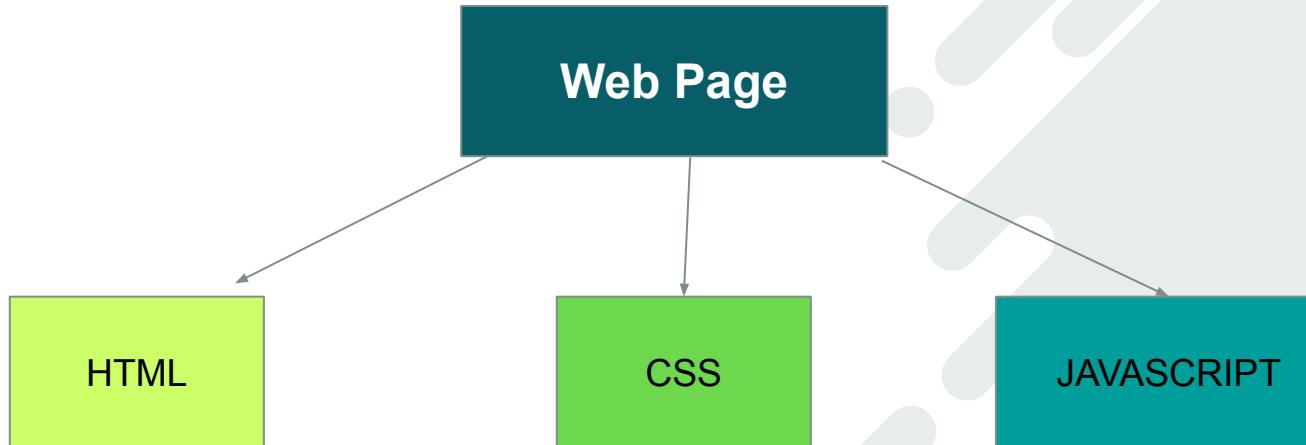


# Fundamentals of Web Technology

# Web Core Technologies



## Create Structure

1. Heading
2. Paragraphs
3. Lists

## Stylize the Website

1. Color
2. Background color
3. Fonts
4. border

## Increase interactivity

1. Dynamic display
2. Widgets
3. User interaction
4. Click to open popup

# Tools for Development



# Browser (chrome preferable)

[https://www.google.com/intl/en\\_au/chrome/](https://www.google.com/intl/en_au/chrome/)



## Get more done with the new Chrome

Now more simple, secure and faster than ever – with Google's smarts built-in.

[Download Chrome](#)

For Windows 10/8.1/8/7 64-bit.

Help make Google Chrome better by automatically sending usage statistics and crash reports to Google. [Learn more](#)

By downloading Chrome, you agree to the [Google Terms of Service](#) and [Chrome and Chrome OS Additional Terms of Service](#)

# Install Visual Studio Code Editor

<https://code.visualstudio.com/Download>

The screenshot shows the official Visual Studio Code website at the top, featuring a dark header with navigation links like 'Docs', 'Updates', 'Blog', 'API', 'Extensions', and 'FAQ'. A prominent 'Download' button is visible. Below the header, a message announces 'Version 1.49 is now available! Read about the new features and fixes from August.' The main content area features a large, bold headline 'Code editing. Redefined.' with the subtitle 'Free. Built on open source. Runs everywhere.' Below this, there's a 'Download for Windows' section with 'Stable Build' and 'Insiders' options, and download links for macOS, Windows x64, and Linux x64. A 'Other downloads' section follows. To the right, the actual Visual Studio Code application is shown in its dark theme. It displays a code editor with several tabs open (e.g., 'serviceWorker.js', 'index.js', 'serviceWorker.js') containing JavaScript code. The sidebar shows the 'EXTENSIONS: MARKETPLACE' section with various extensions listed, such as Python, GitLens, C/C++, ESLint, Debugger for Chrome, Language Support, vscode-icons, and Vetur. At the bottom of the application window, there's a terminal tab showing 'node' and network status information.

# Live Server Extension

File Edit Selection View Go Run Terminal Help

Extension: Live Server - CAR\_HUB\_DEMO - Visual Studio Code [Administrator]

EXTENSI... ⌂ ⌂ ...

live server

**Live Server** 5.6.1  
Launch a development lo...  
Ritwick Dey

**Live Share** 1.0.2902  
Real-time collaborative d...  
Microsoft **Install**

**SQL Server (mssql)** 1.9.0  
Develop Microsoft SQL Se... **Install**

**Live Server** ritwickdey.liveserver

Ritwick Dey | ⚡ 7,548,320 | ★★★★★ | Repository | License | v5.6.1

Launch a development local Server with live reload feature for static & dynamic pages

Disable ▾ Uninstall This extension is enabled globally.

**Details** Feature Contributions Changelog

# BASICS OF HTML

# HTML - Hypertext Markup Language

- Hypertext refers to the way in which Web pages (HTML documents) are linked together.
- Markup language is a computer language that uses tags to define elements within a document.
- HTML Describes the structure of a Web Page
- HTML elements tell the browser how to display the content.

# Most commonly Used HTML Elements in LWC

An HTML element is defined by a start tag, some content, and an end tag:

```
<tagname>Content goes here...</tagname>  <h1>My First Heading</h1>
```

- **<div>** - This tag defines a division in an HTML document.
- **<p>** - This tag used for paragraph that starts on a new line and add some margin before and after a paragraph
- **<h1>-<h6>** - These tags are used to give heading of different size
- **<a>** - This tag defines a hyperlink
- **<img>** - This tags help you to insert an image in web page
- **<ul>** - This will list items using plain bullets.

# HTML Attributes

- HTML attributes provide additional information about HTML elements
- Attributes are always specified in the start tags
- Attributes usually come in name/value pairs like: name="value"

**Example -** <a href="https://google.com">Google</a>

## HTML data-\* Attribute

- The data-\* attribute is used to store custom data private to the page or application.

**Example -** <div data-name="nikhil">Username</div>

# HTML Block and Inline Element

- **Block Level** - Element always starts on a new line and takes up the full width available

**Example** - <div>, <p>, <h1>-<h6><ul>, <header>, <article>

- **Inline Level** - Element does not start on a new line and it only takes up as much width as necessary

**Example** - <span>, <img>, <a>,<label>, <strong>

# Basics of CSS

# CSS

- CSS stands for Cascading Style Sheets
- CSS is used to control the style of a web page

## CSS Syntax

**selector { property: value }**

- **Selector** - It is used to “find” the HTML element
- **Property** - CSS property you want to apply like color, border etc
- **Value** - It’s the value assigned to property like for color property value is red

# CSS Selectors

- CSS selectors are used to "find" (or select) the HTML elements you want to style.
- **Element Selector** - use of html tags for styling

```
p{ color: blue;}
```

- **ID selector** - uses the id attribute of an HTML element to select a specific element. To select an element with a specific id, write a hash (#) character, followed by the id of the element. *The id of an element should be unique within a page*

```
#username{ color: red;}
```

- **Class selector** - uses the class attribute of an HTML element to select a specific element. To select an element with a specific class, write a dot(.) character, followed by the class of the element.

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

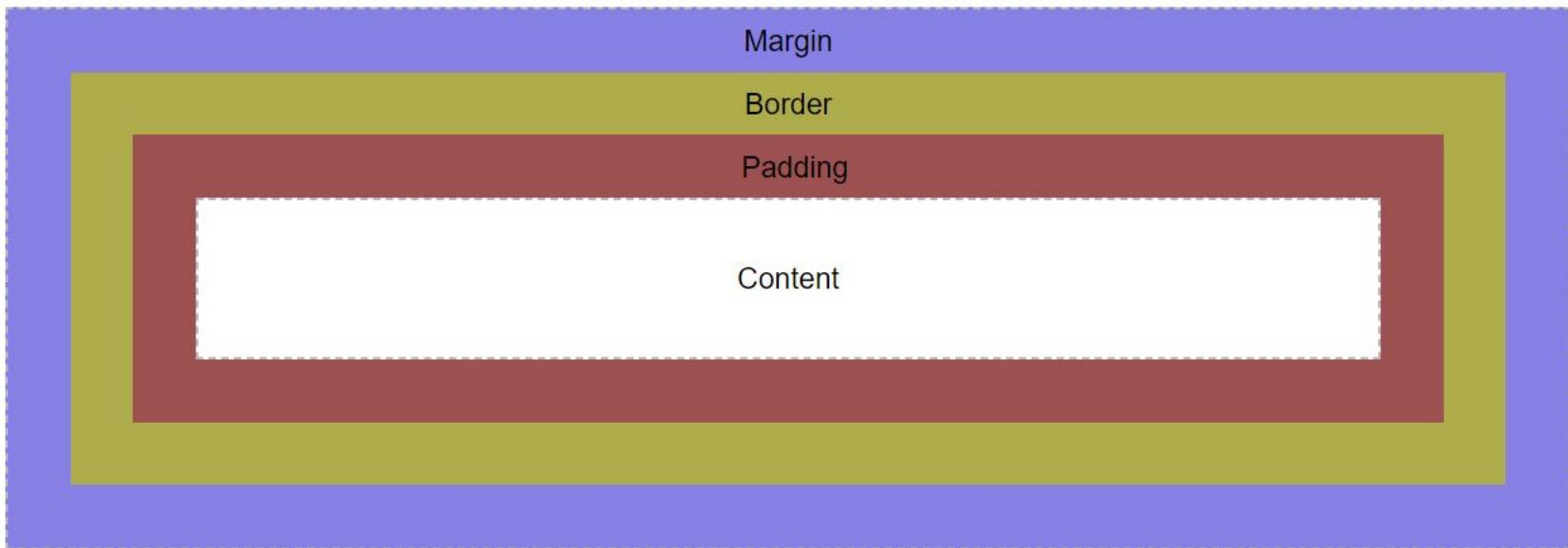
# Type of CSS styles

There are Three types of CSS styles

1. Internal CSS - Not used in LWC
2. Inline CSS
3. External CSS or Third party library - Bootstrap, salesforce lightning design system

# CSS Box Model

It is a box that wraps around every HTML element.



# JavaScript For LWC



# JavaScript

- JavaScript is a scripting or programming language that allows you to implement complex features on web pages
- It brings interactivity to the web pages

# JavaScript Learning Path

Variables

Data types

null vs undefined

Arrow function

Spread Operator

Destructuring

String Interpolation

String Methods

Object methods

Array Methods

Promises

Modules import and export

Events

setTimeout vs setInterval

QuerySelectors



# 1. Variables

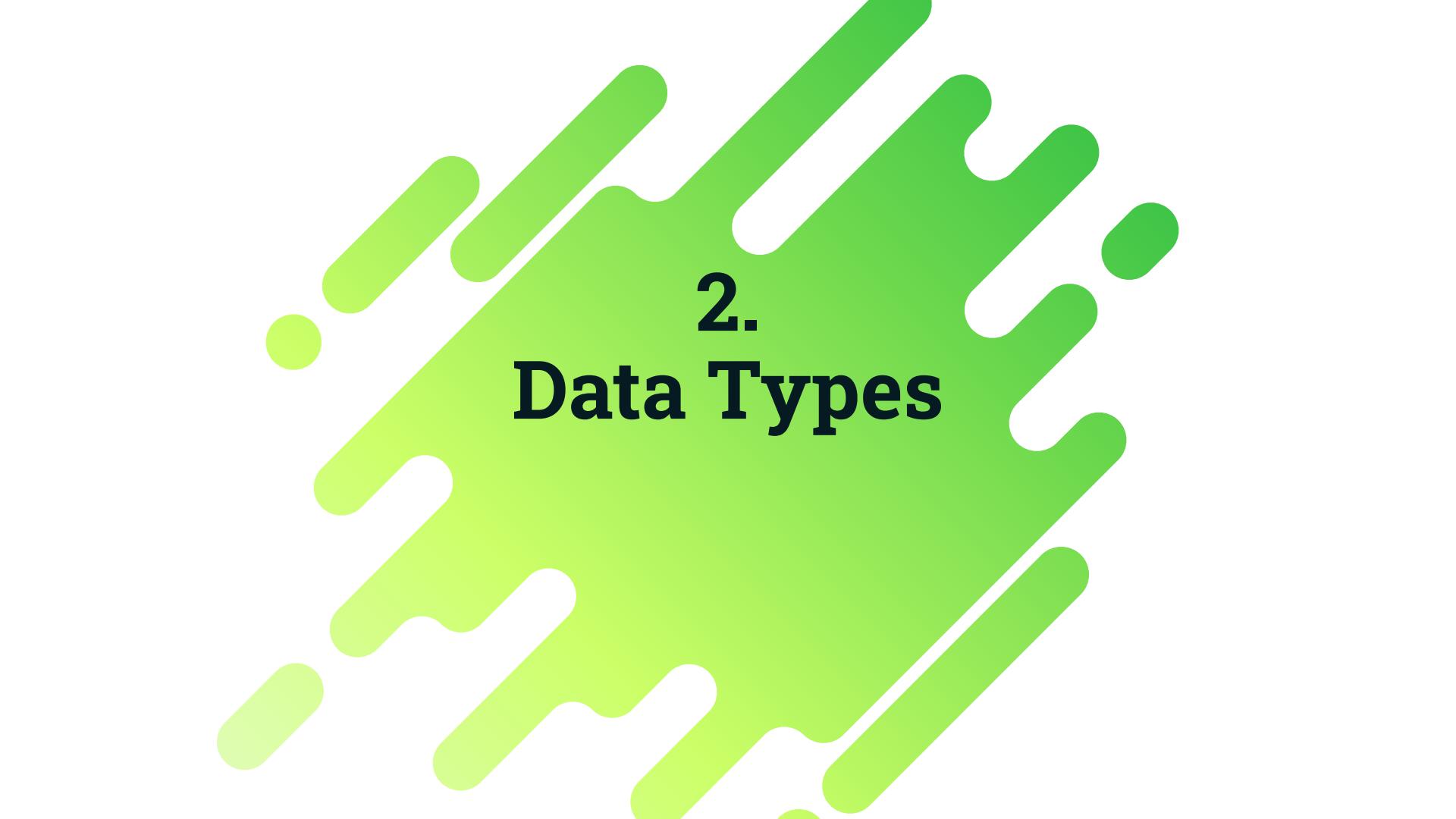
# Variables

- Variables are the containers for storing data values
- `var`, `const` and `let` are the reserved keyword to declare a variable
- JavaScript identifiers are case-sensitive.
- You can assign a value to a variable using equal to (=) operator when you declare it or before using it.

Keyword	<code>const</code>	<code>let</code>	<code>var</code>
Global scope	Yes	Yes	Yes
Function scope	Yes	Yes	Yes
Block scope	Yes	Yes	No
Can be reassigned	No	Yes	Yes

`let` variables can be updated but not re-declared;

`const` variables can neither be updated nor re-declared.



## 2. Data Types

# Data types in JS

There are 8 basic data types in JavaScript.

1. *number*

2. *string*

3. *boolean*

4. *BigInt*

5. *undefined*

6. *null*

7. *object*

8. *Symbol*

Rest all types are Object in some form

- 1. Array
- 2. Date
- 3. Math
- 4. String
- 5. etc.



## 3. **Null vs Undefined**

# Null vs Undefined

SNO	undefined	null
1	If a variable is declared, but not initialized or assigned any value then JS automatically initialized it with undefined	It's a special data type which represent "nothing" or "empty" or "value unknown" It is Assigned explicitly
2	<code>typeof undefined</code> is undefined	<code>typeof null</code> is object
3	<code>undefined == null</code> is true	<code>undefined === null</code> is false

## Equality Comparison

`==` This operator does value comparison. *Example 100 == "100"*

`===` This operator does value plus data type comparison *Example 100 === "100"*

# Arrow Function

SNO	undefined	null
1	If a variable is declared, but not initialized or assigned any value then JS automatically initialized it with undefined	It's a special data type which represent "nothing" or "empty" or "value unknown" It is Assigned explicitly
2	<code>typeof undefined</code> is undefined	<code>typeof null</code> is object
3	<code>undefined == null</code> is true	<code>undefined === null</code> is false

# Equality Comparison

`==` This operator does value comparison. *Example 100 == "100"*

`===` This operator does value plus data type comparison *Example 100 === "100"*



## 4. **Spread Operator**

# Spread Operator

The operator's shape is three consecutive dots and is written as: `...`

Usages of spread operator

1. **Expanding String** - Convert string into list of array
2. **Combining Arrays** - Combine array or add value to array
3. **Combining Object** - Combine Object or add value to Object
4. **Creating New Shallow Copy of Arrays and objects**



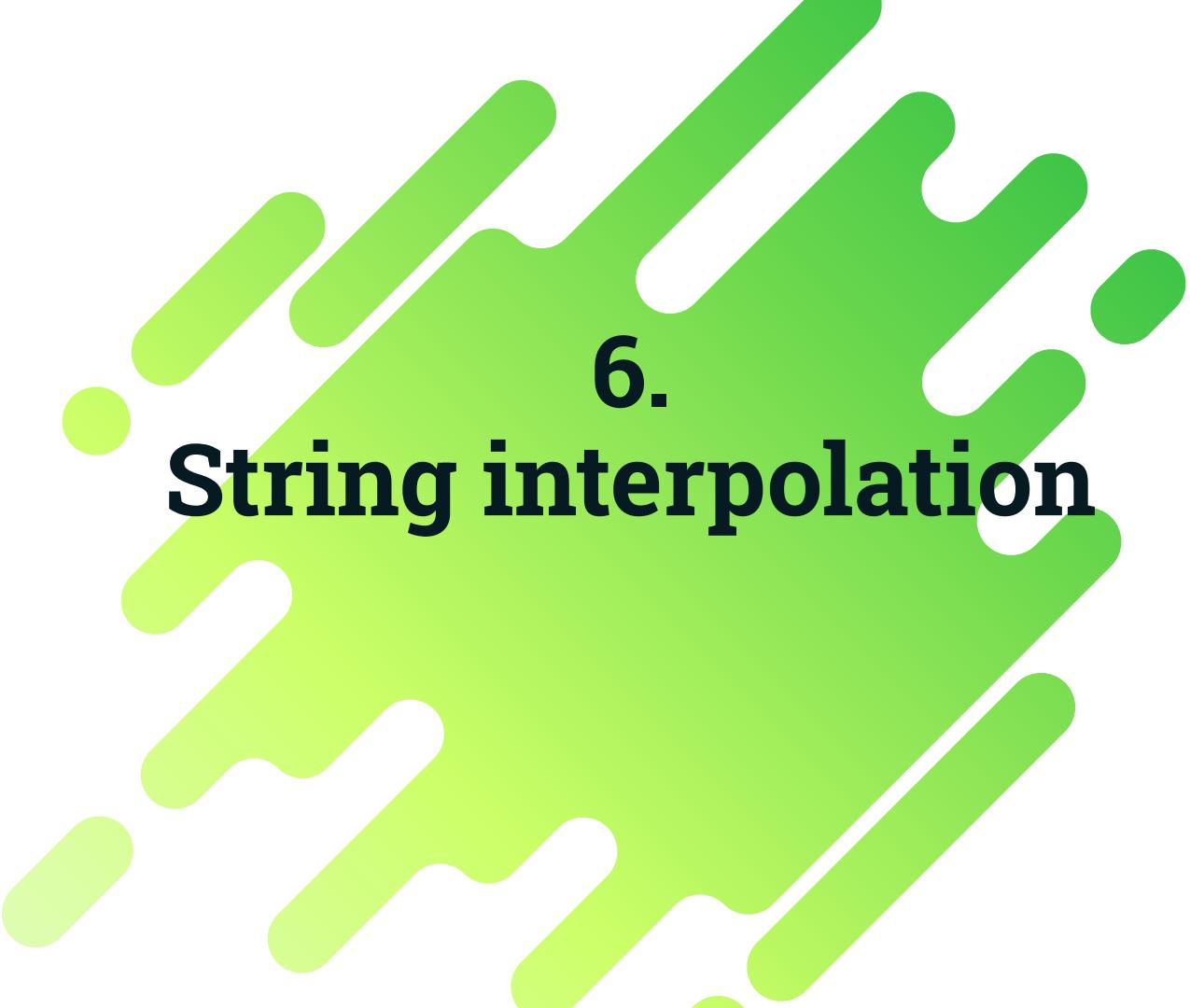
## 5. **Destructuring**

# Destructuring

The two most used data structures in JavaScript are Object and Array

*Destructuring* is a special syntax that allows us to “unpack” arrays or objects into a bunch of variables, as sometimes that’s more convenient.

- Array destructuring
- Object destructuring



## 6. **String interpolation**

# String Interpolation

- String interpolation allows you to embed expressions in the string
- Template strings use back-ticks (``) rather than the single or double-quotes.

```
// string interpolation
var name = "Salesforce Troop";
console.log(`Hello, ${name}!`);

// String Interpolation with expression
var a = 10;
var b = 10;
console.log(`The sum of ${a} and ${b} is ${a+b}
`);
```



## 7. **String Methods**

# String Methods

- JavaScript provides Many methods to play with strings. Below are some of the most commonly used strings method in LWC

1. *includes()*

2. *indexOf()*

3. *startsWith()*

4. *slice()*

5. *toLowerCase()*

6. *toUpperCase()*

7. *trim()*



# 8. **Object/JSON Operations**

# Object/JSON Operations

1. `Object.keys()`

2. `Object.values()`

3. `JSON.stringify()`

4. `JSON.parse()`



# 9. Array Methods

# Array Methods

1. **map()** - loop over the array and return new array based on the value return.
2. **every()** - return true or false if every element in the array satisfy the condition
3. **filter()** - return new array with all the elements that satisfy the condition
4. **some()** - return true if at least one element in the array satisfy
5. **sort()** - sort the elements of an array
6. **reduce()** - this method reduces the array to a single value ( left to right)
7. **forEach()** - this method calls for each array element



## **10.** **Promise**

# Promise

*Promise is an object that may produce a single value sometime in the future*

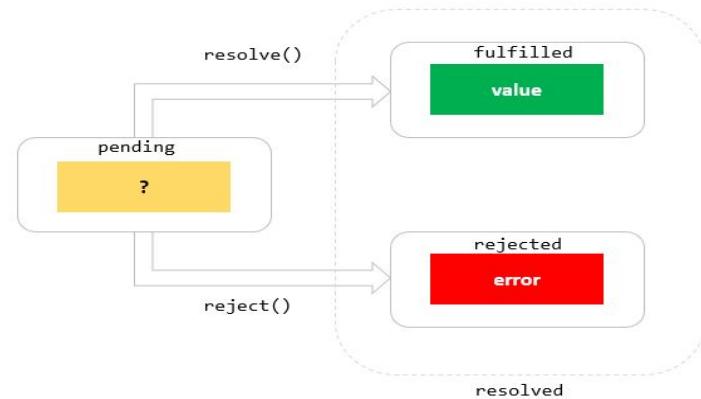
*Promises are used to handle asynchronous operations in JavaScript.*

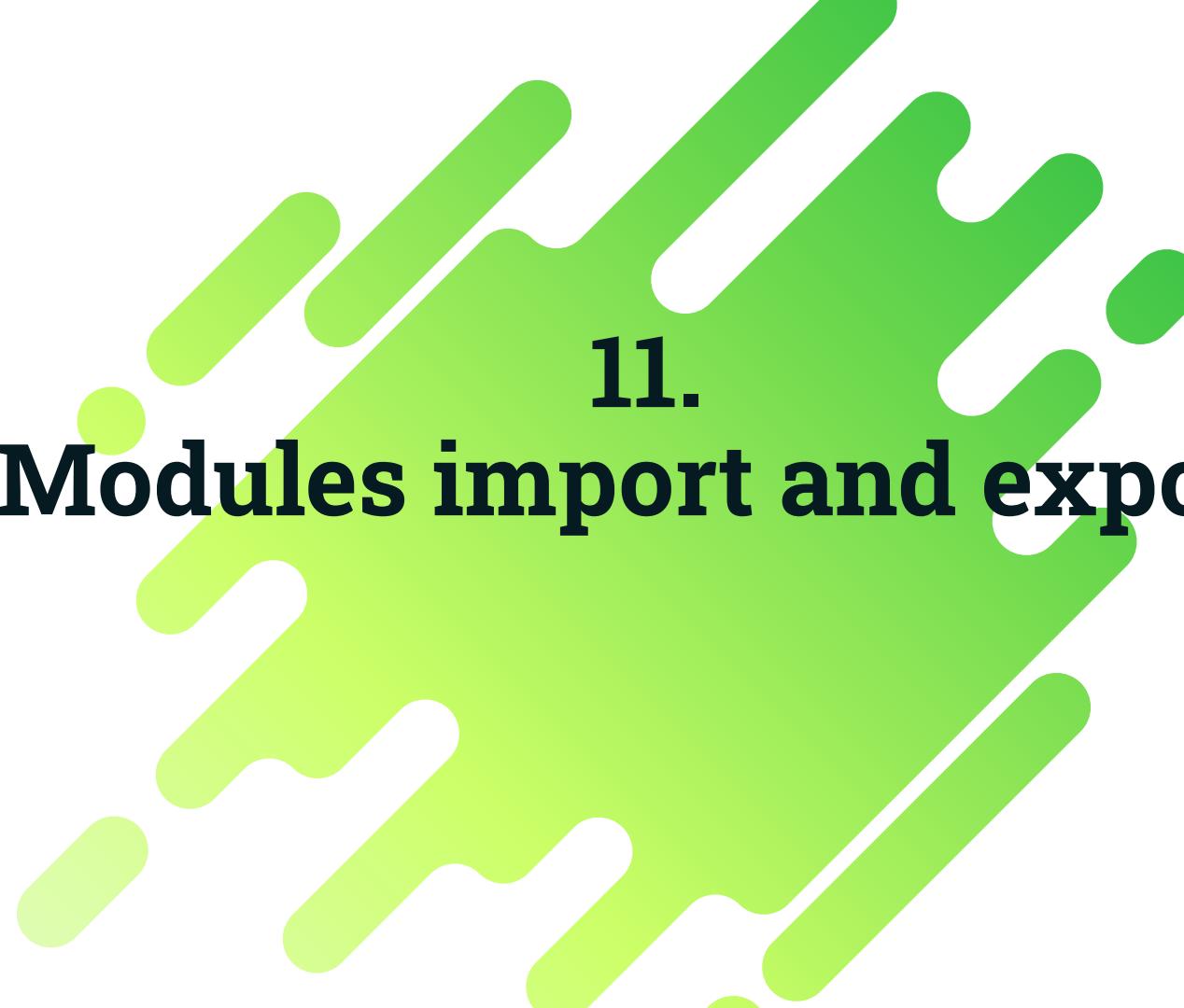
*A Promise has three states*

1. **pending()**
2. **fulfilled()**
3. **rejected()**

*Use case from LWC point of view.*

1. Fetching data from server
2. Loading file from system





11.

# Modules import and export

# Exports

*Exporting - Use **export** keyword to export many variable or many method from a file*

```
export const name = "nikhil"  
  
export function getName(){  
    return "nikhil"  
}
```

*Default export - Use **export default** keyword to export only one variable or a method from a file*

```
export default user = "salesforce"
```

# Imports

**Importing - Use `import` keyword to import variable or method from a given file path or module.**

## **Multiple imports**

```
import {name, getName} from './filepath'
```

## **Imports all exported members**

```
import * as Utils from './filepath'
```

## **Imports a module with a default member**

```
import user from './filepath'
```



## 12. QuerySelector

# querySelector

The `querySelector()` method returns the first element that matches a specified CSS selector(s) in the document.

```
document.querySelector(selector);
```

# querySelectorAll

The `querySelectorAll()` method returns all elements in the document that matches a specified CSS selector(s), as a static `NodeList` object.

```
document.querySelectorAll(selector);
```



## **13.** **Events**

# Event

An event is an action that occurs in the web browser, which the web browser feedbacks to you so that you can respond to it.

*For example*, when users click a button on a webpage, you may want to respond to this click event by displaying a alert box.

**Event handler** - It is a block of code that will execute when the event occurs. It is also known as an event listener.

# Two Common ways to add events

1. **HTML Event Handler attribute** - When we add event through HTML, Event always begin with **on** keyword like onclick, onchange, onkeyup
2. **Event Listener** - Event Handlers provide two main methods for dealing with the registering/deregistering event listeners:
  - addEventListener() – register an event handler
  - removeEventListener() – remove an event handler

# Event Propagation

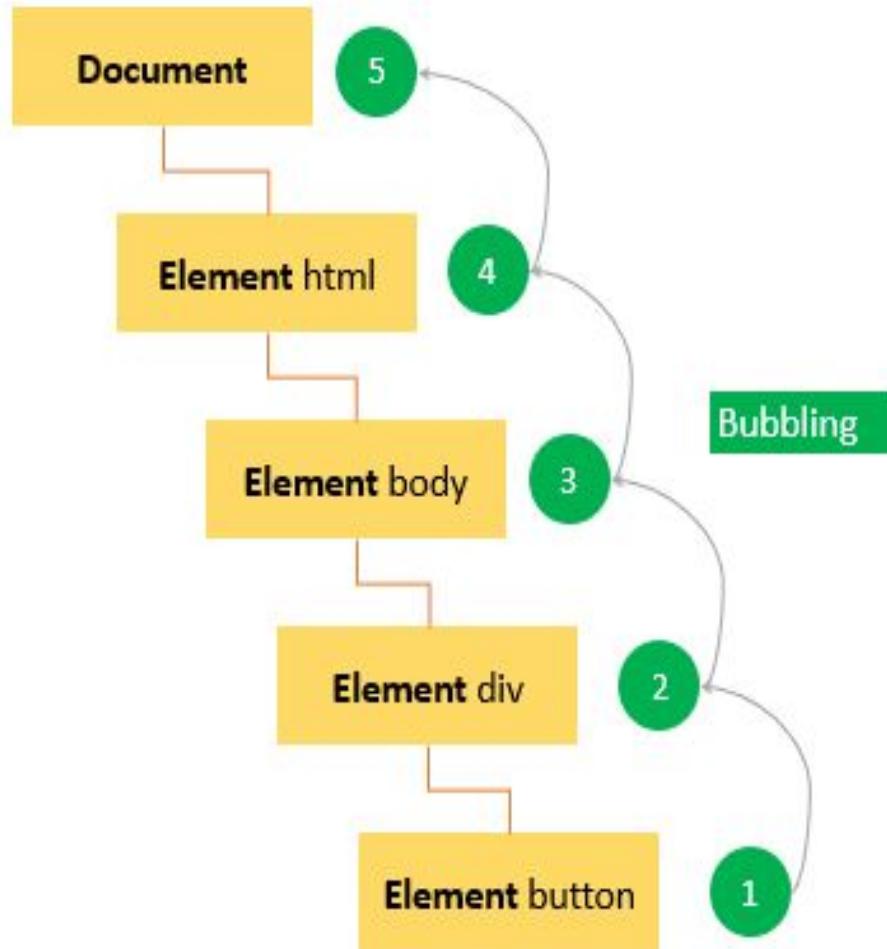
Event Propagation explains the order in which events are received on the page from the element where the event occurs and propagated through the DOM tree.

There are two main event models

1. Event bubbling
2. Event Capturing

# Event Bubbling

In the event bubbling model, an event starts at the most specific element and then flows upward toward the least specific element (the document or even window).



# Custom Event

In JavaScript we can create our own custom event using CustomEvent constructor

## Syntax

```
new CustomEvent("eventName", {options})
```

## Example

```
dispatchEvent is use to trigger the event  
element.dispatchEvent(  
    new CustomEvent("hello",  
        {  
            detail: { name: "John" }  
        }  
    ));
```

Event Name

detail property used to pass data



# 14. Arrow Function

# Arrow Function

Arrow functions allow us to write shorter function syntax

Example

```
function getName(){  
    return "Nikhil"  
}
```

Normal function

```
getName = ()=> {  
    return "Nikhil"  
}
```

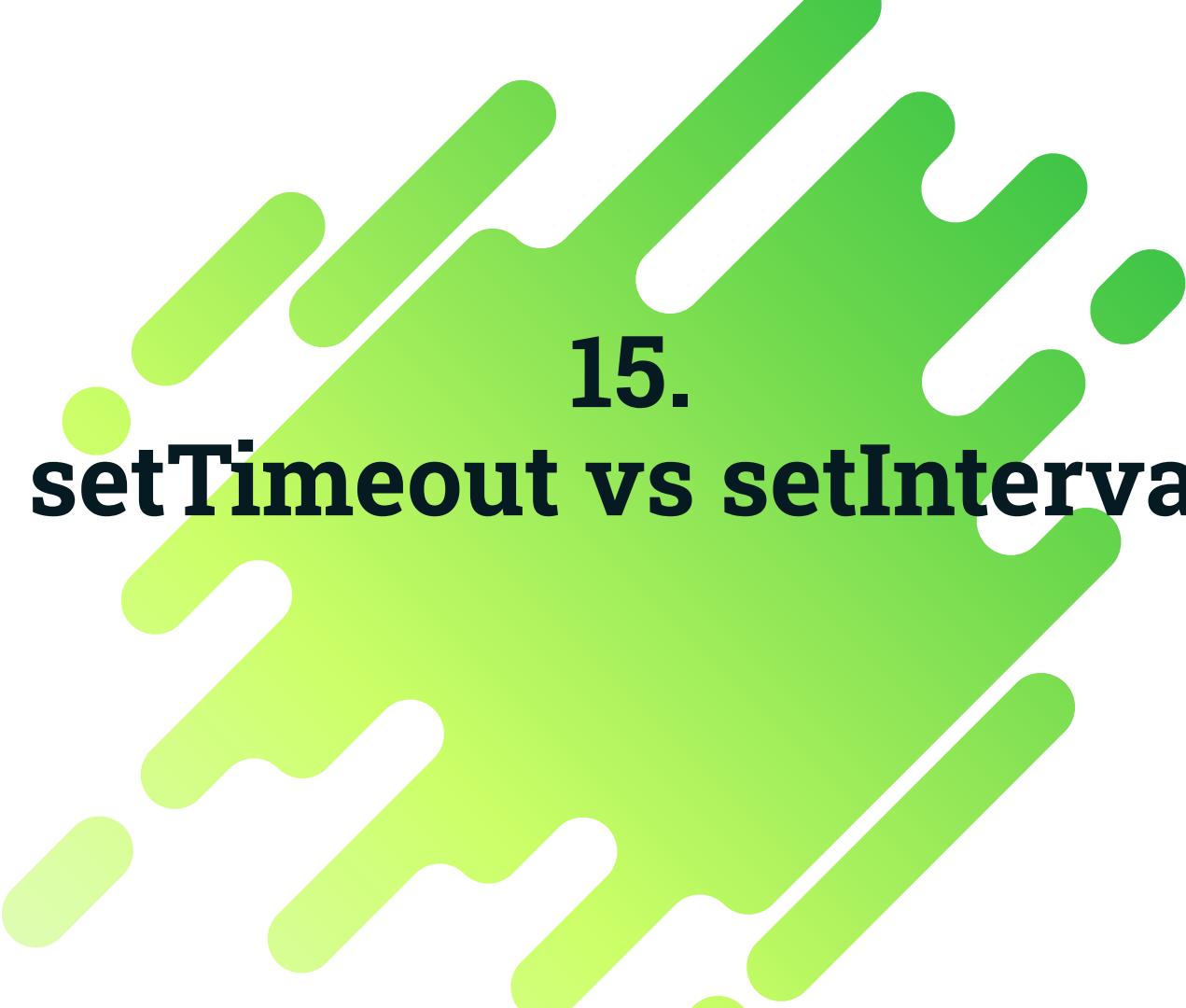
Arrow function with return

```
getName = () => "Nikhil"
```

Arrow function without return

## Benefits of Arrow functions

Arrow syntax automatically binds *this* to the surrounding code's context



# 15. **setTimeout vs setInterval**

## setTimeout

The `setTimeout()` is a method of the window object. The `setTimeout()` sets a timer and executes a callback function after the timer expires.

## setInterval

The `setInterval()` is a method of the window object. The `setInterval()` repeatedly calls a function with a fixed delay between each call.

# Intro to Lightning Web Components



# Lightning Framework

The Lightning Component framework is a UI framework for developing single page applications for mobile and desktop devices.

Lightning Components using two programming models.

1. Aura Components Model
2. Lightning Web Component Model

# Web Stack transformation

2014



2019



Lightning Web Components



Lightning  
Web Components

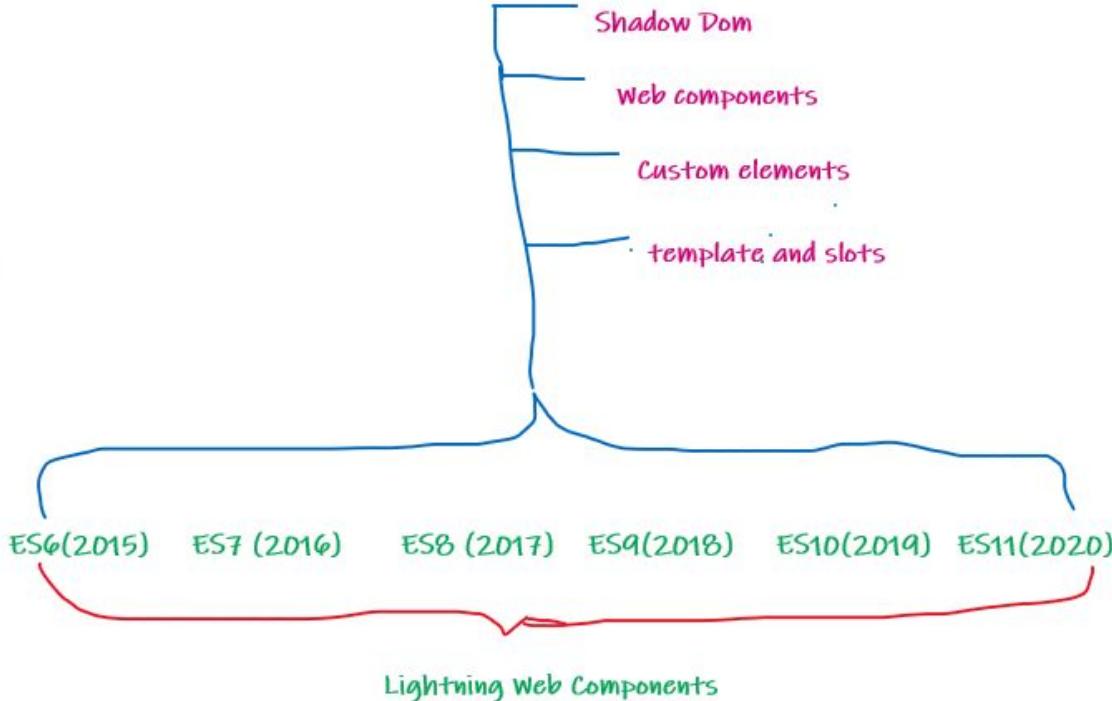
Web Standards

# Aura vs LWC

- Too much code to write
- Rendering wasn't optimized
- Modern features were not available like modules, classes and promises

ES5 (2009)

↓  
Aura build on  
this JavaScript  
Version



# What is Lightning Web Components

Lightning Web Components is a new programming model for building Lightning components. It uses the core concepts of Web Standards

## Benefits

1. Lightweight framework
2. Better Performance
3. No need to learn different framework to develop application in Salesforce
4. Interoperability with lightning Aura components
5. Better testing using Jest
6. Better Security

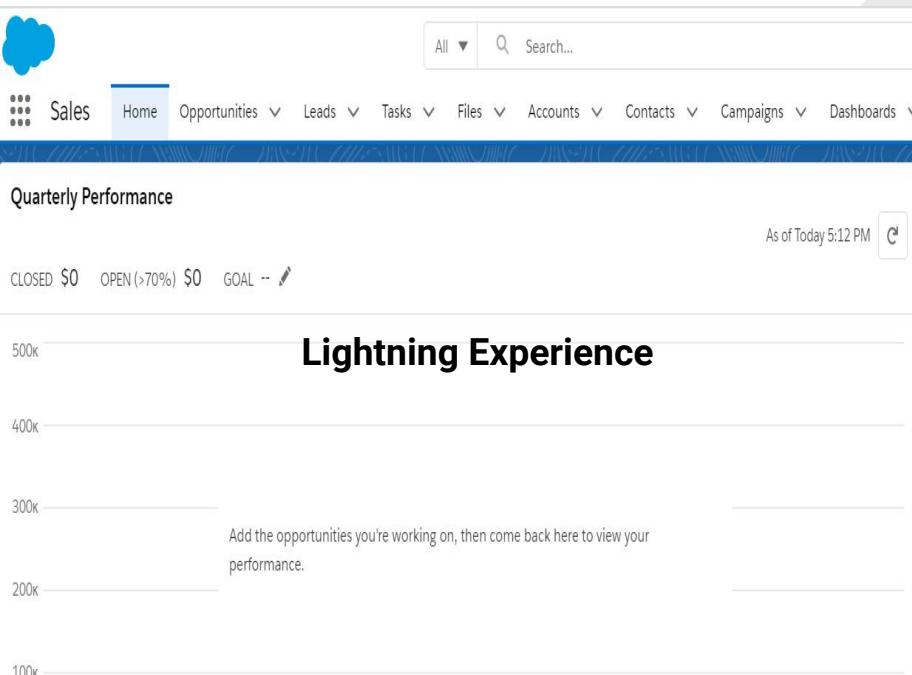
# Coexistence and interoperability

Aura components and Lightning web components can coexist and interoperate, and they share the same high level services:

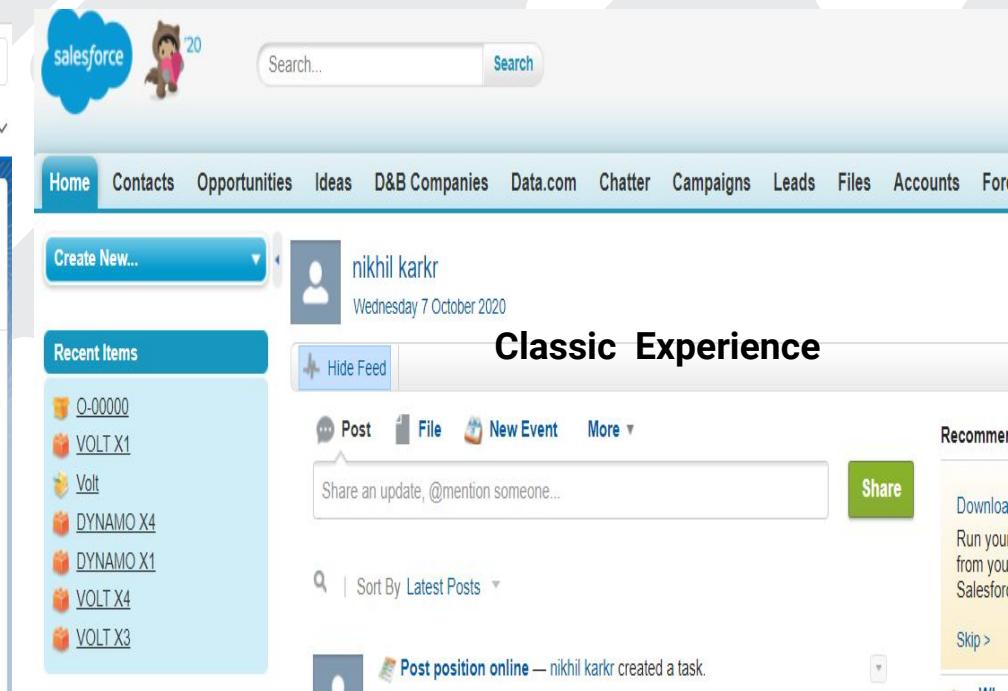
- Aura components and Lightning web components can coexist on the same page
- *Aura components can include Lightning web components but not other way around*
- Aura components and Lightning web components share the same base Lightning components. Base Lightning components were already implemented as Lightning web components.
- Aura components and Lightning web components share the same underlying services (Lightning Data Service, User Interface API, etc.).

# Lightning Experience

Lightning experience is the new user interface (UI) for salesforce installations. It is a significant upgrade from the Salesforce Classic view, bringing modern appearance and functionality to the salesforce Platform.



The screenshot shows the Salesforce Lightning Experience dashboard for the Sales team. At the top, there's a navigation bar with icons for Home, Opportunities, Leads, Tasks, Files, Accounts, Contacts, Campaigns, and Dashboards. Below the navigation is a search bar with a placeholder "Search...". The main content area features a "Quarterly Performance" chart with a blue gradient background. The chart displays performance metrics: CLOSED \$0, OPEN (>70%) \$0, and GOAL with a pencil icon. A note says "Add the opportunities you're working on, then come back here to view your performance." To the right of the chart, the text "Lightning Experience" is displayed in large, bold, black letters. The overall aesthetic is clean and modern with a light blue color palette.



The screenshot shows the Salesforce Classic Experience Chatter feed. At the top, there's a navigation bar with links for Home, Contacts, Opportunities, Ideas, D&B Companies, Data.com, Chatter, Campaigns, Leads, Files, Accounts, and more. Below the navigation is a search bar with a placeholder "Search..." and a "Search" button. The main content area features a "Create New..." button and a user profile for "nikhil karkr" with a timestamp "Wednesday 7 October 2020". A "Recent Items" sidebar lists several items, including "O-00000", "VOLT X1", "Volt", "DYNAMO X4", "DYNAMO X1", "VOLT X4", and "VOLT X3". To the right, the text "Classic Experience" is displayed in large, bold, black letters. Below it, there are buttons for "Post", "File", "New Event", and "More". A text input field says "Share an update, @mention someone..." with a "Share" button. At the bottom, there's a "Sort By Latest Posts" button and a notification: "Post position online — nikhil karkr created a task." The overall aesthetic is more traditional with a white background and a mix of blue and grey colors.

# Setting Up Developer org



# Setting Up Salesforce DX Environment



# Salesforce Developer Experience(DX)

It is a set of tools that streamlines the entire development life cycle. It improves team development and collaboration, facilitates automated testing and continuous integration, and makes the release cycle more efficient and agile.

1. *Visual Studio Code*

2. *Salesforce CLI*

3. *Salesforce Extension Pack*

# Setting Up My Domain and Dev Hub



# My Domain

Having a custom domain is more secure, some salesforce features require it.

## Dev Hub

We have to Enable Dev Hub to

1. Create and Manage orgs from the command line
2. View Information about Scratch orgs
3. Link Namespace orgs

# Setting up Project and Scratch Org



# DevHub VS Scratch org

1. **Dev Hub** - It is the main Salesforce org that you will use to create and manage your scratch orgs.
2. **Scratch Org** - It is a source-driven and disposable deployment of Salesforce code and metadata. Scratch Orgs are driven by source, Sandboxes are copies of production

*Note - Scratch orgs do not replace sandboxes*

# Project creation

```
sfdx force:project:create -n "LWC project"  
-n new project
```

## Authorization of org

```
sfdx force:auth:web:login -a lwclearning -d  
-a alias  
-d default
```

## Scratch Org Creation

Add **hasSampleData:"true"** property to **project-scratch-def.json**

```
sfdx force:org:create -a lwcScratchOne -d 30 -f config/project-scratch-def.json -s  
-a alias  
-d days(number of days scratch org alive 1 day min and 30 day max)  
-f file location of project scratch definition json  
-s set it to default username
```

Anytime you encounter error “no org configuration for name” the run the following command

*Sfdx force:config:set defaultdevhubusername=<the DX DevHub username>*

### **Example**

*sfdx force:config:set defaultdevhubusername= salesforce@dev.com*

# Create Your First Component in LWC



# Create Your First Component in LWC



# Component Naming Convention

The folder and its files must follow these naming rules.

- Must begin with a lowercase letter
- 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)

# Two ways to Create Component

## 1. Using Terminal

```
sfdx force:lightning:component:create --type lwc -n helloWorld
```

## 2. Using Command Palette

VsCode => View => Command Palette => Type Create Lightning Web Component => Hit Enter => Enter desired filename => hit enter => Again hit enter to choose default path

# Component Folder Structure



# Component Folder Structure.



# Different Naming Conventions Available in LWC



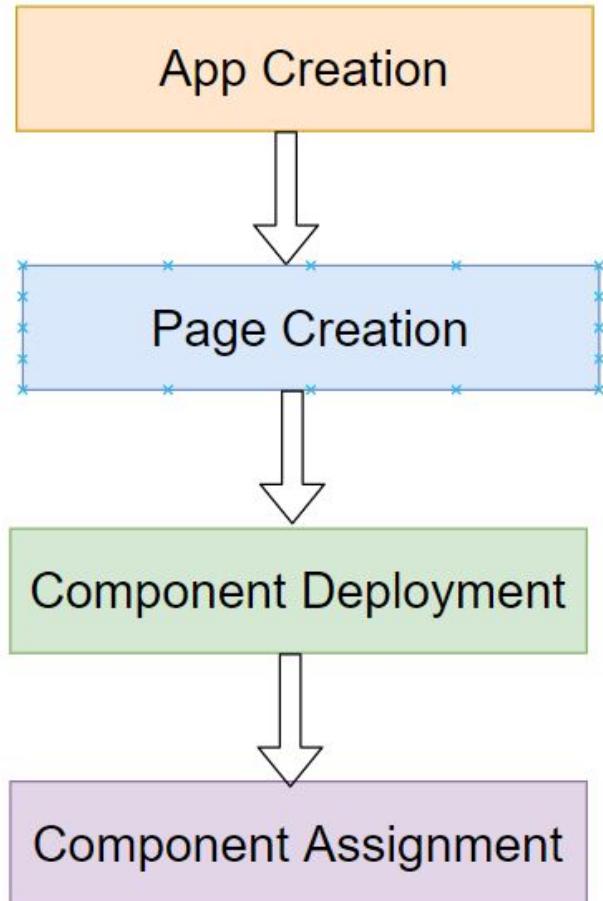
# Naming Conventions for LWC

1. **camelCase** : Each word in the middle of the respective phrase begins with a capital letter.
2. **PascalCase**: It is same like Camel Case where first letter always is capitalized.
3. **kebab-case**: Respective phrase will be transferred to all lowercase with hyphen(-) separating words.

Case Name	camelCase	PascalCase	kebab-case
Example	helloWorld	HelloWorld	hello-world
Usage	component Name	Component class Name	Component reference and HTML attribute Name

# App Creation and Component Deployment



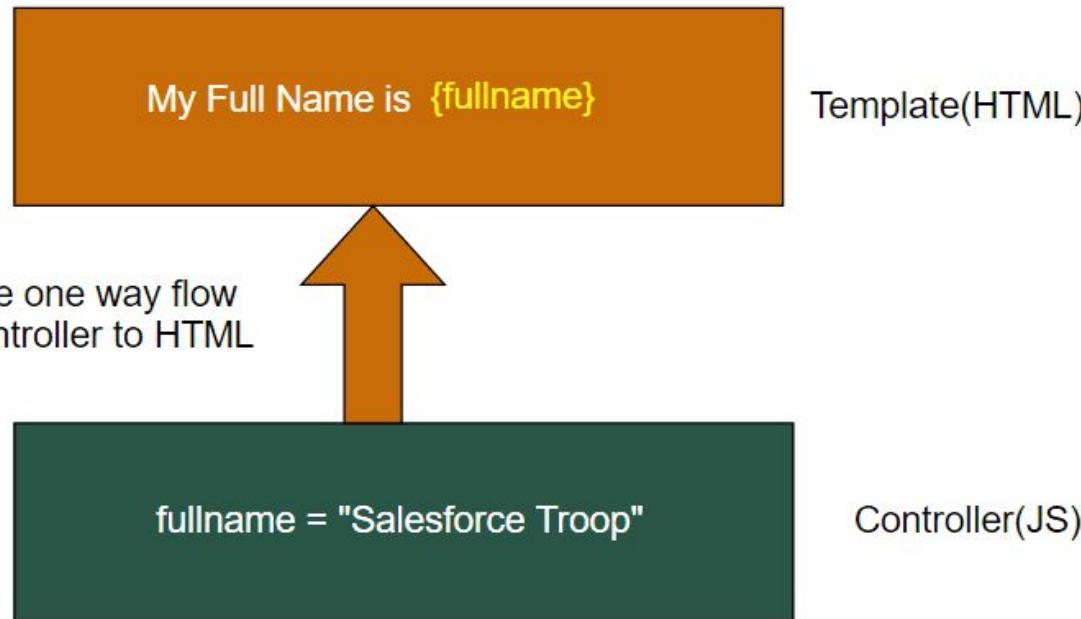


# Local Properties And Data Binding



# Data Binding in a Template

Data binding in the Lightning web component is the synchronization between the controller and the template(HTML).



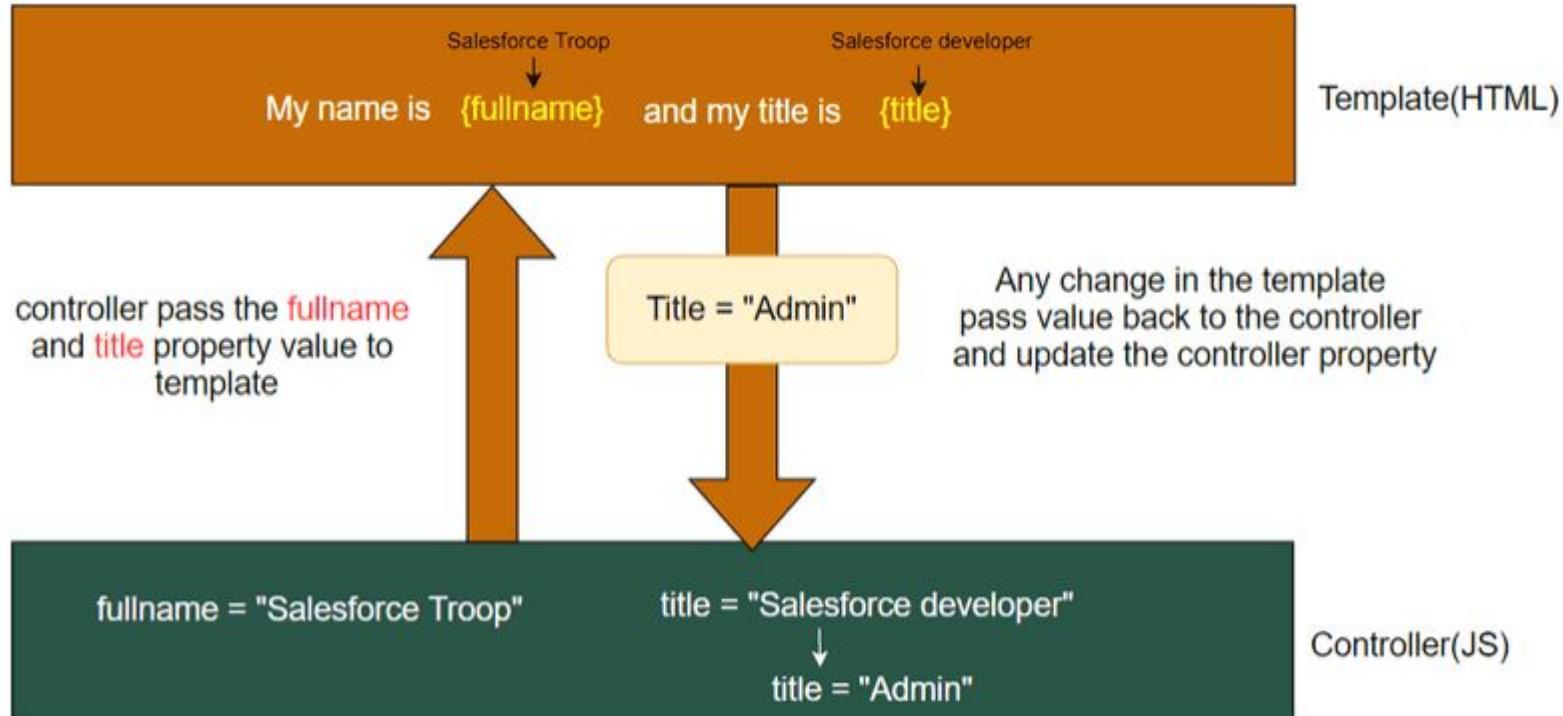
# Things to remember

- 1) In template we can access property value directly if it's primitive or object.
- 2) Dot notation is used to access the property from an object
- 3) LWC doesn't allow computed expressions like `Names[2]` or `{2+2}`
- 4) The property in `{ }` must be a valid JavaScript identifier or member expression. Like `{name}` or `{user.name}`
- 5) Avoid adding spaces around the property, for example, `{ data }`

# Methods And Two way Data Binding



# Two way Data Binding in a Template



# @track properties



# @track Property

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`.

## Normal Property vs @track property

Without using `@track`, the framework observes changes that assign a new value to a field/property. If the new value is not `==` to the previous value, the component re-renders.

# Getters



# What is Getter and When to use it

Not Allowed in Template

```
<div>users[0]</div>
<div>{num1 * num2}</div>
```

Javascript

```
users = ["John", "Mike", "Smith"]
num1 = 10
num2 = 20
```



Template

```
<div>{firstUser}</div>
<div>{sumOfNumbers}</div>
```

Javascript

```
//Example 1
users = ["John", "Mike", "Smith"]
get firstUser(){
    return this.users[0]
}

//Example 2
num1 = 10
num2 = 20
get sumOfNumbers(){
    return this.num1 * this.num2
}
```

# Conditional Rendering



# What is Directive

Directives are special HTML attributes. The LWC programming model has a few custom directives that let you manipulate the DOM using markup

In LWC we have two special directives for conditional rendering

1. *if:true*
2. *if:false*

# If:true directive

- Use this directive to render DOM elements in a template, if the expression is *true*

## Syntax

```
<template if:true={expression}>  
  Show me when expression is true  
</template>
```

 Copy

- The expression can be a JavaScript identifier(property)
- The expression can be a JavaScript dot notation that accesses a property from an object (user.fullName)
- You can't use ternary operator inside the expression
- To compute the value of the expression, use a getter in the JavaScript class.

# if:false directive

- Use this directive to render DOM elements in a template, if the expression is *false*

```
<template if:false={expression}>  
  Show me when the expression is false  
</template>
```

 Copy

- The expression can be a JavaScript identifier(property)
- The expression can be a JavaScript dot notation that accesses a property from an object (user.fullName)
- You can't use ternary operator inside the expression
- To compute the value of the expression, use a getter in the JavaScript class.

# Template Looping



# Template Looping

There are many scenarios in which we have to render the same set of elements with mostly same styling with different data in the HTML. To solve this issue, we have template looping in the LWC.

## *Template Looping Types*

1. for:Each
2. iterator

# for:each loop

Below is the syntax of the `for:each` loop

```
<template for:each={array} for:item="currentItem" for:index="index">  
    -----Here your repeatable template comes-----  
</template>
```

 Copy

Sno.	attributes	Description
1.	<code>for:each={array}</code>	<code>for:each</code> takes the <code>array</code> as an input
2.	<code>for:item="currentItem"</code>	<code>currentItem</code> is the value of the current element. <code>currentItem</code> is an alias and can be anymore. <code>for:item</code> holds the <code>currentItem</code> .
3.	<code>for:index="index"</code>	<code>index</code> is the current element index in the array. <code>for:index</code> holds the <code>index</code> .

# What is key and it's importance

- A `key` is a special string attribute you need to include to the first element inside the template when creating lists of elements.
- Keys help the LWC engine identify which items have changed, are added, or are removed.
- The best way to pick a key is to use a string that uniquely identifies a list item among its siblings.

Note - *Key must be a number or string, it can't be an object*

*You can't use the index as a value for the key*

# For:each Demo

# iterator loop

To apply a special behavior to the first or last item in a list we prefer `iterator` over `for:each`

Below is the syntax of the 'iterator' loop

```
<template iterator:iteratorName={array}>
    -----Here your repeatable template comes-----
</template>
```

 Copy

Sno.	attributes	Description
1.	iterator	It's a keyword to tell template that it's an iterator loop
2.	iteratorName	<code>iteratorName</code> is the value of the current element in the loop. <code>iteratorName</code> is an alias and can be anymore
3.	array	<code>array</code> is the data on which we want to apply loop

# Properties of iterator

Using iterator name you can access the following 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. *For example - iteratorName.index*

**first** — A boolean value indicating whether this item is the first item in the list. *For example - iteratorName.first*

**last** — A boolean value indicating whether this item is the last item in the list. *For example - iteratorName.last*

# Iterator Demo

# Component Composition



# Component Composition

*Composition* is Adding Component Within the body of another component

- Composition enables you to build complex components from simpler building-block components.

## ***How to refer child components name in parent components***

- |                       |   |
|-----------------------|---|
| 1. childComponent     | <c-child-component></c-child-component>           |
| 2. childComponentDemo | <c-child-component-demo></c-child-component-demo> |
| 3. sampleDemoLWC      | <c-sample-demo-l-w-c></c-sample-demo-l-w-c>       |

Replace capital letter with small letter and prefixed with hyphen

Try to avoid continuous capital letters in your component name.

# Shadow DOM

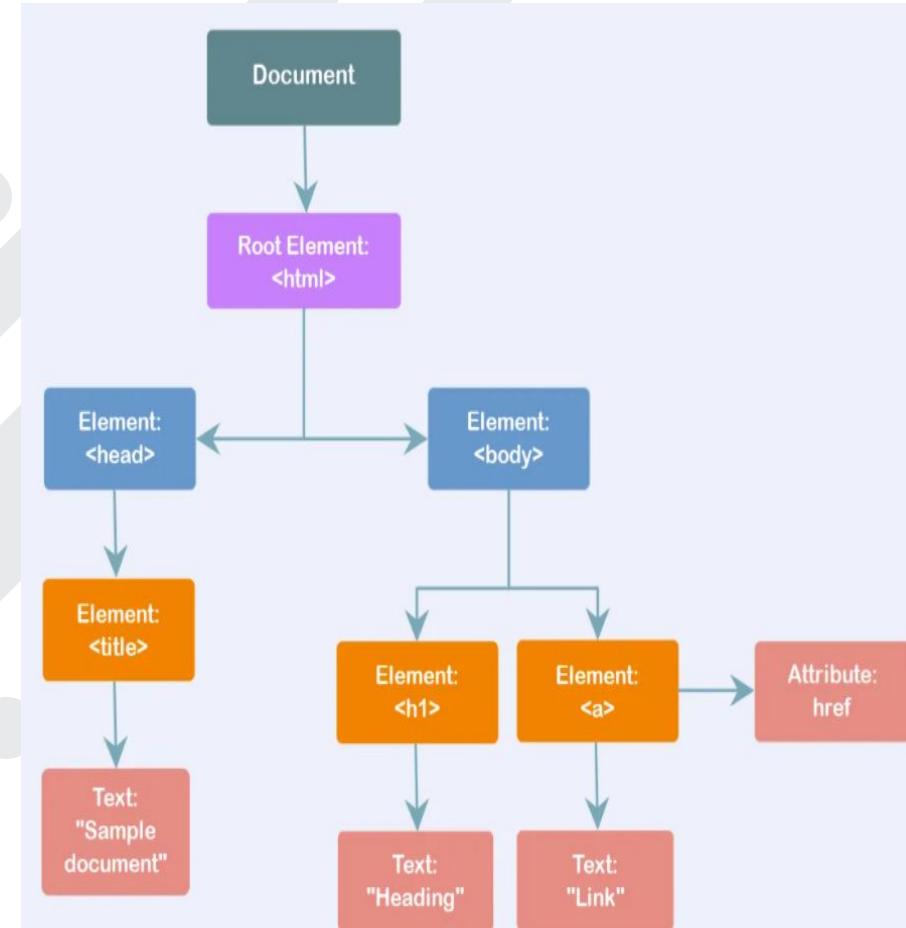


# What is DOM

**Document Object Model (DOM)** is a programming API for HTML and XML documents. It defines the logical structure of documents and the way a document is accessed and manipulated.

Basically, The DOM is a tree structure representation of web page.

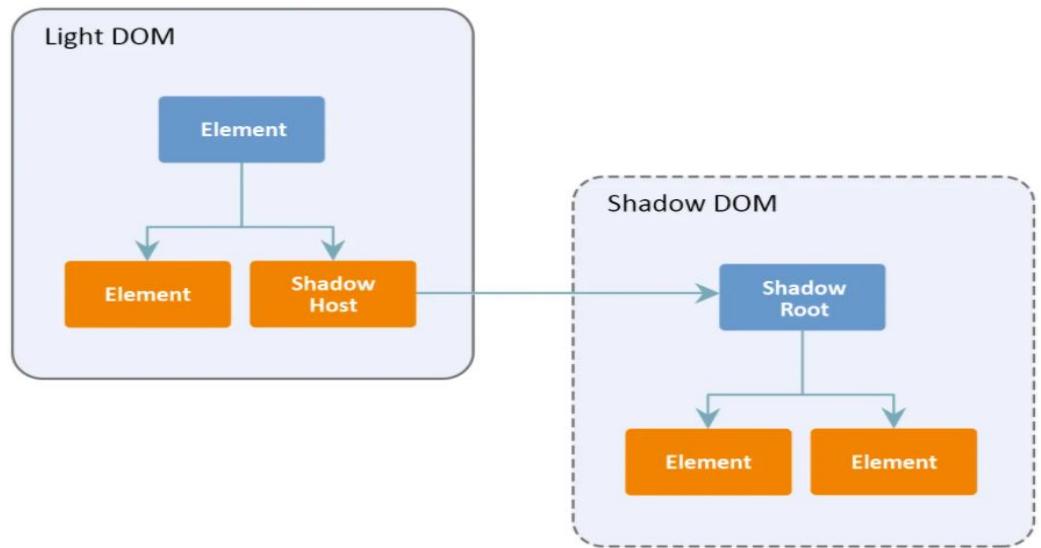
```
<html>
  <head>
    <title>Zero to Hero</title>
  </head>
  <body>
    <h1>Heading</h1>
    <a href="https://salesforcetroop.com">Link</a>
  </body>
</html>
```



# What is Shadow DOM

Shadow DOM brings encapsulation concept to HTML which enables you to link a hidden separated DOM to an element.

**Benefits of Shadow DOM-** DOM queries, event propagation and CSS rules cannot go to the other side of the shadow boundary, thus creating encapsulation.



# Accessing Elements in the Component



# Accessing Elements in LWC

To access elements rendered by a component, use the `template` property.

```
this.template.querySelector(selector);  
this.template.querySelectorAll(selector);  
element.template.querySelectorAll(selector);
```

Note\*\* - Don't use ID selector with `querySelector`

**`lwc:dom="manual"`**

Add this directive to a native HTML element to attach an HTML element as a child.

# QUIZ APP IN LWC



# Things we gonna learn and use

1. Component Creation
2. If:true directive
3. Looping -for:each
4. Getter
5. Methods
6. Properties
7. Form submit and reset
8. Html layout

# CSS in LWC



# CSS in LWC

- 
1. **Inline Styling**
  2. **External CSS**
  3. Using Lightning Design System
  4. SLDS Design Token
  5. Shared CSS in LWC
  6. Dynamic Styling
  7. Third-party css library
  8. Applying CSS across shadow DOM

# Lightning Design System



# Lightning Design System

The Salesforce Lightning Design System includes the resources to create user interfaces consistent with the Salesforce Lightning principles, design language, and best practices.

<https://www.lightningdesignsystem.com/>

Alignment

Borders

Box

Description List

FLOATS

Grid

Horizontal List

Hyphenation

Interactions

Layout

Line Clamp

Margin

Media Objects

Name Value List

Padding

Position

Print

Scrollable

Sizing

Position

Print

Scrollable

Sizing

Text

Themes

Truncation

Vertical List

Visibility

# Lightning Design System Design Tokens



# CSS in LWC

- 
1. Inline Styling
  2. External CSS
  3. Lightning Design System
  - 4. SLDS Design Token**
  5. Shared CSS in LWC
  6. Dynamic Styling
  7. Third-party css library
  8. Applying CSS across shadow DOM

# Lightning Design System Design Tokens

Design tokens are named entities that store visual design attributes, such as margins and spacing values, font sizes and families, or hex values for colors.

# Shared CSS in LWC



# CSS in LWC

- 
1. Inline Styling
  2. External CSS
  3. Lightning Design System
  4. SLDS Design Token
  - 5. Shared CSS in LWC**
  6. Dynamic Styling
  7. Third-party css library
  8. Applying CSS across shadow DOM

# Dynamic Styling



# CSS in LWC

- 
1. Inline Styling
  2. External CSS
  3. Lightning Design System
  4. SLDS Design Token
  5. Shared CSS in LWC
  - 6. Dynamic Styling**
  7. Third-party css library
  8. Applying CSS across shadow DOM

# CSS Behaviour in Parent Child Component



# CSS Behaviour in Parent Child Component

1. Parent CSS can't reach into a child component.
2. A parent component CSS can style a host element(<c-css-child>).
3. A Child Component CSS can reach up and style its own host element
4. You can style to a element pass into the slot from the parent component only

# CSS in Parent Child Component

A component's style sheet can reach up and style its own host element

```
/* cssChild.css */  
:host {  
    display: block;  
    border: 2px solid red;  
}
```

By targeting the host element with `:host`, we've applied styling to `<c-css-child>`, from `cssChild.css`.

Styling to element pass in the slot

You can style to element pass to a slot from the parent css file only

# Component Lifecycle Hooks



# LWC lifecycle Hooks

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

## Mounting Phase

constructor()

connectedCallback()

render()

renderedCallback()

## Unmounting Phase

disconnectedCallback()

## Error Phase

errorCallback()

# Lifecycle Hooks in Mounting Phase



# Lifecycle Hooks in Mounting Phase

constructor()

connectedCallback()

renderedCallback()

# Constructor Method

## Points to remember

- This hook is invoked, when a component instance is created.
- You have to call super() first inside this
- At this point, the component properties won't be ready yet.
- To access the host element, use this.template
- This method lifecycle flows from Parent to Child component.
- we can't access child elements in the component body because they don't exist yet.
- Don't add attributes to the host element in the constructor

# connectedCallback Method

## Points to remember

- Called when the element is inserted into a document.
- This hook flows from parent to child.
- You can't access child elements in the component body because they don't exist yet.
- You can access the host element with `this.template`.
- **use it to:** Perform initialization tasks, such as fetch data, set up caches, or listen for events (such as publish-subscribe events)
- **Do not use:** `connectedCallback()` to change the state of a component, such as loading values or setting properties. Use getters and setters instead

# renderedCallback Method

## Points to remember

- Fires when a component rendering is done.
- It can fire more than once.
- This hook flows from child to parent.
- When a component re-renders, all the expressions used in the template are reevaluated.

## Do not use renderedCallback()

1. to change the state or update property of a component
2. Don't update a wire adapter configuration object property in renderedCallback(), as it can result in an infinite loop.

# Lifecycle Hooks in Unmounting Phase



# disconnectedCallback Method

## Points to remember

- Fires when a component is removed from the DOM.
- It flows from parent to child
- This callback method is specific to Lightning Web Components, it isn't from the HTML custom elements specification.

# Lifecycle Hooks in Error Phase



# errorCallback(err,stack) Method

## Points to remember

- This method is called when a descendant component throws an error in one of its callbacks.
- The error argument is a JavaScript native error object, and the stack argument is a string.
- This callback method is specific to Lightning Web Components, it isn't from the HTML custom elements specification

# Render Method

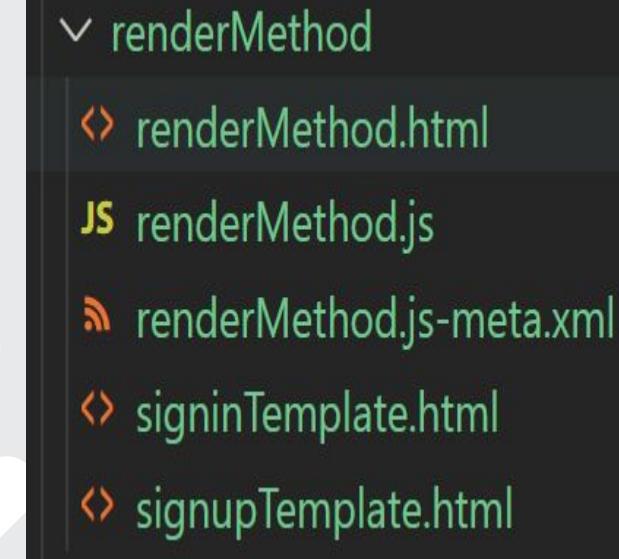


# render Method

Render is a method that tells the component which template to load based on some conditions. It always return the template reference

## Points to remember

- The `render()` method is not technically a lifecycle hook. It is a protected method on the ***LightningElement*** class.
- Call this method to update the UI. It may be called before or after `connectedCallback()`
- It's rare to call `render()` in a component. The main use case is to conditionally render a template.



```
renderMethod
├── renderMethod.html
└── renderMethod.js
  └── renderMethod.js-meta.xml
    ├── signinTemplate.html
    └── signupTemplate.html
```

# When to prefer multiple template over **if:true/if:false**

- **if:true/if:false** is recommended whenever there is small template to hide and show
- Ideally it's always recommended to break down your component into smallest unit.
- Whenever we have a scenario in which we have same business logic but we want to render a component with more than one look and feel.
- Whenever we have two designs in same component but not want to mix the HTML in one file.

# Components Communication



# Components Communication Approaches

Parent To Child Communication

Child To Parent Communication

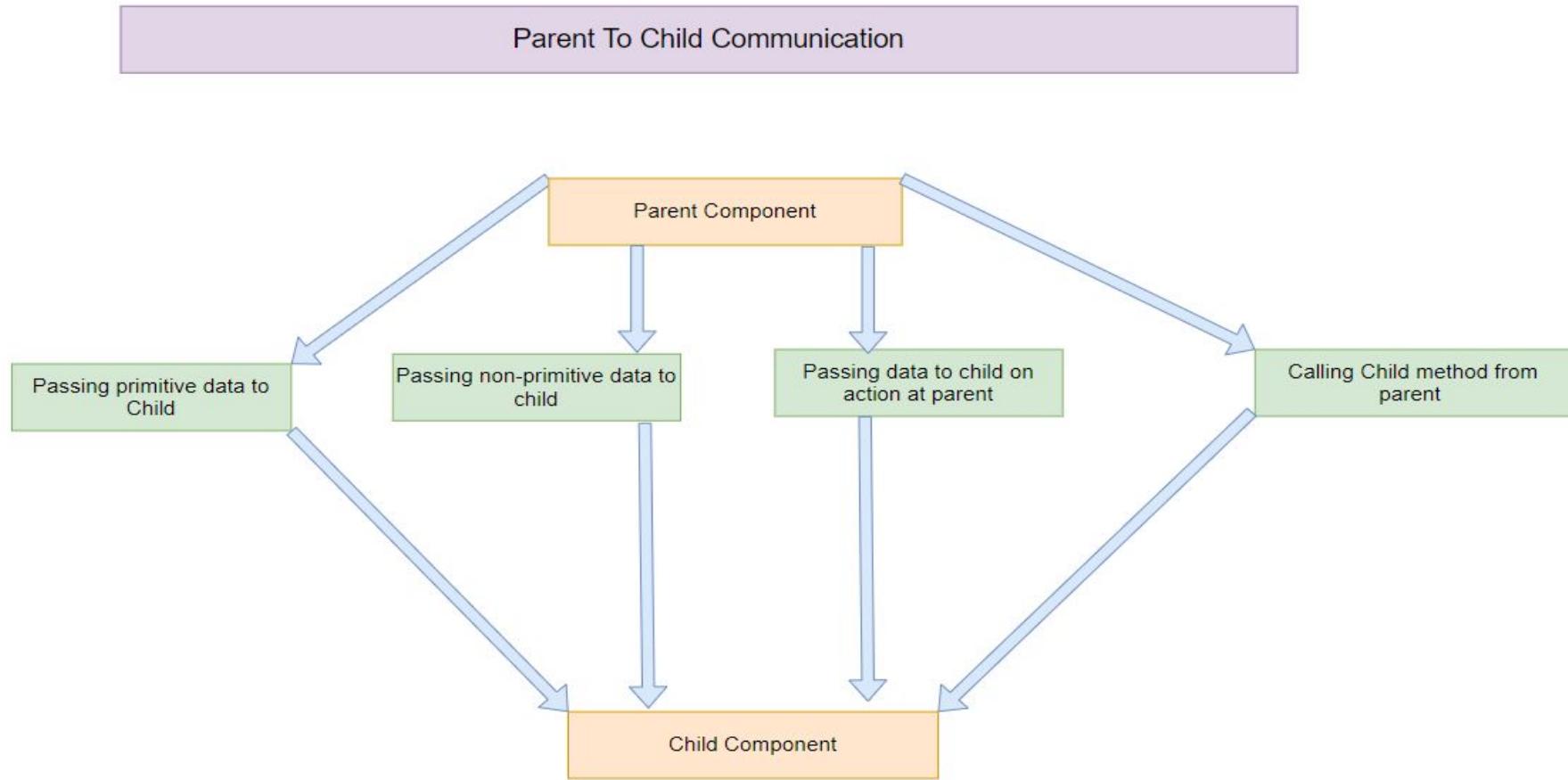
Sibling Component Communication Using PubSub

Communication Across VF pages, Aura and LWC Using Lightning Messaging Service

# Parent to Child Communication Approaches



# Parent to Child Communication



# Parent To Child Communication using primitive Data type

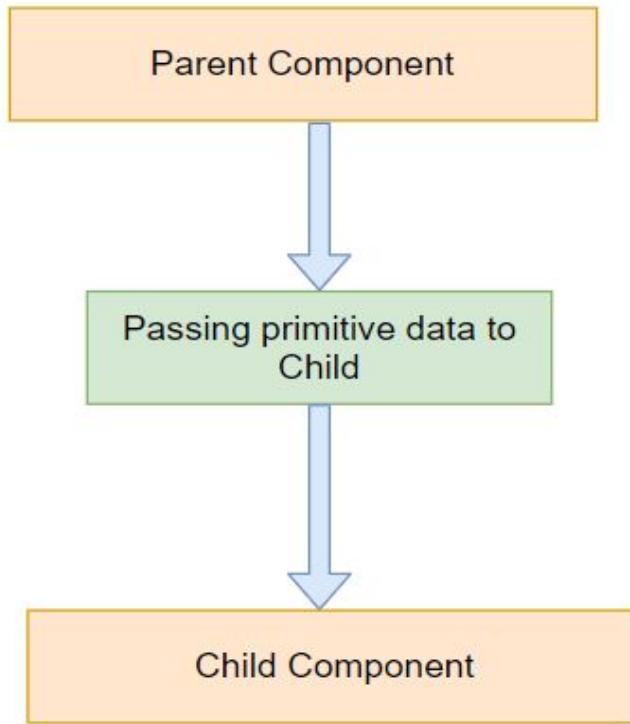


# @api decorator

1. To make a field/property or method public, decorate it with @api decorator
2. When we want to expose the property we decorate the field with @api.
3. An owner component that uses the component in its HTML markup can access the component's public properties via HTML attributes.
4. Public properties are reactive in nature and if the value of the property changes the component's template re-renders.

```
import { LightningElement, api } from 'lwc';
export default class Child extends LightningElement {
    fullname
    |
    @api username
}
```

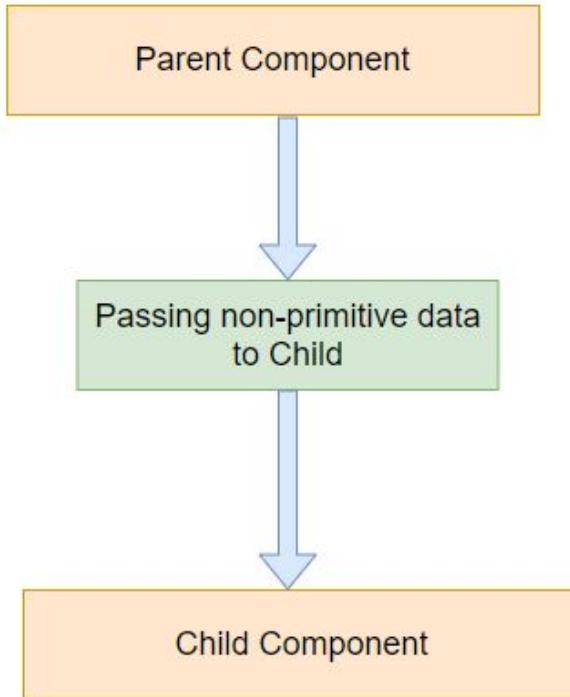
## Case 1 - Simple data passing



# Parent To Child Communication using non-primitive Data type



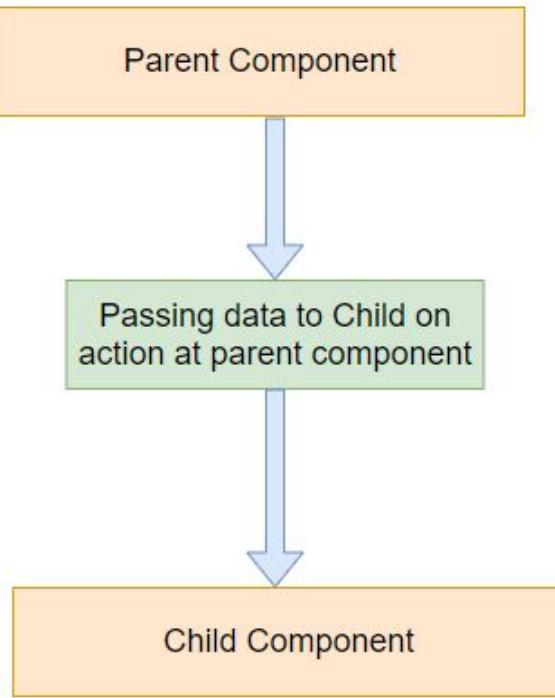
## Case 2 - Complex data passing



# Parent To Child Communication on action at parent component



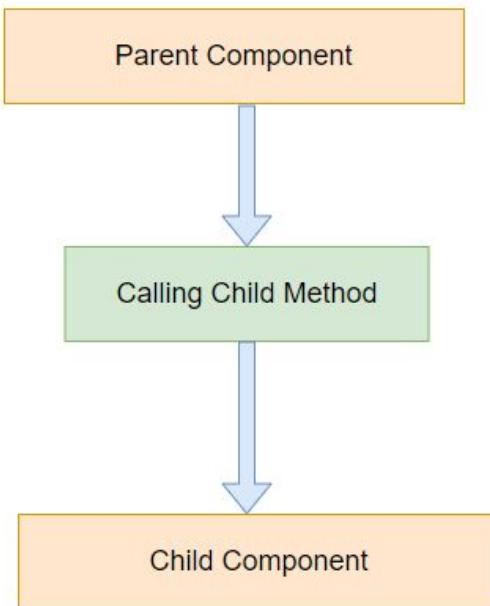
## Case 3 - Passing data To child on action at Parent Component



# Calling Child Method from Parent component



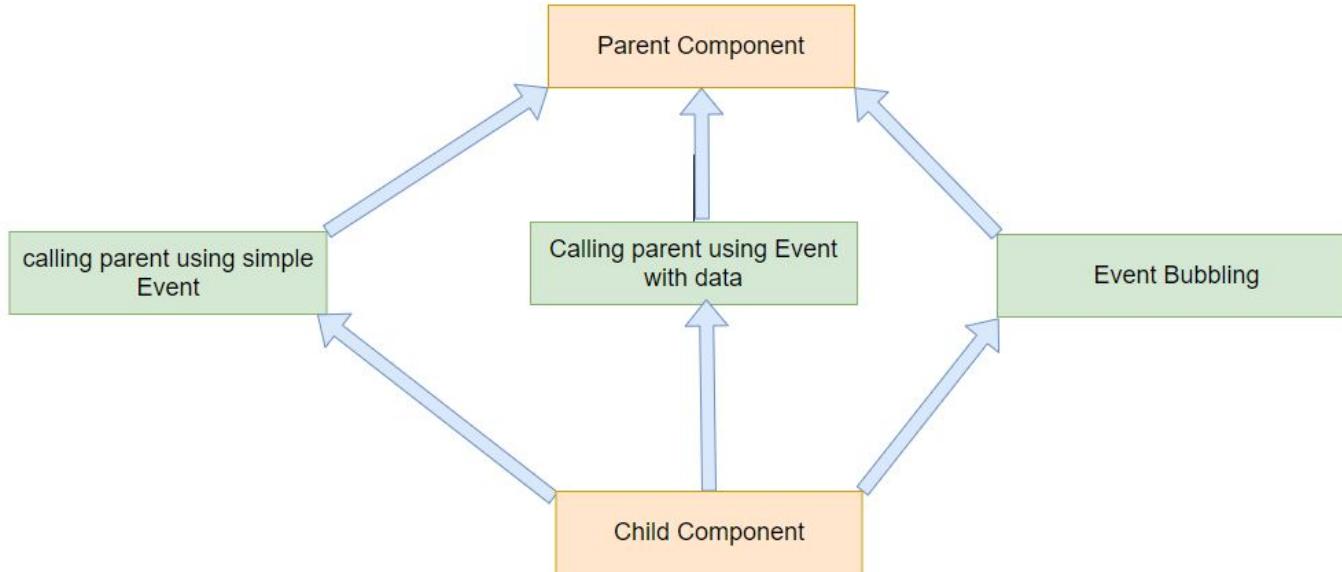
## Case 4 - Calling Child Method from parent Component



# Child to Parent Communication Approaches



## Child to Parent Communication



# Child to Parent Communication Using Simple Event



# Create and Dispatch Events

## CustomEvent Syntax

```
new CustomEvent('event name', { bubbles:false, composed:false, details:null})
```

These are the default values

```
new CustomEvent('show', { bubbles:true, details:'hello'})
```

default set to false

- 1) Event name is defined as 'show'
- 2) by default bubble and composed is false but explicitly we have made bubbles true for event bubbling
- 3) details is used to pass data with event and here i am pass "hello"

## dispatchEvent Syntax

```
EventTarget.dispatchEvent(eventcreated)
```

```
this.dispatchEvent(new CustomEvent('show', { bubbles:true, details:'hello'}))
```

- 1) 'this' is the eventTarget
- 2) dispatchEvent trigger the CustomEvent

**Event Name Naming Conventions-** Only String, No uppercase letters, No spaces, Use underscores to separate words

# Child to Parent Communication Using Event with Data



# Child to Parent Communication Using Event Bubbling

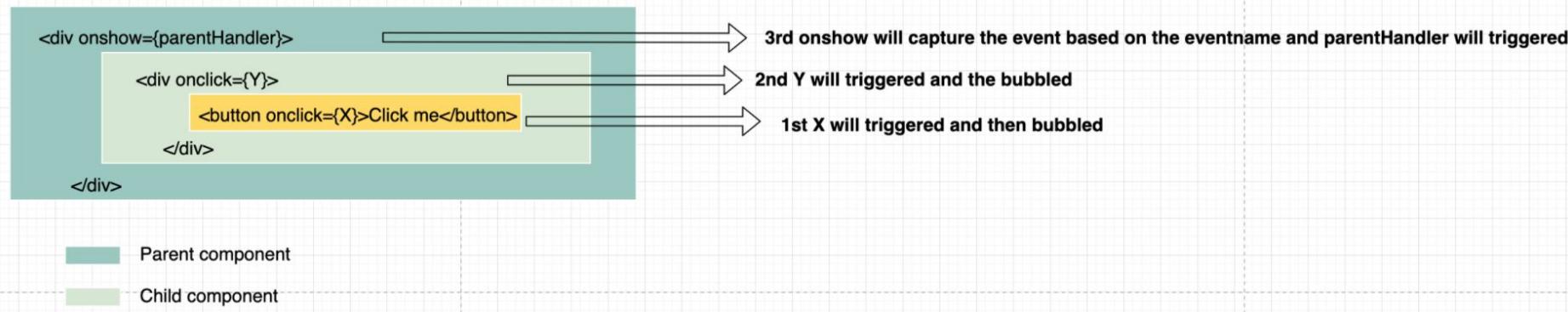


# Event Bubbling

Lightning web component has only two event propagation

- **bubbles** - A Boolean value indicating whether the event bubbles up through the DOM or not. Defaults to **false**.
- **composed** - A Boolean value indicating whether the event can pass through the shadow boundary. Defaults to **false**.

## Event Bubbling Demo



```
this.dispatchEvent(new CustomEvent('show', { bubbles:true, details:'hello'}))
```

# Component Communication Using PubSub Module

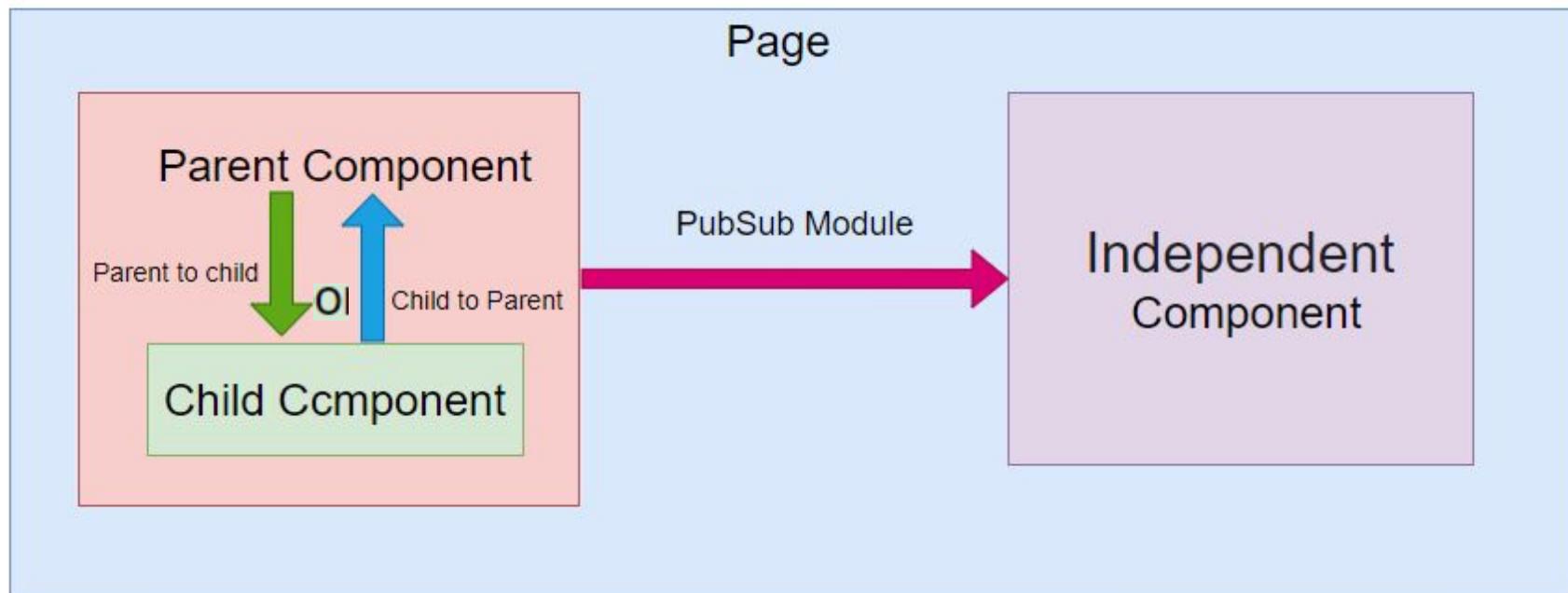


# Component Communication Using PubSub

There are two ways to communicate between Independent Components

- 1) pubsub
- 2) Lightning Messaging Service

**NOTE \*\*** Use this approach, if Lightning Messaging Service not serve your purpose. It's an Old technique to communicate with the independent components in LWC



# Method Names

- 1) Publish Method - This method is use to publish an Event

```
pubsub.publish('event name', dataToPass);
```

- 2) Subscribe Method - This method is use to subscribe an Event

```
pubsub.subscribe('event name', callback);
```

- 3) Unsubscribe Method - This method is use to unsubscribe the subscribed event.

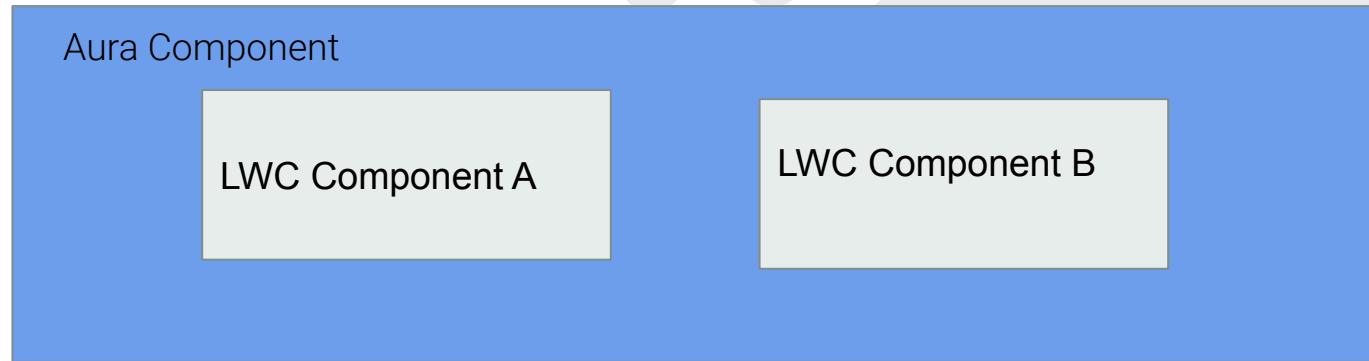
```
pubsub.unsubscribe('event name', callback)
```

# Aura Coexistence



# Aura Coexistence

1. Lightning Web component and Aura component can work together.
2. Aura components can contain Lightning web components. However, the opposite doesn't apply. *Lightning web components can't contain Aura components.*



# Setter Method



# Setter Method

This method is used to modify the data coming from parent component.

If Object is passed as data to setter, to mutate the object we have to create a shallow copy.

```
@api
get detail(){
    return this.userObject
}

set detail(data){
    let newData = //mutation of logic
    this.userObject = newData
}
```

# Passing Markup Using Slots



# Passing Markup into Slots

1. Slot is useful to pass HTML markup into the another component.
2. <slot></slot> markup is used to catch the HTML passed by parent component
3. You can't pass an Aura component into a slot

**There are two types of Slots**

Named Slots

When name attribute is defined in slot element `<slot name="user"></slot>`

Unnamed Slots

When a slot without a name attribute `<slot></slot>`

# Building Reusable Component



# Applying CSS Across Shadow DOM



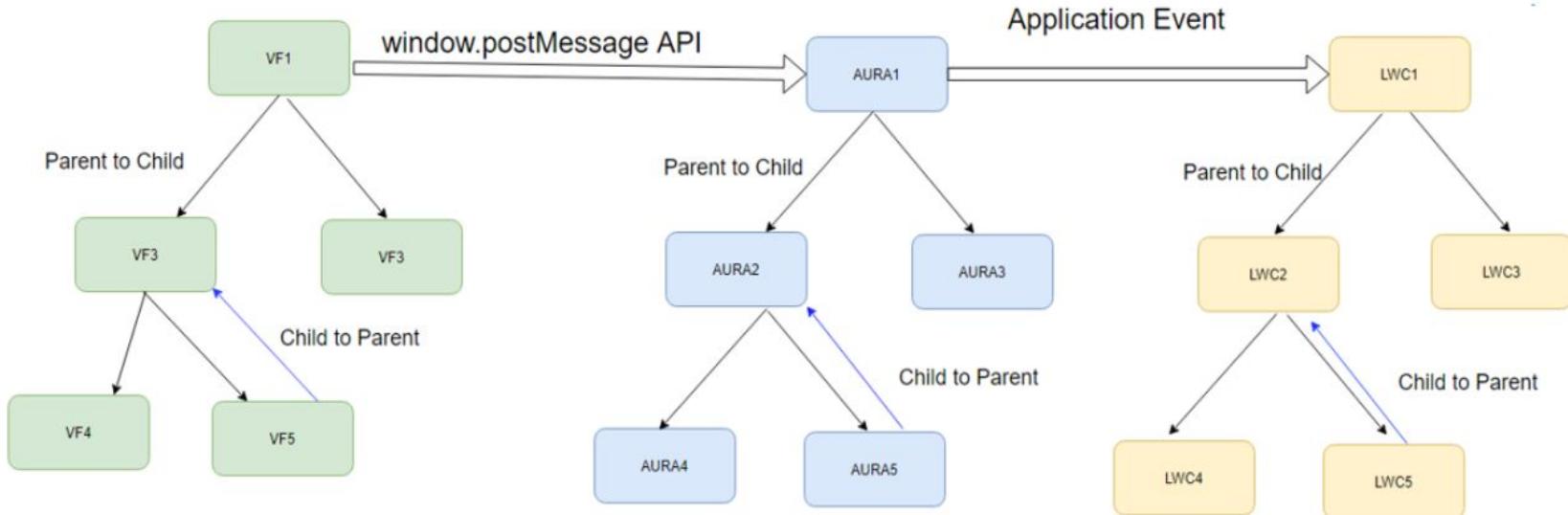
# Lightning Messaging Service



# Lightning Messaging Service

LMS is the first Salesforce technology which enables you to communicate between Visualforce, Aura component and Lightning web component anywhere in lightning pages including utility components.

In Winter 20, Salesforce is released "Lightning Message Service" (LMS)

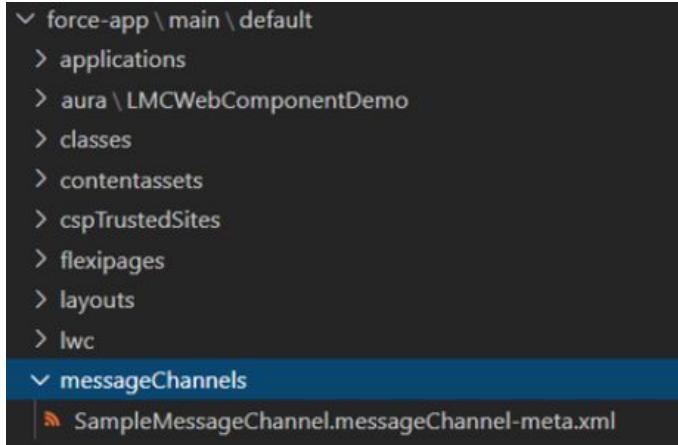


# Prerequisite to Create the Message Channel

1. Create a folder called *messageChannels* under the path *force-app\main\default*

2. Create a file *CHANNELNAME.messageChannel-meta.xml* inside the folder

Example - SampleMessageChannel.messageChannel-meta.xml



## Prerequisite cont..

3. Add following XML definition to message channel file

```
<?xml version="1.0" encoding="UTF-8"?>
<LightningMessageChannel xmlns="http://soap.sforce.com/2006/04/metadata">
    <description>This is a sample Lightning Message Channel.</description>
    <isExposed>true</isExposed>
    <lightningMessageFields>
        <description>Variable used for lms data</description>
        <fieldName>lmsData</fieldName>
    </lightningMessageFields>
    <masterLabel>SampleMessageChannel</masterLabel>
</LightningMessageChannel>
```

4. Update the manifest/package.xml file by adding the *LightningMessageChannel*

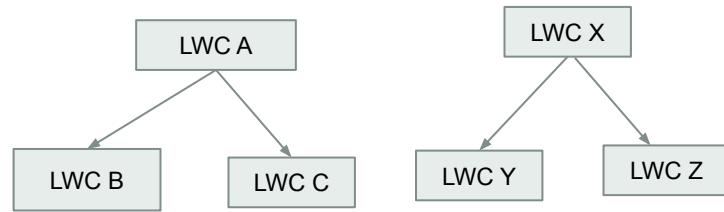
```
<types>
    <members>*</members>
    <name>LightningMessageChannel</name>
</types>
```

5. API Version should be above 47 & above

# LWC TO LWC Communication using LMS



# LWC TO LWC Communication using LMS



1. Reference a message channel in LWC

```
import SAMPLEMC from "@salesforce/messageChannel/SampleMessageChannel__c"
```

2. Import LMS API

```
import { APPLICATION_SCOPE, publish, subscribe, unsubscribe, MessageContext } from 'lightning/messageService';
```

3. MessageContext Wire Adapter is used to get information of all LWC using LMS

```
@wire(MessageContext)  
context;
```

# LWC TO LWC Communication using LMS

## 4. Publish a message

```
//publish(messageContext, messageChannel, message)
publish(this.context, SAMPLEMC, message)
//this.context is the MessageContext object
//SAMPLEMC is the reference of the Message Channel
//message is the data to publish
```

## 5. Subscribe a Message channel to receive the message

```
//subscribe(messageContext, messageChannel, listener, subscriberOptions)
this.subscription = subscribe(this.context, SAMPLEMC, (message)=>{this.handleMessage(message)}, { scope: APPLICATION_SCOPE})
//this.context is the context of the current lightning web component using LMS
//SAMPLEMC is the reference of the Message Channel
//listener is a function to receive messages on a message channel from anywhere in the application,
//pass the subscriberOptions parameter as {scope: APPLICATION_SCOPE}
```

## 6. Unsubscribe to a Message channel

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

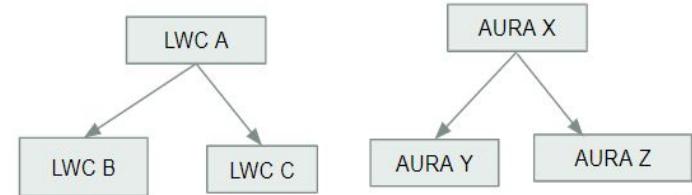
# LWC TO AURA Communication using LMS



# LWC TO AURA Communication using LMS

1. Add lightning:messageChannel component to Aura Component

```
<lightning:messageChannel  
type="SampleMessageChannel__c"  
aura:id="SampleMessageChannel"  
onMessage="{!c.handleMessage}"  
scope="APPLICATION"/>
```



2. Specify a controller action that read the message argument to get the values in the message payload

```
handleMessage: function (component, message, helper) {  
    if (message != null && message.getParam("lmsData") != null){  
        component.set("v.messageRecieved", message.getParam("lmsData").value)  
    }  
}
```

3. To Publish a message find the reference of the message channel and call the publish method by passing the payload

```
component.find("SampleMessageChannel").publish(message)
```

# LWC, AURA and VisualForce Page Communication using LMS



# LWC TO VisualForce Page Communication using LMS

1. Get the message channel reference using \$MessageChannel Global Variable

```
let SAMPLEMC = "{$!$MessageChannel.SampleMessageChannel__c}"
```

2. Publish a Message

```
sforce.one.publish(SAMPLEMC, payload)
```

3. Subscribe a Message

```
subscriptionToMC = sforce.one.subscribe(SAMPLEMC, msgHandler, {scope:"APPLICATION"})
```

4. unsubscribe a Message

```
sforce.one.unsubscribe(subscriptionToMC)
```

# Component Composition Extra Features



Setter Method

Passing markup using  
slots

CSS Behaviour in  
composition

# Salesforce Resources, Component Context and Notification



# Salesforce Resources, Component Context and Notification

Images from static resources

Using Third Party Js

Using Third party CSS Library

Access Content Asset Files

Internationalization

Access Labels

Permission Check

Get Current User Information

Access Client Form Factor

Fetch Record Id and Object Name

Toast Notifications

# Images from Static Resources



# Static Resources

Import static resources from the `@salesforce/resourceUrl` scoped module. Static resources can be archives (such as **.zip** and **.jar** files), **images**, **style sheets**, **JavaScript**, and **other files**.

```
import myResource from '@salesforce/resourceUrl/resourceReference';
```

*myResource* –A name that refers to the static resource.

*resourceReference* –The name of the static resource.

A static resource name can contain only underscores and alphanumeric characters, and must be unique in your org. It must begin with a letter, not include spaces, not end with an underscore, and not contain two consecutive underscores.

# Third Party JavaScript Libraries in LWC



# Third-Party JavaScript Libraries in LWC

## Syntax

Import the static resource.

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

Import methods from the `platformResourceLoader` module.

```
import { loadScript } from 'lightning/platformResourceLoader';
```

`loadScript(Reference to the component, fileUrl): Promise`

# Using Third Party CSS Library



# CSS in LWC

- 
1. Inline Styling
  2. External CSS
  3. Lightning Design System
  4. SLDS Design Token
  5. Shared CSS in LWC
  6. Dynamic Styling
  - 7. Third-party css library**
  8. Applying CSS across shadow DOM

# Use Third-Party CSS Libraries

## Syntax

Import the static resource.

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

Import methods from the `platformResourceLoader` module.

```
import { loadStyle} from 'lightning/platformResourceLoader';
```

`loadStyle(Reference to the component, fileUrl): Promise`

# Content Asset Files



# Content Asset Files

Import content asset files from the `@salesforce/contentAssetUrl` scoped module. Convert a Salesforce file into a content asset file to use the file in custom apps and Community templates.

## Syntax

```
import myContentAsset from '@salesforce/contentAssetUrl/contentAssetReference';
```

# Internationalization Properties



# Internationalization Properties

Lightning web components have internationalization properties that you can use to adapt your components for users worldwide, across languages, currencies, and timezones.

In a single currency organization, Salesforce administrators set the currency locale, default language, default locale, and default time zone for their organizations. Users can set their individual language, locale, and time zone on their personal settings pages.

## Syntax

```
import internationalizationPropertyName from '@salesforce/i18n/internationalizationProperty'
```

INTERNATIONALIZATION PROPERTY	DESCRIPTION	SAMPLE VALUE
lang	Language	en-US
dir	Direction of text	ltr
locale	Locale	en-CA
currency	Currency code	CAD
firstDayOfWeek	First day of the week	1
dateTime.shortDateFormat	Short style date format	MM/dd/yyyy
dateTime.mediumDateFormat	Medium style date format	MMM d, yyyy
dateTime.longDateFormat	Long style date format	MMMM d, yyyy
dateTime.shortDateTimeFormat	Short style datetime format	MM/dd/yyyy h:mm a
dateTime.mediumDateTimeFormat	Medium style datetime format	MMM d, yyyy hh:mm:ss a
dateTime.shortTimeFormat	Short style time format	h:mm a
dateTime.longTimeFormat	Long style time format	h:mmss a
number.currencyFormat	Currency format	, ##0.00
number.currencySymbol	Currency symbol	\$
number.decimalSeparator	Decimal separator	.
number.groupingSeparator	Grouping separator	,
number.numberFormat	Number format	, ##0.###
number.percentFormat	Percent format	, ###0%
timeZone	Time zone	America/Los_Angeles

# Access Labels



# Labels

Import labels from the `@salesforce/label` scoped module.

Custom labels are text values stored in Salesforce that can be translated into any language that Salesforce supports. Use custom labels to create multilingual applications that present information (for example, help text or error messages) in a user's native language.

## Syntax

```
import LabelName from '@salesforce/label/LabelReference';
```

Note \*\* - The name of the label in your org in the format `namespace.labelName`

# Check Permissions



# Check Permissions

Import Salesforce permissions from the `@salesforce/userPermission` and `@salesforce/customPermission` scoped modules.

## Syntax

```
import hasPermission from '@salesforce/userPermission/PermissionName'  
  
import hasPermission from '@salesforce/customPermission/PermissionName'
```

# Access Client Form Factor



# Access Client Form Factor

To access the form factor of the hardware the browser is running on, import the `@salesforce/client/formFactor` scoped module.

## Syntax

```
import formFactorPropertyName from '@salesforce/client/formFactor'
```

## Possible Values are

Large –A desktop client.

Medium –A tablet client.

Small –A phone client.

# Get Information About the Current User



# Get Information About the Current User

To get information about the current user, use the `@salesforce/user` scoped module.

## Syntax

```
import property from '@salesforce/user/property';
```

`property` supports only two properties

- 1) `Id` - User's ID
- 2) `isGuest` - Boolean value indicating whether the user is a community guest user or not

# Fetch Record Id and Object Name



# Fetch Record Id and Object Name

**@api recordId** - If a component with a **recordId** property is used on a Lightning record page, the page sets the property to the ID of the current record.

**@api objectApiName** - If a component with an **objectApiName** property is used on a Lightning record page, the page sets the property to the API name of the current object.

# Toast Notification



# Toast Notifications

Toast is a popup that alert user with some information. Toast can be of success, error, info or warning.

```
import { ShowToastEvent } from 'lightning/platformShowToastEvent';
const evt = new ShowToastEvent({
    title: "toast title",
    message: "toast message",
    variant: "toast variant" //(success, info, warning or error),
});
this.dispatchEvent(evt);
```

`messageData` - url and label values that replace the {index} placeholders in the message string.

**Mode** - Determines how persistent the toast is. Valid Values ares

- 1) **dismissable**(default) - Remains visible until the user clicks the close button or 3 seconds has elapsed, whichever comes first.
- 2) **pester** - visible for 3 seconds
- 3) **sticky** - remain visible until the user clicks the close button

# Component configuration in LWC



# Component Configuration file Tag (.js-mext.xml)

The configuration file defines the metadata values for the component, including the design configuration for the Lightning App Builder and Experience Builder.

Sno	Tags	Description
1	<code>apiVersion</code>	Salesforce API version 45.0 or higher.
2	<code>description</code>	A short description of the component, usually a single sentence. The description appears in list views, like the list of Lightning Components in Setup, and as a tooltip in Lightning App Builder and Experience Builder.
3	<code>isExposed</code>	If <code>isExposed</code> is <code>false</code> , 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 <code>isExposed</code> to <code>true</code> and define at least one <code>&lt;target&gt;</code> , which is a type of Lightning page.
4	<code>masterLabel</code>	The title of the component. Appears in list views, like the list of Lightning Components in Setup, and in the Lightning App Builder and in Experience Builder.

# Component Configuration file Tag

Sno	Tags	Description
5	<b>targets</b>	Specifies where the component can be added, such as on a type of Lightning Page or in Embedded Service Chat. It support <b>target</b> subtag
6	<b>target</b>	A lightning page type. Valid values are <code>lightning_AppPage, lightning_HomePage, lightning_RecordPage, lightning_UtilityBar, lightning_FlowScreen, lightning_Tab, lightning_Inbox, lightningCommunity_Page, lightningCommunity_Default, lightningSnapin_ChatMessage, lightningSnapin_Minimized, lightningSnapin_PreChat</code>
7	<b>targetConfigs</b> and <b>targetConfig</b>	Configure the component for different page types and define component properties.use targetConfig for each different page type configuration. The <b>targetConfig</b> tag supports the <b>property</b> , <b>propertyType</b> , <b>objects</b> , and <b>supportedFormFactors</b> subtags.
8	<b>property</b>	Specifies a public property of a component that can be set in Lightning App Builder, App Manager, Lightning Flow Builder, or Experience Builder. The component author defines the property in the component's JavaScript class using the <code>@api</code> decorator.
9	<b>Objects and object</b>	Object is subtag of objects. Object tag allow you to expose component to specific object

# Component Configuration file Tag

The `property` tag supports these attributes:

Sno	attribute	Description
1	<code>datasource</code>	Renders a field as a picklist, with static values.
2	<code>label</code>	Displays as a label for the attribute in the tool.
3	<code>Max</code> and <code>min</code>	The maximum and minimum allowed value for an attribute of type <code>Integer</code>
4	<code>name</code>	Required if you're setting properties for your component. The attribute name. This value must match the property name in the component's JavaScript class.
5	<code>required</code>	Specifies whether the attribute is required. The default value is <code>false</code> .
8	<code>type</code>	<p>The attribute's data type. These values are valid for all targets.</p> <ul style="list-style-type: none"><li>• <code>Boolean</code></li><li>• <code>Integer</code></li><li>• <code>String</code></li><li>• <code>Color</code></li></ul>

# LWC in lightning Tab



# LWC in Utility Bar



# Navigation Service



# Types of Navigations we gonna Learn

Navigate to Home Page

Navigate to Chatter

Navigate to New Record

Navigate To New Record With Default Values

Navigate to List View

Navigate to Files

Navigate to Record Page in View and Edit Mode

Navigate to Tab

Navigate to Record Relationship Page

Navigate to External Web Page

Navigate to LWC Component

Navigate to AURA Component

Navigate to VF pages

Fetch Current Page Reference

# Steps to Use Navigation Service

In the component's JavaScript class, import the lightning/navigation module

```
import { NavigationMixin } from 'lightning/navigation';
```

Apply the NavigationMixin function to your component's base class

```
export default class MyCustomElement extends NavigationMixin(LightningElement) {}
```

*To dispatch the navigation request, call the navigation service's*

**[NavigationMixin.Navigate](pageReference, [replace])**

**pageReference**- It is an object that defines the page

**Replace** - It's a boolean values which is false by default. If this value is **true** it pageReference replaces the existing entry in the browser history.

# PageReference Types

To navigate in Lightning Experience, Lightning communities, or the Salesforce mobile app, define a PageReference object.

These page reference types are supported.

- App
- Lightning Component
- Knowledge Article
- Login Page
- Named Page (Communities)
- Named Page (Standard)
- Navigation Item Page
- Object Page
- Record Page
- Record Relationship Page
- Web Page

# Navigate To Home

Type - Named Pages



# Navigate To Chatter

Type - Named Pages



# Navigate To New Record

Type - Object Pages



# Navigate To New Record With Default Values

Type - Object Page



# Navigate To List View

Type - Object Page



# Navigate To Files

Type - Object Page



# Navigate To Record Page in View and Edit Mode

Type - Record Page



# Navigate To Tab

Type - nav Item Page



# Navigate To Record Relationship Page

Type - record relationship page



# Navigate To External Web Page

Type - web page



# Navigate To LWC Page

Type - web page



# Navigate To AURA Component

Type - component



# Navigate To VF Page

Type - web page



# Fetch Current Page Reference



# Lightning Message Service



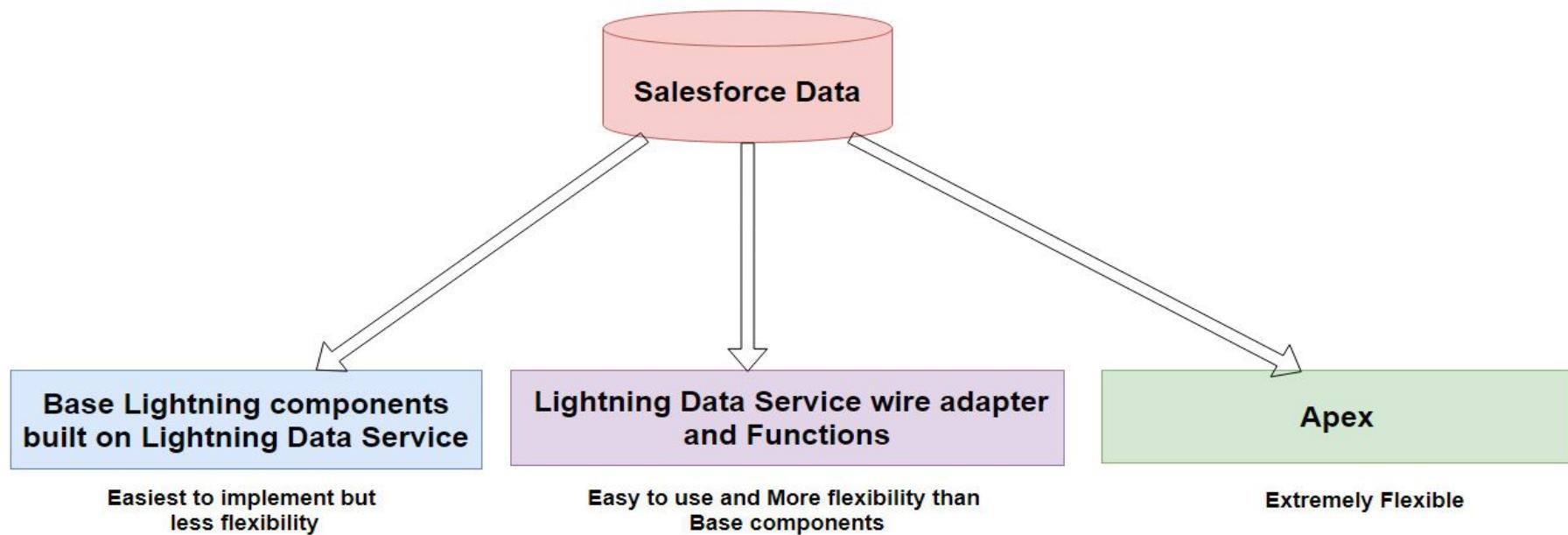
# Work With Data In LWC



# Work With Data in Lightning Web Components

There are many ways of interacting with Salesforce data in the Lightning web components.

Knowing which approach to use for a particular use case helps you to write less code, easier code, and code that is more maintainable. The efficiency of your components is improved by using the best solution for each situation.

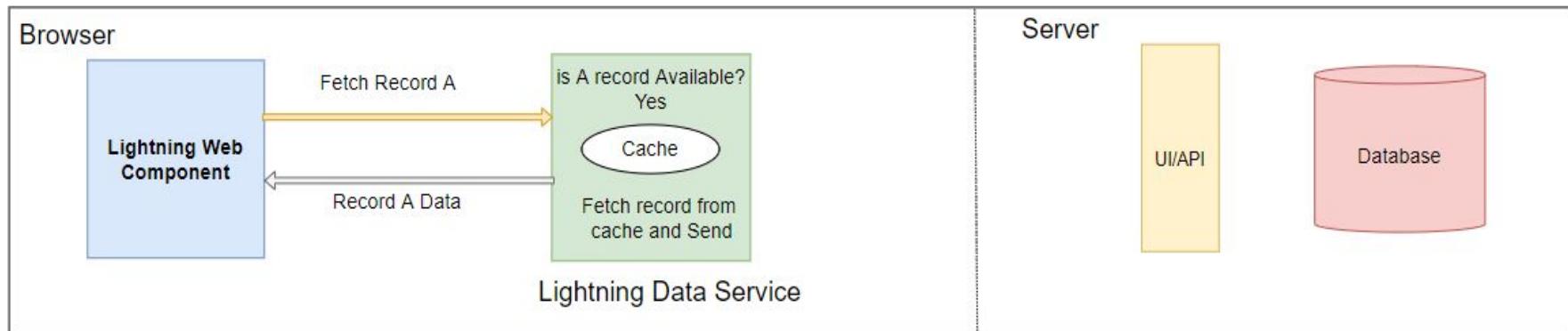
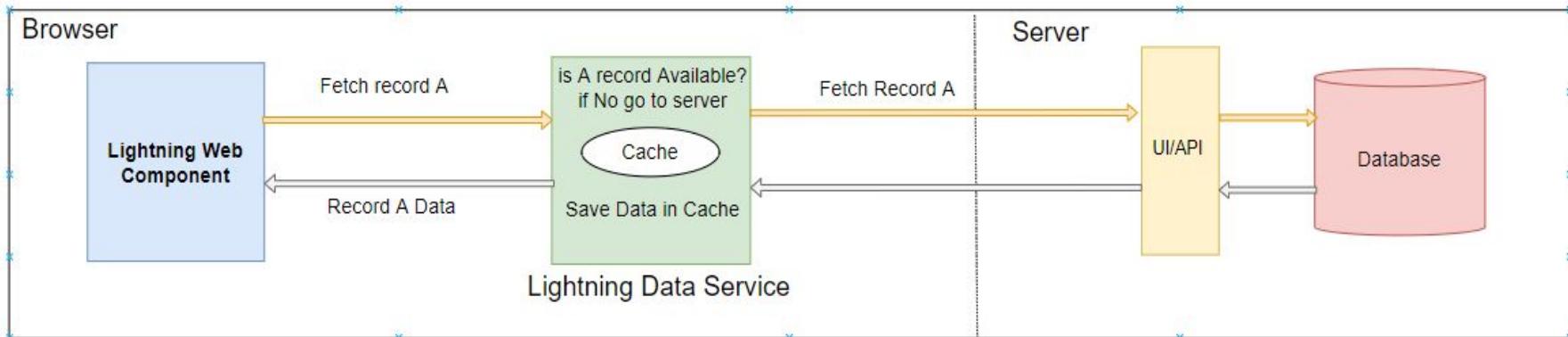


# Lightning Data Service



# Lightning Data Service

Lightning Data Service is a centralized data caching framework and it is built on top of User Interface API



# Lightning Data Service

1. Lightning Data Service is a centralized data caching framework and it is built on top of User Interface API
2. UI API gives you data and metadata in a single response and also respect CRUD access, field-level security settings, and sharing settings.
3. LDS displays only records and fields for which users have CRUD access and FLS visibility
4. LDS caches results on the Client
5. LDS, Invalidates cache entry when salesforce data and metadata changes
6. Optimizes server calls

# Base Lightning Components



# Base Lightning Components

Base Lightning Components are built on Lightning Data Service. So, Lightning Data Service is used behind the scenes by base components and inherits its caching and synchronisation capabilities

There are three types of base lightning components built on LDS are

- 1) lightning-record-form
- 2) lightning-record-edit-form
- 3) lightning-record-view-form

## When to Use these form?

- Create a metadata-driven UI or form-based UI similar to the record detail page in Salesforce.
- Display record values based on the field metadata.
- Hide or show localized field labels.
- Display the help text on a custom field.
- Perform client-side validation and enforce validation rules.

# Base Lightning Components

- **lightning-record-edit-form**—Displays an editable form.
- **lightning-record-view-form**—Displays a form in view mode.
- **lightning-record-form**—Supports edit, view, and read-only modes.

FEATURE	LIGHTNING - RECORD - FORM	LIGHTNING - RECORD - VIEW - FORM	LIGHTNING - RECORD - EDIT - FORM
Create Records	✓		✓
Edit Records	✓		✓
View Records	✓	✓	
Read-Only Mode	✓	✓	
Layout Types	✓		
Multi Column Layout	✓		
Custom Layout for Fields		✓	✓
Custom Rendering of Record Data		✓	✓

With **lightning-record-form**, you can specify a layout and allow admins to configure form fields declaratively. You can also specify an ordered list of fields to programmatically define what's displayed. **lightning-record-form** allows you to view and edit records.

# lightning-record-form



# lightning-record-form

Use the lightning-record-form component to quickly create forms to add, view, or update a record.

The lightning-record-form component provides these helpful features:

- Switches between view and edit modes automatically when the user begins editing a field in a view form
- Provides Cancel and Save buttons automatically in edit forms
- Uses the object's default record layout with support for multiple columns
- Loads all fields in the object's compact or full layout, or only the fields you specify

lightning-record-form is less customizable. To customize the form layout or provide custom rendering of record data, use lightning-record-edit-form (add or update a record) and lightning-record-view-form (view a record).

*Note\*\* Whenever possible, to boost performance, define fields instead of a layout. Specify a layout only when the administrator, not the component, needs to manage the provisioned fields.*

# Lightning-record-form key attributes

**object-api-name** - This attribute is always required. The *lightning-record-form* component requires you to specify the object-api-name attribute to establish the relationship between a record and an object.

**Note- Event and Task objects are not supported.**

**record-id** - This attribute is required only when you're editing or viewing a record.

**fields** - pass record fields as an array of strings. The fields display in the order you list them.

**layout-type** - Use this attribute to specify a Full or Compact layout. Layouts are typically defined (created and modified) by administrators. .

**modes** - This form support three mode

- **edit** - Creates an editable form to add a record or update an existing one. . Edit mode is the default when record-id is not provided, and displays a form to create new records.
- **view** - Creates a form to display a record that the user can also edit. The record fields each have an edit button. View mode is the default when record-id is provided.
- **readonly** - Creates a form to display a record that the user can also edit

**columns** - Use this attribute to show multiple columns in the form

# Create a Record Using lightning-record-form

Import references to Salesforce objects and fields from `@salesforce/schema`.

```
// Syntax
import objectName from '@salesforce/schema/objectReference';

import fieldName from '@salesforce/schema/object.fieldReference';
```

## Syntax

```
<lightning-record-form
    object-api-name={objectName}
    fields={fieldList}
    onsuccess={successHandler}>
</lightning-record-form>
```

# Display a record using lightning-record-form

We can display a record in two modes

- 1) **view** - In this mode, form using output fields with inline editing enabled.
- 2) **read-only** - In this mode, form loads with output fields only and you will not see edit icons or submit and cancel buttons.

## Syntax

```
<lightning-record-form  
    record-id="0010000002niCSzAAM"  
    object-api-name={objectName}  
    fields={fieldList}></lightning-record-form>
```

# Edit a record using lightning-record-form

Component using view mode by default. We can allow users to update fields directly in edit mode

## Syntax

```
<lightning-record-form  
    record-id="Your Record Id"  
    object-api-name="Your Object API Name"  
    columns="2"  
    mode="edit"  
    layout-type="Compact">  
</lightning-record-form>
```

# lightning-record-view-form



# lightning-record-view-form

- Use the lightning-record-view-form component to create a form that displays Salesforce record data for specified fields associated with that record. The fields are rendered with their labels and current values as read-only.
- You can customize the form layout or provide custom rendering of record data. If you don't require customizations, use lightning-record-form instead.
- To specify read-only fields, use lightning-output-field components inside lightning-record-view-form.

FEATURE	LIGHTNING-RECORD-FORM	LIGHTNING-RECORD-VIEW-FORM	LIGHTNING-RECORD-EDIT-FORM
Create Records	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Edit Records	<input checked="" type="checkbox"/>		<input checked="" type="checkbox"/>
View Records	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	
Read-Only Mode	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	
Layout Types	<input checked="" type="checkbox"/>		
Multi Column Layout	<input checked="" type="checkbox"/>		
Custom Layout for Fields		<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Custom Rendering of Record Data		<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>

# lightning-record-edit-form



# lightning-record-edit-form

- This component is used to create and edit the records.
- It provides custom layout of fields and custom rendering of record data

FEATURE	LIGHTNING - RECORD - FORM	LIGHTNING - RECORD - VIEW - FORM	LIGHTNING - RECORD - EDIT - FORM
Create Records	✓		✓
Edit Records	✓		✓
View Records	✓	✓	
Read-Only Mode	✓	✓	
Layout Types	✓		
Multi Column Layout	✓		
Custom Layout for Fields		✓	✓
Custom Rendering of Record Data		✓	✓

# Reset the lightning-record-edit-form



# Edit the lightning-record-edit-form



# Adding Custom Label to the Fields in Lightning-record-edit-form



# Custom Validation in lightning-record-edit-form



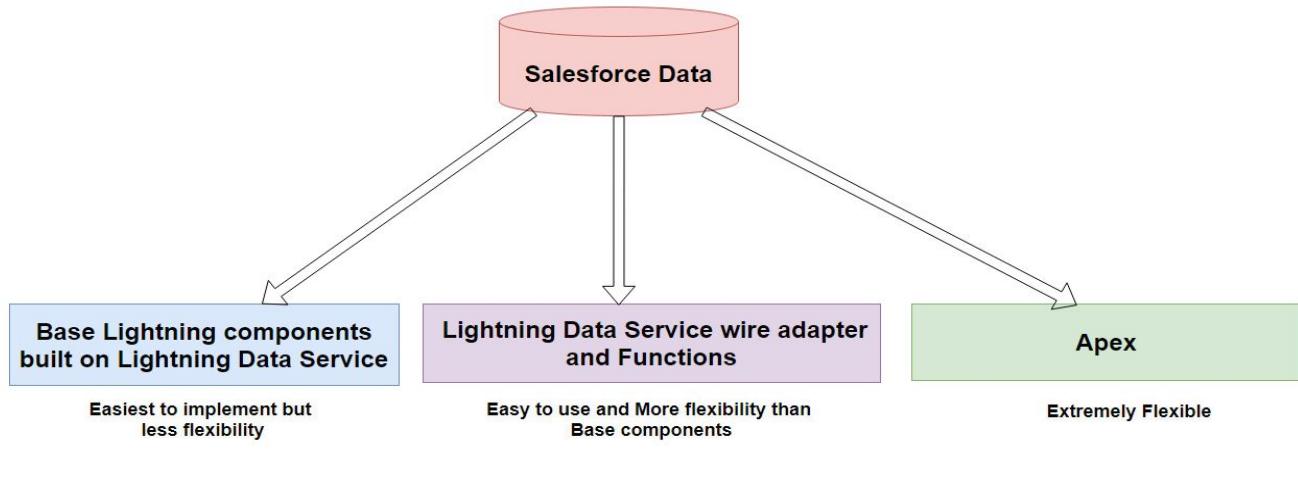
# Custom Validation approach

***lightning-input-field*** does not support client-side custom validation. If you want to implement your own client-side validation, consider using ***lightning-input*** nested in ***lightning-record-edit-form*** instead.

# Lightning Data Service Wire Adapters and Functions



# Lightning Data Service Wire Adapters and Functions



1. Use this to get the raw record data like picklist, recordInfo, objectInfo etc.
2. Want more customization to create a forms
3. You want to perform business logic

The wire adapters and JavaScript functions are available in [lightning/ui\\*Api](#) modules, which are built on User Interface API. User Interface API supports all custom objects and many standard objects

## lightning/ui\*Api Modules

→ **lightning/uiObjectInfoApi**

Use to get object metadata, and get picklist values.

→ **lightning/uiListApi(Beta)**

Get the records and metadata for a list view.

→ **lightning/uiRecordApi**

Use to record data and get default values to create records. It also includes JavaScript APIs to create, delete, update, and refresh records.

→ **lightning/uiAppApi(Beta)**

Use to get data and metadata for apps displayed in the Salesforce UI.

# @wire service and fetch user details

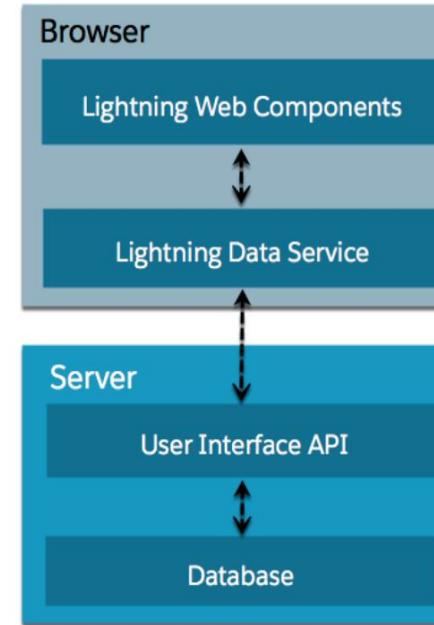


# What is @Wire Service?

1. Wire service is built on Lightning Data Service
2. LWC component use @wire in their JavaScript class to read data from one of the wire adapters in the lightning/ui\*Api namespace.
3. @wire is a reactive service
4. The wire adapter defines the data shape that the wire service provisions in an immutable stream

## Syntax

```
import { adapterId } from 'adapterModule';
@wire(adapterId, adapterConfig)
propertyOrFunction;
```



## *How to Import Reference of Salesforce Standard Object*

### *Syntax*

```
import objectName from '@salesforce/schema/object';
```

### *Example*

```
import ACCOUNT_OBJECT from '@salesforce/schema/Account';
```

## *How to Import Reference to Salesforce custom Object*

### *Syntax*

```
import objectName from '@salesforce/schema/object';
```

### *Example*

```
import PROPERTY_OBJECT from '@salesforce/schema/Property__c';
```

## *How to Import References to Salesforce Fields*

### Syntax

```
import FIELD_NAME from '@salesforce/schema/object.field';
```

### Example

```
import ACCOUNT_NAME from '@salesforce/schema/Account.Name';  
import PROPERTY_NAME from '@salesforce/schema/Property__c.Name';
```

## *How to Import Reference to a field via a relationship*

### Syntax

```
import REF_FIELD_NAME from '@salesforce/schema/object.relationship.field';
```

### Example

```
import ACCOUNT_OWNER from '@salesforce/schema/Account.Owner.Name';
```

# How @wire is reactive



# getObjectInfo

`lightning/uiObjectInfoApi`



# getObjectInfo

Use this wire adapter to get metadata about a specific object. The response includes metadata describing the object's fields, child relationships, record type, and theme.

## Syntax

```
import { LightningElement, wire } from 'lwc';
import { getObjectInfo } from 'lightning/uiObjectInfoApi';
import ACCOUNT_OBJECT from '@salesforce/schema/Account';

export default class Example extends LightningElement {
    @wire(getObjectInfo, { objectApiName: ACCOUNT_OBJECT })
    propertyOrFunction;
}
```

1. First import wire
2. Import adapter
3. Import dependencies of adapter i.e. fields or object reference
4. Call adapter using @wire
5. Pass adapterconfig

# getObjectInfos

`lightning/uiObjectInfoApi`



# getObjectInfos

Use this wire adapter to get metadata for multiple objects. The response includes metadata describing the fields, child relationships, record type, and theme for each object.

## Syntax

```
import { LightningElement, wire } from 'lwc';
import { getObjectInfos } from 'lightning/uiObjectInfoApi';
import ACCOUNT_OBJECT from '@salesforce/schema/Account';
import OPPORTUNITY_OBJECT from '@salesforce/schema/Opportunity';

export default class GetObjectInfosExample extends LightningElement {
    @wire(getObjectInfos, { objectApiNames: [ ACCOUNT_OBJECT, OPPORTUNITY_OBJECT ] })
    propertyOrFunction;
}
```

# getPicklistValues

`lightning/uiObjectInfoApi`



# getPicklistValues

Use this wire adapter to get the picklist values for a specified field.

## Syntax

```
import { LightningElement, wire } from 'lwc';
import { getPicklistValues } from 'lightning/uiObjectInfoApi';
import INDUSTRY_FIELD from '@salesforce/schema/Account.Industry';

export default class Example extends LightningElement {
    @wire(getPicklistValues, { recordTypeId: '012000000000000AAA', fieldApiName: INDUSTRY_FIELD })
    propertyOrFunction;
}
```

**recordTypeId** - The ID of the record type. Use the Object Info *defaultRecordTypeId* property, which is returned from [getObjectInfo](#)

**fieldApiName** - The API name of the picklist field

# getPicklistValuesByRecordType

[lightning/uiObjectInfoApi](#)



# getPicklistValuesByRecordType

Use this wire adapter to get the values for every picklist of a specified record type

## Syntax

```
import { LightningElement, wire } from 'lwc';
import { getPicklistValuesByRecordType } from 'lightning/uiObjectInfoApi';
import ACCOUNT_OBJECT from '@salesforce/schema/Account';

export default class Example extends LightningElement {
    @wire(getPicklistValuesByRecordType, { objectApiName: ACCOUNT_OBJECT, recordTypeId: '012000000000000AAA' })
    propertyOrFunction
}
```

**recordTypeId** - The ID of the record type. Use the Object Info *defaultRecordTypeId* property, which is returned from *getObjectInfo*

**objectApiName** - The API name of the Object

# getRecord

## lightning/uiRecordApi



# getRecord

Use this wire adapter to get the record's data

## Syntax

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

@wire(getRecord, { recordId: string, fields: string|string[], optionalFields?: string|string[]})
propertyOrFunction

@wire(getRecord, { recordId: string, layoutTypes: string|string[],
    modes?: string|string[], optionalFields?: string|string[]})
propertyOrFunction
```

**recordId**- The ID of the record type.

**fields**- A field or an array of fields to return  
or

**layoutType** - it support two values Compact or Full(default)

**Modes** - used with layout. Values supported are Create, Edit and View(default)

**optionalFields** - a field name or an array of field names. If a field is accessible to the user, it includes in response otherwise it will not throw an error.

# `getFieldValue`   & `getFieldDisplayValue`

`lightning/uiRecordApi`



# getFieldValue

Use this to gets a field's value from a record

## Syntax

```
import { getFieldValue } from 'lightning/uiRecordApi';
getFieldValue(record: Record, field: string)
```

# getFieldDisplayValue

Use this to gets a field's value in formatted and localized format from a record

## Syntax

```
import { getFieldDisplayValue } from 'lightning/uiRecordApi';
getFieldDisplayValue(record, field)
```

# getRecordUi

lightning/uiRecordApi



# getRecordUi

Use this wire adapter to get layout information, metadata, and data to build UI for one or more records.

## Syntax

```
import { LightningElement, wire } from 'lwc';
import { getRecordUi } from 'lightning/uiRecordApi';

export default class Example extends LightningElement {
    @wire(getRecordUi, { recordIds: string|string[], layoutTypes: string|string[],
                        modes: string|string[], optionalFields?: string|string[] })
    propertyOrFunction;
}
```

**recordIds**- IDs of records to load.

**layoutType** - it support two values Compact or Full

**Modes** - used with layout. Values supported are Create, Edit and View

**optionalFields** - a field name or an array of field names. If a field is accessible to the user, it includes in response otherwise it will not throw an error.

# getRecord

Use this wire adapter to get the record's data

## Syntax

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

@wire(getRecord, { recordId: string, fields: string|string[], optionalFields?: string|string[]})
propertyOrFunction

@wire(getRecord, { recordId: string, layoutTypes: string|string[],
    modes?: string|string[], optionalFields?: string|string[]})
propertyOrFunction
```

**recordId**- The ID of the record type.

**fields**- A field or an array of fields to return  
or

**layoutType** - it support two values Compact or Full(default)

**Modes** - used with layout. Values supported are Create, Edit and View(default)

**optionalFields** - a field name or an array of field names. If a field is accessible to the user, it includes in response otherwise it will not throw an error.

# createRecord

## lightning/uiRecordApi



# createRecord(recordInput)

CreateRecord is use to creates a record

## Syntax

```
import { createRecord } from 'lightning/uiRecordApi';
createRecord(recordInput: Record): Promise<Record>
```

**recordInput**-A Record Input object is used to create the record. This object take object apiName and fields details as input

# updateRecord

## lightning/uiRecordApi



# updateRecord(recordInput, clientOptions)

Use this function to updates a record. Provide the recordId of the record to update in recordInput.

## Syntax

```
import { updateRecord } from 'lightning/uiRecordApi';
updateRecord(recordInput: Record, clientOptions?: Object): Promise<Record>
```

**recordInput**-A Record Input object is used to update the object.

**clientOptions- (optional)** - To check for conflicts before you update a record

# deleteRecord

## lightning/uiRecordApi



# deleteRecord(recordId)

Use this function to delete a record

## Syntax

```
import { deleteRecord } from 'lightning/uiRecordApi';
deleteRecord(recordId: string): Promise<void>
```

**recordId**—(Required) The ID of the record to delete.

# getNavItems(beta)

lightning/uiAppsApi



# getNavItems(Beta)

Use this wire adapter to retrieve the items in the navigation menu

*lightning/uiAppsApi is for evaluation purposes only, not for production use.*

## Syntax

```
import { LightningElement, wire } from 'lwc';
import { getNavItems } from 'lightning/uiAppsApi';

export default class Example extends LightningElement {
    @wire(getNavItems, {
        navItemNames: ['standard-Account'],
        pageSize: 30,
    })
    propertyOrFunction;
```

# getListUi(beta)

## lightning/uiListApi



# getListUi(Beta)

Use this wire adapter to get the records and metadata for a list view

*lightning/uiListApi is for evaluation purposes only, not for production use.*

## Syntax

```
import { LightningElement, wire } from 'lwc';
import { getListUi } from 'lightning/uiListApi';
import CONTACT_OBJECT from '@salesforce/schema/Contact';

export default class Example extends LightningElement {
    @wire(getListUi, { objectApiName: CONTACT_OBJECT, listViewApiName: 'AllContacts' })
    propertyOrFunction;
}
```

**objectApiName**- The API name of a supported object

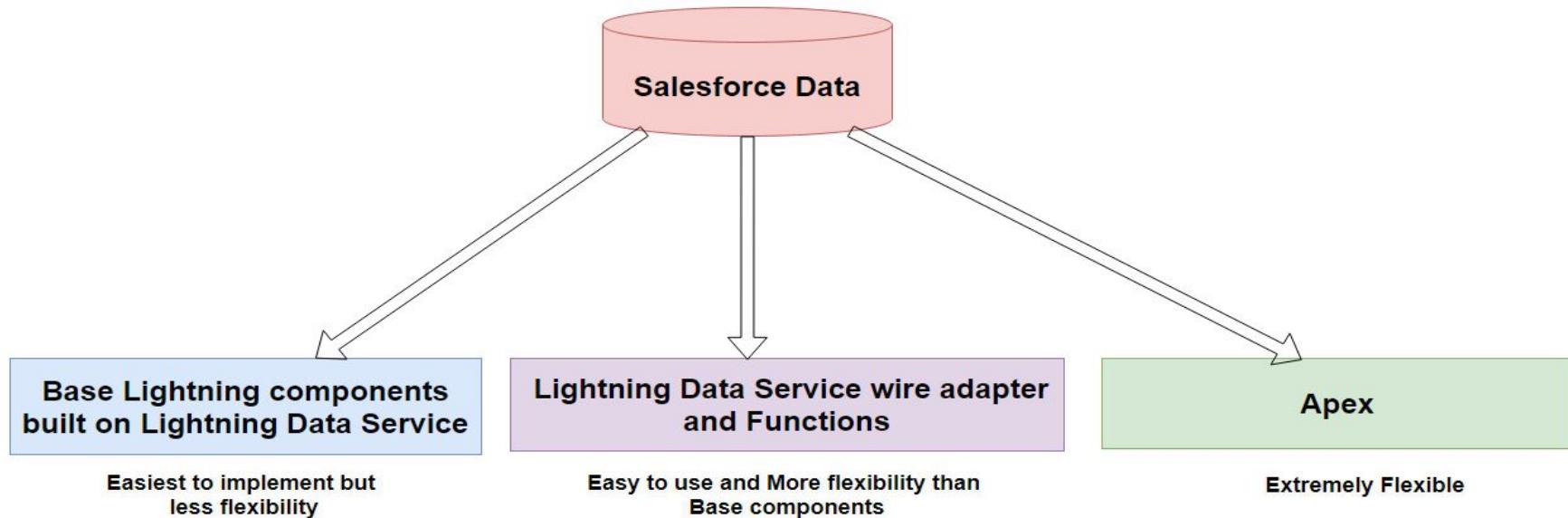
**listViewApiName** - The API name of a list view such as AllAccounts, AllContacts etc.

**sortBy, pageSize, pageToken**

# Apex in LWC



# Apex in LWC

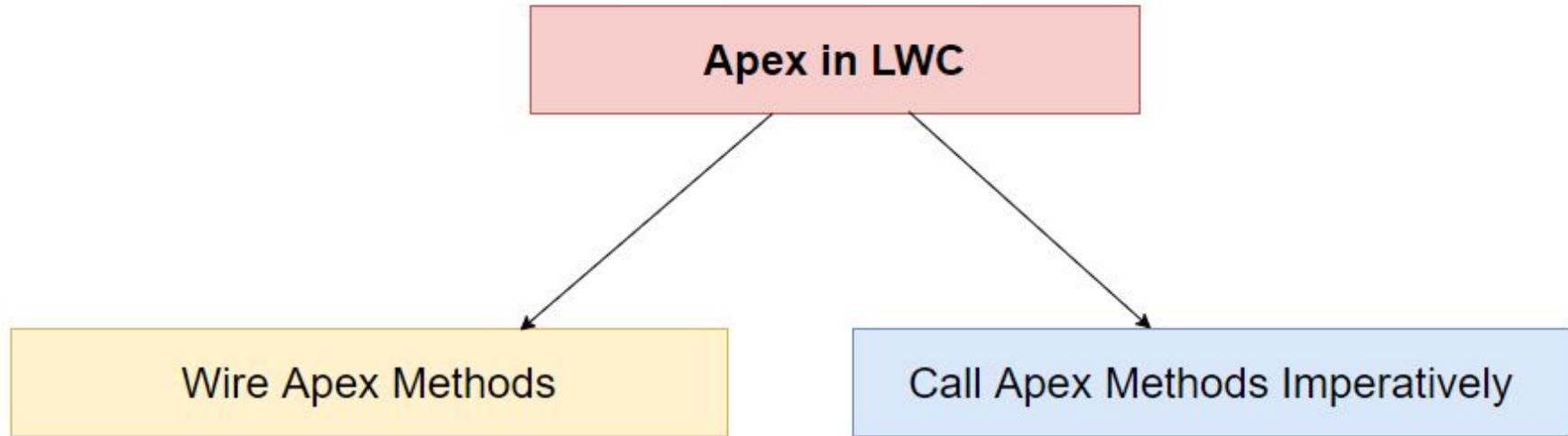


If you can't use a base component, and you can't use the Lightning Data Service wire adapters or functions, use Apex.

# When to Use Apex approach

- To work with objects that aren't [supported by User Interface API](#), like Task and Event.
- To work with operations that User Interface API doesn't support, like loading a list of records by criteria (for example, to load the first 200 Accounts with Amount > \$1M).
- To perform a transactional operation. For example, to create an Account and create an Opportunity associated with the new Account. If either create fails, the entire transaction is rolled back.

# Two Ways to Call Apex Methods in LWC



# Expose Apex Methods to LWC



# Expose Apex Methods to LWC

1. Apex Method must be **static** and either **global** or **public**
2. Method should be annotated with **@AuraEnabled**

## Syntax

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

# Import Apex Methods



# Import Apex Methods

Use default import syntax in JavaScript to import an Apex method via the `@salesforce/apex` scoped packages

## Syntax

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

- **apexMethodName**—A symbol that identifies the Apex method.
- **apexMethodReference**—The name of the Apex method to import.
- **Classname**—The name of the Apex class.
- **Namespace**—If the class is in the same namespace as the component, don't specify a namespace. If the class is in a managed package, specify the namespace of the managed package.

# Wire Apex Method



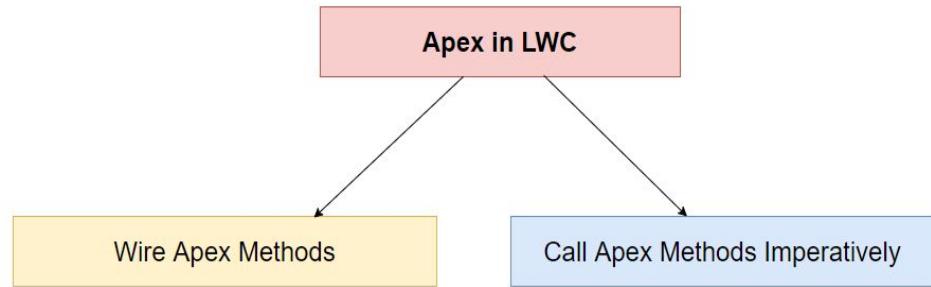
# Wire Apex Method with Parameters



# wire Apex Methods

We can wire an apex methods in two ways

- 1) Wire an Apex Method to a Property
- 2) Wire an Apex Method to a Function



## Syntax

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

- **apexMethodName**—A symbol that identifies the Apex method.
- **apexMethodParams** -- An object with properties that match the parameters of the apexMethod

**Note\*\*\* If a parameter value is null, the method is called but if the value is undefined, the method isn't called.**

# Call Apex Methods Imperatively



# Call Apex Methods Imperatively

Use this approach over @wire in the following situations

- 1) To call a method that isn't annotated with `cacheable=true`, which includes any method that inserts, updates, or deletes data.
- 2) To control when the invocation occurs.
- 3) To work with objects that aren't [supported by User Interface API](#), like Task and Event.
- 4) To call a method from an ES6 module that doesn't extend [LightningElement](#)

## Syntax

```
fetchData() {
    getAccountList()
        .then((result) => {
            this.accounts = result;
            this.error = undefined;
        })
        .catch((error) => {
            this.error = error;
            this.accounts = undefined;
        });
}
```

## Apex in LWC

Wire Apex Methods

Call Apex Methods Imperatively

# Apex Imperative Method with Parameters



# Apex in LWC



# refreshApex function



# refreshApex()

To refresh Apex data provisioned via `@wire`, call `refreshApex()`. The function provisions the data using the configuration bound to the `@wire` and updates the cache.

## Syntax

```
import { refreshApex } from '@salesforce/apex';
refreshApex(valueProvisionedByWireService)
```

**valueProvisionedByWireService**—A property annotated with `@wire`, or if you annotated a function, the argument that the wired function receives.

# refreshApex function



# getRecordNotifyChange()

Call this function to notify Lightning Data Service that a record has changed outside its mechanisms, such as via imperative Apex or Visualforce, or by calling User Interface API via a third-party framework..

## Syntax

```
import { getRecordNotifyChange } from 'lightning/uiRecordApi';

getRecordNotifyChange(items: Array<{recordId: string}>)
```

**items**—(Required) An array of objects with an object shape of `{ recordId: string }`

# Error Handling in



# Books Listing App With Rest API Callout



# PDF Generation In LWC



# Maps In LWC



# Generate CSV in LWC



# Reusable Charts in LWC



# Filtering in LWC



# Sorting in LWC



# Reusable Modal Component



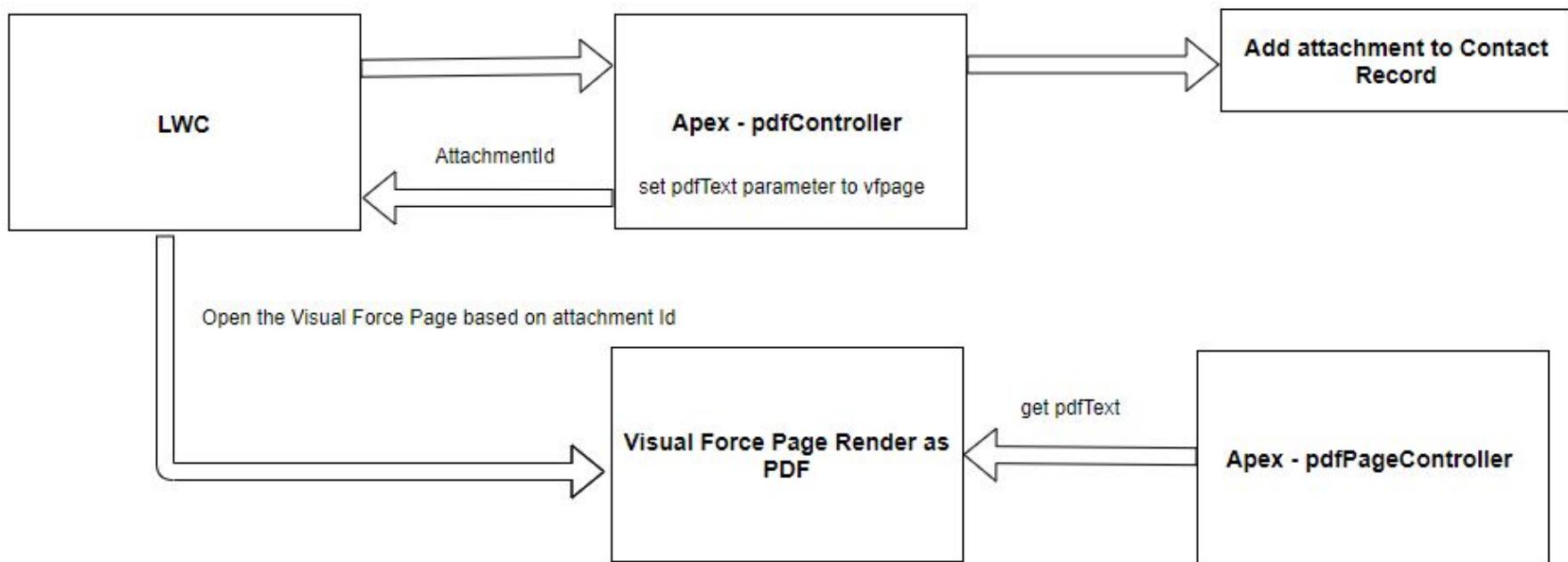
# Start and End Date Validation in LWC



# Sorting in LWC



# PDF Generation Approach



# Retrieve Data From Old Scratch org and Push to New Scratch Org



# Steps to create Message Channel

-

# getRecord

Use this wire adapter to get a record's data

## Syntax

```
import { LightningElement, wire } from 'lwc';
import { getObjectInfo } from 'lightning/uiObjectInfoApi';
import ACCOUNT_OBJECT from '@salesforce/schema/Account';

export default class Example extends LightningElement {
    @wire(getObjectInfo, { objectApiName: ACCOUNT_OBJECT })
    propertyOrFunction;
}
```

1. First import wire
2. Import adapter
3. Import dependencies of adapter i.e. fields or object reference
4. Call adapter using @wire

# Error handling in @wire



# Error response types

The body of the error response depends on the API that returns it.

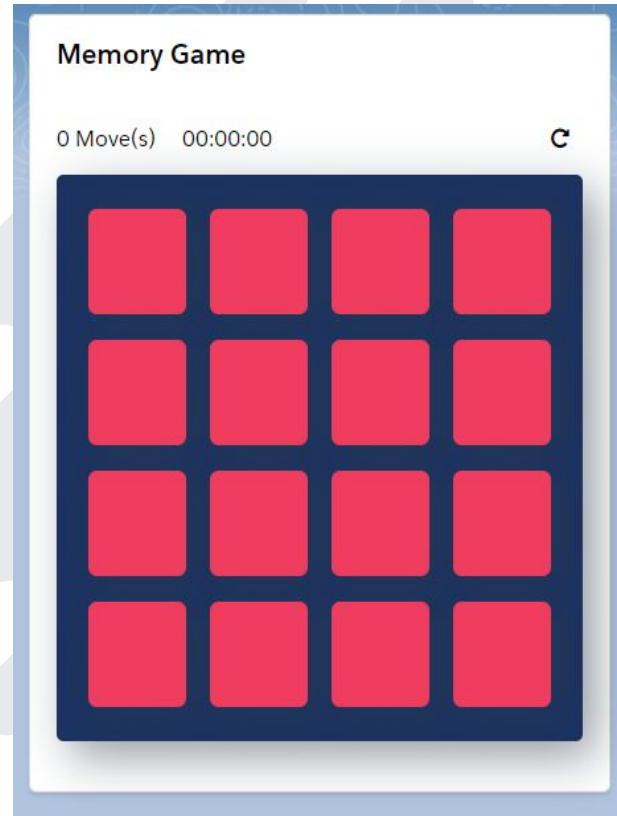
- UI API read operations, such as the getRecord wire adapter, return error.body as an array of objects.
- UI API write operations, such as the createRecord wire adapter, return error.body as an object, often with object-level and field-level errors.
- Apex read and write operations return error.body as an object.
- Network errors, such as an offline error, return error.body as an object.

# Memory Game in LWC



# Memory Game in LWC

- Lightning Page Creation
- LWC Component Creation
- CSS Grid creation
- Adding and removing classes
- querySelector
- Loading static resources
- renderCallback
- Looping in HTML
- Timing Functions(setInterval)
- Timer start and stop
- SLDS Modal
- getter



# HTML to PDF in LWC



# Car Hub Project



# Project Setup



# Car Hub App Creation



# Car Object Creation



# Adding Custom Fields to Car Object



# Adding Static resources and Creating Car records



# Custom Page Template Design



# Component Creation and Placement



# Filter Section Designing



# Fetching Cars from Apex to carListItem



# Creating Car Tiles



# Creating Car Tiles



# getCars Method with filter logic



# LMS implementation And Filtering



# LMS implementation And Filtering



# Passing recordId from carTile to carCard



# Placeholder Component



# Navigate to record Page from car Card Component



# Similar Car Component



# Summary



# CAR HUB Project Learnings

- Project Setup and org setup
- App Creation and setting up the branding
- Object and fields Creation
- Lightning Page Creation
- Static Resources and Data loading
- Custom Page Layout Designing
- Different form elements and filtering
- Lightning Message Service
- Wire adapter and functions calling



FORD ECOSPORT  
MSRP:USD 100,000.00



FORD ENDEAVOUR  
MSRP:USD 900,000.00



HONDA JAZZ  
MSRP:USD 567,567.00



HONDA WRV  
MSRP:USD 238,909.00



HYUNDAI VENUE  
MSRP:USD 100,300.00



HYUNDAI HYUNDAI20  
MSRP:USD 683,899.00

# CAR HUB Project Learnings

- Imperative Apex calling
  - Parent to Child and Child to parent data passing
  - Lifecycle hooks and Navigation
  - getRecord, getObjectInfo and getPicklistValues
- Adapter
- LDS ( lightning-record-view-form)
  - Creating placeholder component
  - Styling, looping, properties handling



FORD ECOSPORT  
MSRP:USD 100,000.00



FORD ENDEAVOUR  
MSRP:USD 900,000.00



HONDA JAZZ  
MSRP:USD 567,567.00



HONDA WRV  
MSRP:USD 238,909.00



HYUNDAI VENUE  
MSRP:USD 100,300.00



HYUNDAI HYUNDAI20  
MSRP:USD 683,899.00

# Job Website using LWC OSS



# Project Setup



# LWC OSS

LWC OSS is an open source Javascript framework. Now you can build apps anywhere with your choice of tools and library.

You just need the knowledge of lightning web components and you are good to start with LWC OSS.

## Setup

```
$ npx create-lwc-app my-app  
$ cd my-app  
$ npm run watch
```

# Navbar component creation



# Bootstrap Library Setup



# Search Box Component



# Calling Job API



# Job Card Component



# Utility Creation



# Job Description Component



# View Detail Button



# Apply Now Button



# Back to Jobs From Description



# Real-time job search



# Heroku Deployment



# Custom Carousel Using LWC



Presented By - Salesforce Troop





```
*****  
*****Deploy your Lightning Web Component OSS to Heroku in 5 steps*****  
*****
```

```
*****Step 1 - Create LWC OSS APP*****  
npx create-lwc-app my-app  
cd my-app  
npm run watch
```

```
*****Step 2 - Heroku setup and heroku app creation*****  
//Sign up - https://signup.heroku.com/  
//CLI setup - https://devcenter.heroku.com/articles/heroku-cli  
heroku -v  
heroku login  
heroku create
```

```
*****Step 3 - Create Procfile file*****  
web: npm run serve
```

```
*****Step 4 - Initialise Git and add a Heroku remote to existing git *****  
git init  
git add .  
git commit -m "Initial Commit"  
git remote add heroku "heroku git url" //heroku is the alias
```

```
*****Step 5 - 5. Deployment and test *****  
git push heroku main  
heroku open
```

# HTML to PDF in LWC

- Lightning Page Creation
- LWC Component Creation
- Invoice creation
- Inline styling
- Visualforce Page
- Pdf controller in apex
- Display controller for vfpage
- Imperative Apex call
- Fetching recordId
- Open PDF in new window

1 / 1

SPARKSUITE

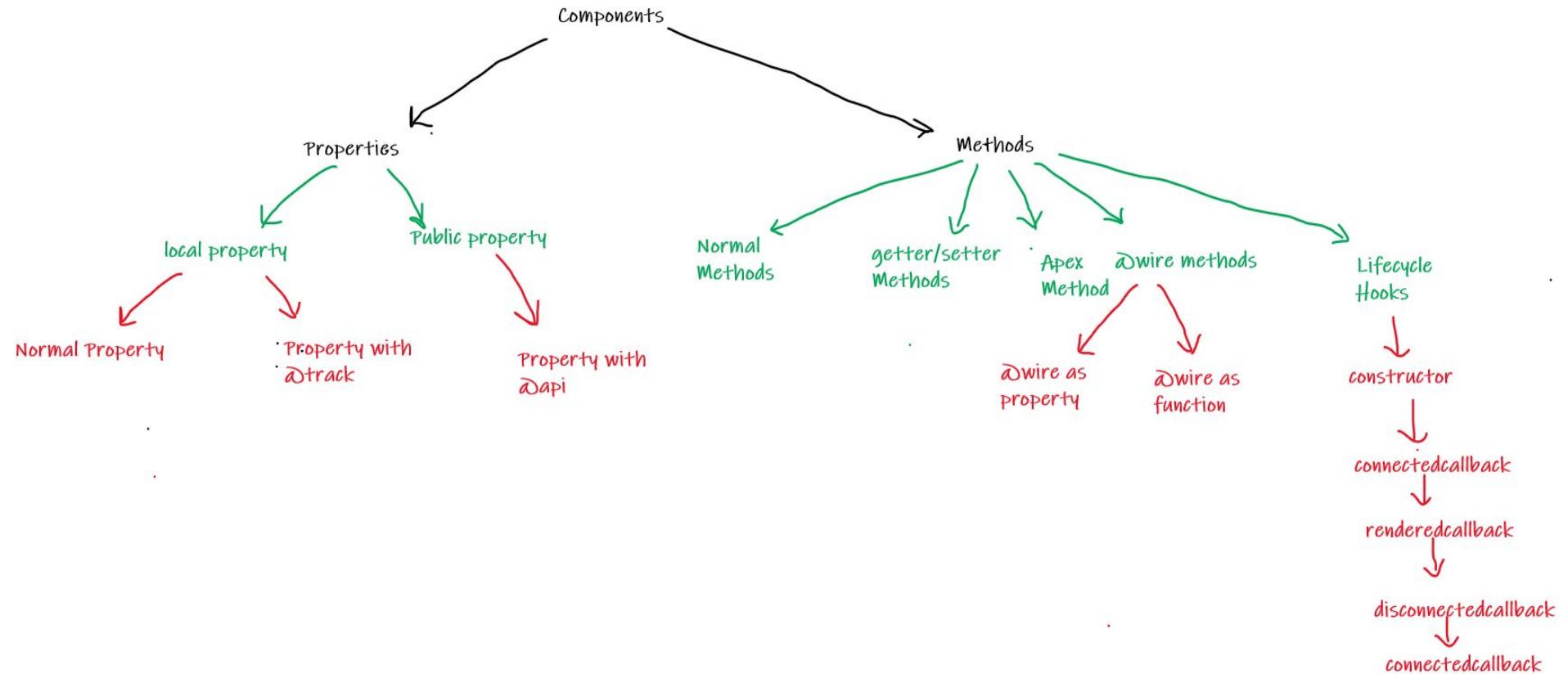
Invoice #: 123  
Created: January 1, 2015  
Due: February 1, 2015

Sparksuite, Inc.  
12345 Sunny Road  
Sunnyville, CA  
12345

Acme Corp.  
John Doe  
john@example.com

Payment Method	Check #
Check	1000
Website design	\$300.00
Hosting (3 months)	\$75.00
<b>Total: \$385.00</b>	

# High Level Component Understanding



# Component Naming Convention

The folder and its files must follow these naming rules.

- Must begin with a lowercase letter
- 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)