
    <apiVersion>45.0</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.

DECORATERS

Decorators are often used in JavaScript to extend the behavior of a class, property, getter, setter, or method.

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

@api	For exposing public property, this is reactive.
@track	Private reactive property
@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>
```

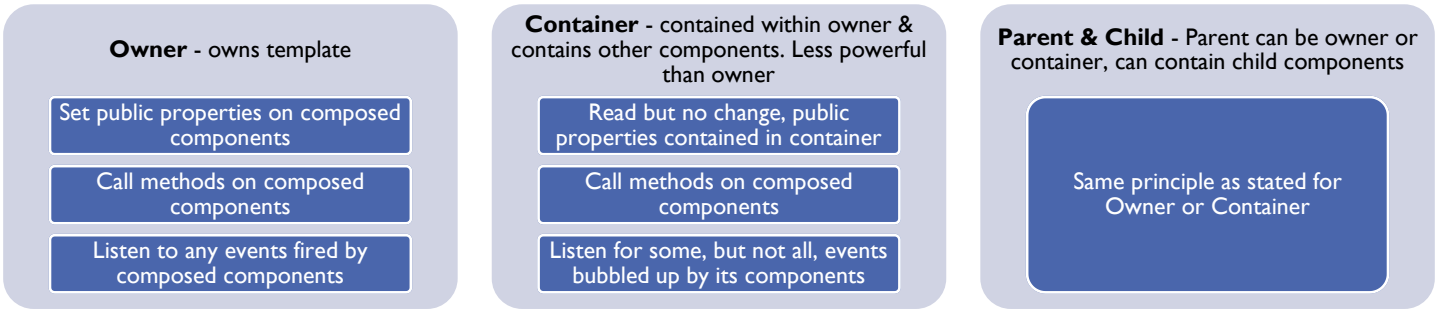
```
// myComponent.js
import { LightningElement, track, api } from 'lwc';

export default class myComponent extends LightningElement {
```

```
        </div>
    <p>{itemName}</p>
    <lightning-button label="Change Item"
                      onclick={handleClick}>
        </lightning-button>
</template>
```

```
  @api title = 'Sample Example';
   @track
   itemName = 'Bike';
   handleClick(){
      this.itemName = 'Cycle';
      console.log('itemName is ' + this.itemName);
   }
}
```

COMPOSITION



Setting property to Children

- To communication down to component hierarchy owner can set a property.
- Data-binding of property values are **one-way** from owner to child, as opposed to Aura.
- Child must treat property value as readonly.
- To trigger mutation for then owner's property, child can trigger an event to parent. If parent owns the data, parent can change property value, which propagates down to child component.

Parent

```
<!-- todoapp.html -->
<template>
    <c-todowrapper>
        <c-todoitem item-name={itemName}></c-todoitem>
    </c-todowrapper>
    <p>item in todoapp: {itemName}</p>
    <p><button onclick={updateItemName}>Update item name in todoapp</button></p>
</template>
```

Value binding

```
// 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";
    }
}
```

Child

```
<!-- c-todoitem.html -->
<template>
    <p>item in todoitem: {itemName}</p>
    <p><button onclick={updateItemName}>Update item name in todoitem</button></p>
</template>
```

```
// 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.

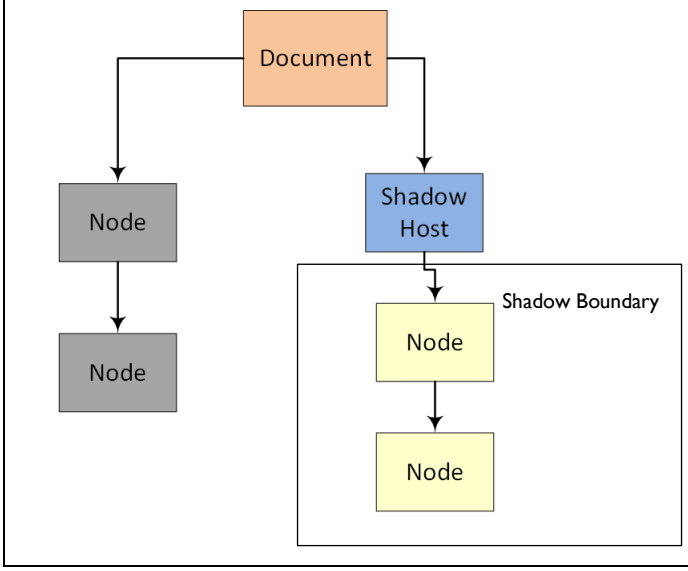
Parent	Child
<!-- methodCaller.html --> <template> <c-video-player video-url={video}></c-video-player> <button onclick={handlePlay}>Play</button> </template>	**<!-- videoPlayer.html -->** <template> <video autoplay> <source src={videoUrl} type={videoType} /> </video> </template>
// methodCaller.js import { 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(); } } /* *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.* */	**// 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(); } } get videoType() { return 'video/' + this.videoUrl.split('.').pop(); } }

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.
Access Static Resource	import myResource from '@salesforce/resourceUrl/resourceReference';
Access Labels	import labelName from '@salesforce/label/labelReference';
Access Current UserId	import Id from '@salesforce/user/Id';

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.

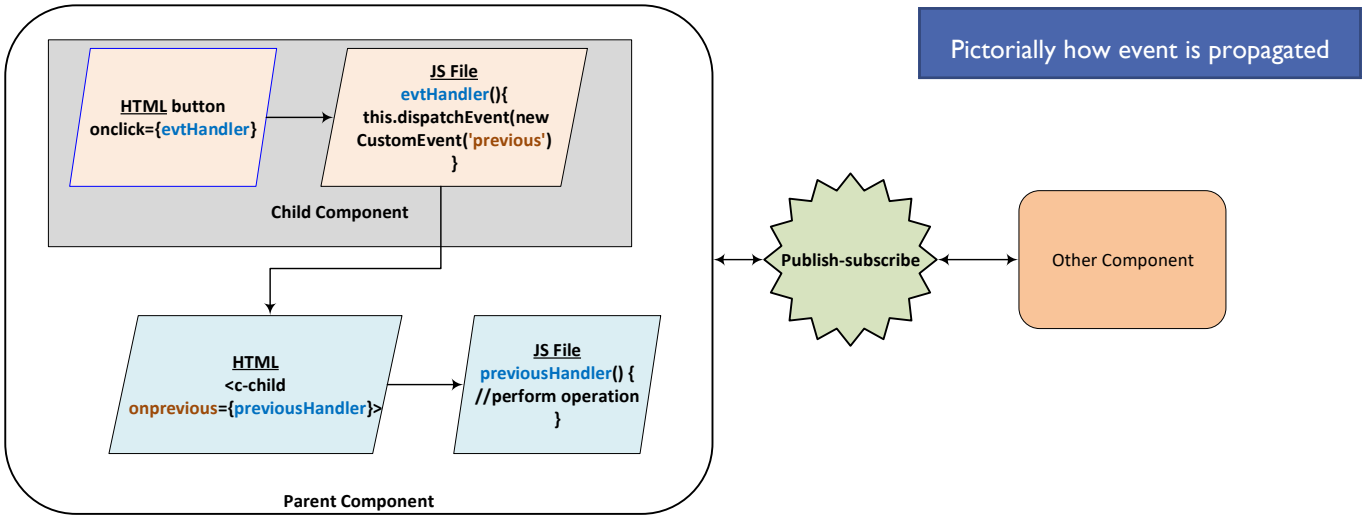
Shadow DOM is a web standard that encapsulates the elements of a component to keep styling and behavior consistent in any context.




```
<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>
```

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); }



WORKING WITH SALESFORCE DATA

<table>
<tr><td>Lightning Data Service (LDS)</td><td>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.</td></tr>
<tr><td>Using Base Components - Load a record:</td><td>

```
<template>
    <lightning-record-form
        record-id={recordId}
        object-api-name="Account"
        layout-type="Compact"
        mode="view">
    </lightning-record-form>
</template>
```

```
// myComponent.js

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

</td></tr>
<tr><td>Using Base Components - Edit a record:</td><td>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}>
    </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];

}
```

</td></tr>
<tr><td>Using Base Components - Create a record:</td><td>

```
<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.
    }
}
```

</td></tr>
</table>

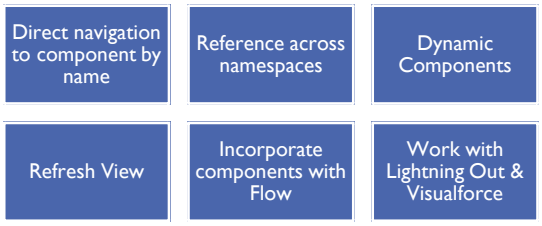
Get Data With wire service – Get record data	This is reactive, which is built on LDS. Wire adapter is one of lightning/ui*Api modules. import { LightningElement, api, wire } from 'lwc'; import { getRecord } from 'lightning/uiRecordApi'; import ACCOUNT_NAME_FIELD from '@salesforce/schema/Account.Name'; export default class Record extends LightningElement { @api recordId; @wire(getRecord, { recordId: '$recordId', fields: [ACCOUNT_NAME_FIELD]}) record; }
Get Data With wire service – Create records	createAccount is getting called from button click event. createRecord returns a promise object that resolves when record is created. createAccount() { const fields = {}; fields[NAME_FIELD.fieldApiName] = this.name; const recordInput = { apiName: ACCOUNT_OBJECT.objectApiName, fields }; createRecord(recordInput) .then(account => { this.accountId = account.id; this.dispatchEvent(new ShowToastEvent({ title: 'Success', message: 'Account created', variant: 'success', }),); }) .catch(error => { this.dispatchEvent(new ShowToastEvent({ title: 'Error creating record', message: error.body.message, variant: 'error', }),); }); }
Handle Errors	@api recordId; @track error; @wire(getRecord, { recordId: '$recordId', fields }) wiredRecord({error, data}) { if (error) { this.error = 'Unknown error'; if (Array.isArray(error.body)) { this.error = error.body.map(e => e.message).join(', '); } else if (typeof error.body.message === 'string') { this.error = error.body.message;

	} this.record = undefined; } else if (data) { // Process record data } }
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; **Wire a Apex method with dynamic parameter:** @wire(findContacts, { searchKey: '$searchKey' }) contacts; **Import Objects and Fields from @salesforce/schema** getSObjectValue(sobject, fieldApiName);

AURA COMPONENT CO-EXISTANCE

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



REFERENCES

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

Author: Santanu Boral, Salesforce MVP, 18x certified, Salesforce certified Application Architect and System Architect.

Date: 21st July, 2019