**Salesforce VIVA Questions**

1. **Hands-on Scenario in Assessment**: This is a practical exercise or real-world scenario where you apply Salesforce concepts (e.g., configuring flows, reports, or automation) to solve problems or implement solutions.
2. **Difference between Lookup and Master-Detail**:
   * **Lookup**: A relationship where one object references another. It’s optional, and the child can exist independently.
   * **Master-Detail**: A tightly bound relationship where the child record cannot exist without the master. Deleting the master deletes the child.
3. **Roll-Up Summary Field**: A field on a master record that calculates aggregate values (e.g., COUNT, SUM) of related detail records in a master-detail relationship.
4. **Screen Flow vs Record-Triggered Flow**:
   * **Screen Flow**: User-interactive flow for capturing input and performing operations.
   * **Record-Triggered Flow**: Automatically triggered when a record is created, updated, or deleted (no user interaction).
5. **Junction Object**: A custom object used to create many-to-many relationships between two objects, typically via two master-detail relationships.
6. **Formula Field**: A read-only field that calculates a value dynamically based on other field values or expressions.
7. **Data Security**:
   * **Object Level Security**: Control access to objects (e.g., via Profiles). Implement using profiles or permission sets.
   * **Field Level Security**: Control access to fields on objects. Implement through profiles or permission sets.
   * **Record Level Security**: Control access to individual records (e.g., using Sharing Rules, Role Hierarchies, or Manual Sharing).
8. **Data Loader vs Data Import Wizard**:
   * **Data Loader**: For bulk data operations (insert, update, delete) via CSV files. More control over data operations.
   * **Data Import Wizard**: A user-friendly tool for importing data into Salesforce, suitable for non-technical users.
9. **Difference between Profile and Permission Set**:
   * **Profile**: Defines user permissions for objects, fields, and other resources (cannot be shared).
   * **Permission Set**: Additional permissions granted to users (can be shared).
10. **Reports and Dashboards**:
    * **Reports**: Summarized data views in various formats.
    * **Dashboards**: Visual representation of reports.
    * **Report Formats**: Tabular, Summary, Matrix, and Joined.
11. **Governor Limits**: Limits set by Salesforce to ensure fair resource usage, such as limits on API calls, records processed, etc.
12. **Salesforce Editions**: Different packages with varying feature sets, including Essentials, Professional, Enterprise, and Developer Editions.
13. **Custom Report Type**: A user-defined report type that allows for custom relationships between objects in reports.
14. **Governor Limit**: Same as above. Salesforce enforces limits to prevent resource overuse.
15. **With Sharing vs Without Sharing**:
    * **With Sharing**: Applies the organization-wide sharing rules to the class.
    * **Without Sharing**: Ignores sharing rules, allowing access to all records.
16. **SOQL vs SOSL**:
    * **SOQL**: Queries records from a single object.
    * **SOSL**: Searches across multiple objects.
17. **Apex Triggers**: Apex code executed in response to record changes. Best practices include bulkification, handling DML operations outside loops, and using context variables (like Trigger.new).
18. **LWC**: Lightning Web Components are modern, fast components built with standard web technologies (e.g., JavaScript, HTML).
19. **LWC Bundle**: A set of files that together define a Lightning Web Component, including HTML, CSS, JavaScript, and metadata files.
20. **@wire, @api, @track**:
    * **@wire**: Fetches data reactively from Salesforce or Apex.
    * **@api**: Exposes a public property or method for parent-child communication.
    * **@track**: Marks properties for reactivity in the component.
21. **Wire to a Property vs Wire to a Function**:
    * **Wire to a Property**: Automatically reacts to changes in data.
    * **Wire to a Function**: Provides more control and allows for custom logic.
22. **Call Apex Imperative Approach**: Invoking Apex methods from LWC using @wire or Imperative Apex (e.g., using this.myApexMethod()).
23. **Callback and Arrow Function**:
    * **Callback**: A function passed as an argument to another function.
    * **Arrow Function**: A concise syntax for functions that preserves the lexical scope of this.
24. **Parent-Child Communication in LWC**:
    * **Parent to Child**: Pass data via properties (@api).
    * **Child to Parent**: Use events (CustomEvent) to notify parent.
25. **Lightning Message Service (LMS)**: Enables communication between components that aren't in the same DOM tree, implemented using message channels.
26. **Lightning Data Service**: Simplifies data access and handling in LWC without needing Apex, using standard Salesforce objects and CRUD operations.
27. **Service Component**: A component that encapsulates logic, reusable across other components or processes in Salesforce.
28. **NavigationMixin in LWC**: Used for navigating between pages in Salesforce (e.g., opening records, URLs).
29. **Static vs Dynamic Schema in LWC**:
    * **Static Schema**: The object model is fixed and predefined.
    * **Dynamic Schema**: Allows querying of objects and fields dynamically at runtime.
30. **Invoke LWC from Flow**: Use Flow components to embed and invoke LWC in flows.
31. **Use of Lightning Tabset**: Organizes multiple tabs in a single container to allow users to switch between views.
32. **Asynchronous Apex**: Apex that runs asynchronously, using methods like @future, Queueable, or Batch Apex.
33. **Integration**: Connecting Salesforce to external systems.
    * **SOAP API**: Uses XML and is suitable for legacy systems.
    * **REST API**: Uses JSON and is lightweight for modern web services.
    * **REST API Methods**: GET, POST, PUT, DELETE.
34. **Apex Testing**: Unit tests for validating Apex code functionality. Best practices include writing tests for bulk operations and ensuring code coverage.
    * **Annotations**: @isTest, @testSetup.

**HON CODE EXPLANATION**

This is an Apex controller class written for Salesforce, designed to interact with a custom object Sports\_Program\_Object\_\_c. It provides two main functionalities: retrieving a list of sports programs and deleting a specific sports program. Here's a breakdown of each part:

**Class Declaration**

public with sharing class SportsAcademyController {

* **public**: This means the class is accessible by other classes or components (like Lightning Web Components or Visualforce pages).
* **with sharing**: This enforces sharing rules. It means the code respects the organization’s sharing settings for visibility on records. For example, a user may only access records they have permission to see.
* **SportsAcademyController**: The name of the class.

**Method 1: getSportsProgram()**

@AuraEnabled(cacheable=true)

public static List<Sports\_Program\_Object\_\_c> getSportsProgram() {

return [SELECT Id, Name FROM Sports\_Program\_Object\_\_c];

}

* **@AuraEnabled(cacheable=true)**:
  + This annotation makes the method accessible from Lightning Components (such as Lightning Web Components or Aura Components).
  + The cacheable=true flag means the result of this method can be cached on the client side for performance optimization, especially when the data does not change frequently.
* **public static List<Sports\_Program\_Object\_\_c> getSportsProgram()**:
  + This defines a **static method** (meaning it can be called without creating an instance of the class) that returns a list of Sports\_Program\_Object\_\_c records.
  + The method queries Salesforce for a list of Sports\_Program\_Object\_\_c records, specifically fetching the Id and Name fields from this custom object.
* **return [SELECT Id, Name FROM Sports\_Program\_Object\_\_c];**:
  + The query uses Salesforce Object Query Language (SOQL) to retrieve all records from the Sports\_Program\_Object\_\_c object and return their Id and Name fields.

**Method 2: deleteSportsProgram(Id recordId)**

@AuraEnabled

public static void deleteSportsProgram(Id recordId) {

delete [SELECT Id FROM Sports\_Program\_Object\_\_c where Id= :recordId];

}

* **@AuraEnabled**: This annotation also makes this method accessible from Lightning Components (similar to the first method). However, it doesn't include cacheable=true because this is a **mutation operation** (deleting records) and thus can't be cached.
* **public static void deleteSportsProgram(Id recordId)**:
  + This is another **static method**, but it doesn't return anything (void).
  + It takes one parameter: recordId (of type Id), which is the identifier of the record to be deleted.
* **delete [SELECT Id FROM Sports\_Program\_Object\_\_c where Id= :recordId];**:
  + This query deletes the record from the Sports\_Program\_Object\_\_c object where the Id matches the provided recordId.
  + delete is a DML (Data Manipulation Language) operation that removes the specified record from Salesforce.

**Key Points:**

1. **getSportsProgram()** is a @AuraEnabled method that retrieves and returns a list of sports programs (custom object Sports\_Program\_Object\_\_c) with their Id and Name.
2. **deleteSportsProgram(Id recordId)** is a @AuraEnabled method that deletes a specific sports program from Salesforce, based on the provided record ID.
3. Both methods are static and designed to be called from external components, such as a Lightning Web Component, allowing front-end interfaces to access or delete data without directly interacting with the database.

This is the markup for a **Lightning Web Component (LWC)** in Salesforce. It uses lightning-card and other Lightning components to display a list of sports programs and allow the user to delete a selected program. Here's a breakdown of the code:

**Code Breakdown**

**<lightning-card> Component**

<lightning-card title="Sports Program">

* **<lightning-card>**: This is a standard Lightning Web Component that creates a styled card with a title.
* **title="Sports Program"**: Sets the title of the card to "Sports Program".

**List of Programs**

<template if:true={programs}>

<ul>

<template for:each={programs} for:item="program">

<li key={program.Id}>

<lightning-button

label={program.Name}

data-id={program.Id}

onclick={handleSelect}>

</lightning-button>

</li>

</template>

</ul>

</template>

* **<template if:true={programs}>**: This checks if the programs property (which is presumably an array of sports programs) has data. If programs is not empty, it renders the contents inside.
* **<ul> and <li>**: The programs will be listed in an unordered list (<ul>), with each program represented by a list item (<li>).
* **<template for:each={programs} for:item="program">**: This is an iteration directive in LWC. It loops through each item in the programs array, and for each program, it creates a <li> element.
  + **program**: The variable representing the current item in the loop (each program object).
* **<lightning-button>**: For each program, a Lightning button is created with:
  + **label={program.Name}**: The button label will be the name of the sports program.
  + **data-id={program.Id}**: The data-id attribute is set to the ID of the program. This is often used to pass data in a custom attribute, which can be accessed later.
  + **onclick={handleSelect}**: This binds the handleSelect method to the button’s click event, allowing the user to select the program when the button is clicked.

**Delete Button for Selected Program**

<template if:true={selectedProgramId}>

<lightning-button

label="Delete"

variant="destructive"

onclick={handleDelete}>

</lightning-button>

</template>

* **<template if:true={selectedProgramId}>**: This checks if selectedProgramId has a value. If it does, it indicates that a program has been selected, so it will render the delete button.
* **<lightning-button>**:
  + **label="Delete"**: The label of the button is set to "Delete".
  + **variant="destructive"**: This makes the button appear in a red color (a "destructive" action, signaling that it will delete something).
  + **onclick={handleDelete}**: This binds the handleDelete method to the button’s click event, allowing the user to delete the selected program.

**Summary of Functionality**

* **List of Programs**: The component renders a list of sports programs, where each program is displayed as a button. The program's name is shown on the button, and when the button is clicked, the handleSelect method is triggered.
* **Delete Button**: If a program is selected (i.e., selectedProgramId has a value), a "Delete" button appears. When clicked, it triggers the handleDelete method to remove the selected program.

**Key Points to Remember:**

* The programs array is assumed to be an array of sports programs, each containing at least an Id and Name field.
* The handleSelect method is responsible for setting the selectedProgramId when a program is chosen.
* The handleDelete method will delete the selected program using the selectedProgramId.
* This code uses LWC directives like if:true for conditional rendering and for:each for looping through arrays.

**Conclusion**

This component is designed to display a list of sports programs as buttons, allowing users to select one and then delete it. The selectedProgramId is used to track the selected program, and the delete button appears only once a program is selected. The component relies on the handler functions (handleSelect and handleDelete) to manage the selection and deletion of programs.

This is the JavaScript part of a **Lightning Web Component (LWC)** in Salesforce. It manages the retrieval and deletion of sports programs by interacting with an Apex controller (SportsAcademyController). Here’s a detailed explanation of the code:

**Code Breakdown**

**1. Imports**

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

import getSportsProgram from '@salesforce/apex/SportsAcademyController.getSportsProgram';

import deleteSportsProgram from '@salesforce/apex/SportsAcademyController.deleteSportsProgram';

import { refreshApex } from '@salesforce/apex';

* **LightningElement**: The base class that all Lightning Web Components (LWCs) must extend.
* **track**: This decorator is used to make properties reactive. If a tracked property changes, the component automatically re-renders.
* **wire**: This decorator is used to wire Apex methods to a component, allowing automatic updates when the data changes or when the method is re-executed.
* **getSportsProgram**: This imports the Apex method getSportsProgram from the SportsAcademyController Apex class to retrieve sports program records.
* **deleteSportsProgram**: This imports the Apex method deleteSportsProgram from the Apex controller, allowing deletion of a specific sports program.
* **refreshApex**: This is a utility function to refresh the data of an Apex-wired property after a DML operation (like deletion) has been performed.

**2. Component Properties**

export default class SportsProgram extends LightningElement {

@track programs;

selectedProgramId;

* **@track programs**: This property holds the list of sports programs fetched from the Apex controller. It's marked with @track so the component will re-render when the list changes.
* **selectedProgramId**: This property stores the ID of the selected sports program. It is used to track which program the user has chosen for deletion.

**3. Wire Service: Fetching Data**

@wire(getSportsProgram)

wiredPrograms({error, data}) {

if (data) {

this.programs = data;

} else if (error) {

console.error(error);

}

}

* **@wire(getSportsProgram)**: This decorator binds the getSportsProgram Apex method to the wiredPrograms method. It makes the component automatically call the Apex method and pass the result (or error) into wiredPrograms.
  + **data**: Contains the fetched list of sports programs from the getSportsProgram Apex method. If the data is returned successfully, it is assigned to the programs property.
  + **error**: Contains any error returned by the Apex method. If there's an error, it is logged to the console.

**4. Handle Program Selection**

handleSelect(event) {

this.selectedProgramId = event.target.dataset.id;

}

* **handleSelect**: This method is triggered when a user clicks on one of the program buttons in the UI (from the previous markup you shared). The button's data-id attribute holds the ID of the selected program.
  + **event.target.dataset.id**: This accesses the data-id attribute of the clicked button to get the ID of the selected program.
  + The ID is then stored in the selectedProgramId property, marking it as the program selected by the user for deletion.

**5. Handle Deletion**

handleDelete() {

if (this.selectedProgramId) {

deleteSportsProgram({ recordId: this.selectedProgramId })

.then(() => refreshApex(this.programs))

.catch(error => console.error(error));

}

}

* **handleDelete**: This method is triggered when the user clicks the "Delete" button. It performs the following steps:
  1. **Check if a program is selected**: If selectedProgramId is set (i.e., a program has been selected), it proceeds to delete the selected program.
  2. **deleteSportsProgram({ recordId: this.selectedProgramId })**: This calls the deleteSportsProgram Apex method, passing the selected program’s ID. It returns a Promise.
  3. **.then(() => refreshApex(this.programs))**: After successfully deleting the program, it calls refreshApex(this.programs) to refresh the list of sports programs (programs) and update the UI with the latest data.
  4. **.catch(error => console.error(error))**: If there’s an error during the deletion, it logs the error to the console.

**Summary of Functionality**

1. **Fetching Data**: The component fetches a list of sports programs from the getSportsProgram Apex method using the @wire decorator. When the data is returned, it updates the programs property, which is then used to display the list of programs in the UI.
2. **Selecting a Program**: When a user clicks on a program button, the handleSelect method is invoked. It stores the ID of the selected program in selectedProgramId.
3. **Deleting a Program**: If a program is selected, the user can click the "Delete" button, which triggers the handleDelete method. This method calls the deleteSportsProgram Apex method to delete the program. After deletion, the list of programs is refreshed using refreshApex.

**Key Concepts Used:**

* **@wire**: Automatically fetches data and handles updating the component when the data changes.
* **@track**: Makes properties reactive, ensuring the component re-renders when their values change.
* **Apex Integration**: Calls Apex methods to fetch and delete data from Salesforce.
* **refreshApex**: Refreshes the data after a DML operation (e.g., delete) to ensure the UI stays in sync with the backend.

**Conclusion**

This LWC manages the UI for listing sports programs and deleting them. It utilizes the @wire service for data fetching, tracks user interactions for selecting programs, and invokes Apex methods for deleting records while keeping the UI in sync with the backend data.