Of course. Here is a comprehensive Salesforce developer interview kit designed for a senior developer with 8 years of experience who is returning after a one-year break.
This kit focuses on advanced topics, best practices, and hands-on scenarios to help you successfully clear your interviews.

# 🧠 Core Concepts Refresher

This section is a high-level review of critical areas. Given your experience, you know these concepts, but a refresher on the nuances is key.

## Apex

- **Governor Limits & Bulkification:** This is non-negotiable. Always think in terms of bulk. The core limits to keep top-of-mind are **SOQL queries (100 sync, 200 async)**, **DML statements (150)**, and **CPU time (10,000 ms sync, 60,000 ms async)**. The best practice is to never have SOQL or DML inside a loop. Use Maps to hold query results for efficient data access and to update records.
- **Trigger Frameworks:** At a senior level, you're expected to use a trigger framework. Don't write logic directly in the .trigger file. Your trigger should only delegate to a handler class. This makes the code organized, reusable, and allows you to control the order of execution.
    - **Key Principles:** One trigger per object, logic-less triggers, and context-specific handler methods (e.g., beforeInsert, afterUpdate).
- **Asynchronous Apex:** Knowing when to use which tool is crucial.
    - **Future (@future):** Simple, fire-and-forget. Use for making callouts from triggers or when you need to start an operation in its own transaction. **Limitation:** Can only accept primitive data types as parameters.
    - **Queueable Apex:** The modern replacement for Future methods. It's more powerful. **Advantages:** Can accept non-primitive types (like sObjects), can chain jobs (one Queueable can start another), and you get a Job ID to monitor its status. This is your default choice for most async processing.
    - **Batch Apex:** For processing thousands or millions of records. It breaks the job into manageable chunks. Implement the Database.Batchable interface (start, execute, finish). Use Database.Stateful to maintain state across chunks.
    - **Schedulable Apex:** To run Apex at a specific time. Implement the Schedulable interface. It's essentially a way to programmatically schedule a job that often calls a Batch or Queueable class.
- **SOQL/SOSL Optimization:**
    - Use selective queries with indexed fields in the WHERE clause (Id, Name, External IDs, custom fields marked as External ID).
    - Avoid querying for fields you don't need.
    - Use the SOQL Query Planner in the Developer Console to analyze query performance.
    - For searching across multiple objects in a single query based on a text string, use **SOSL**.

## Lightning Web Components (LWC)

- **Lifecycle Hooks:** These hooks allow you to execute code at specific phases of a component's life.
    - constructor(): Called when the component is created. Don't access attributes here.
    - connectedCallback(): Called when the component is inserted into the DOM. This is where you'd typically fetch data or set up listeners.
    - renderedCallback(): Called after the component's template is rendered. Use it for interacting with the DOM, but be careful of infinite loops.
    - disconnectedCallback(): Called when the component is removed from the DOM. Use for cleanup tasks.
- **Component Communication:**
    - **Parent-to-Child:** Use public properties decorated with @api.
    - **Child-to-Parent:** Dispatch a CustomEvent. The parent listens for the event with an on<eventname> handler.
    - **Unrelated Components:** Use the **Lightning Message Service (LMS)**. It's the standard, modern way. It works across the DOM, including with Aura and Visualforce pages. The older "pub-sub" model is a custom implementation that is no longer recommended.
- **Working with Salesforce Data:**
    - **@wire:** Declarative. Use this to read Salesforce data. It's great because it's cached and automatically provisions a stream of data (if the data changes, the component re-renders).
    - **Imperative Apex:** Call an Apex method on demand (e.g., in response to a button click). This is necessary for any DML operation (Create, Update, Delete) or for calling a method that you don't want to be cached.

# ❓ Conceptual Interview Questions & Answers

### Q1: You have a trigger on the Account object. In the same transaction, a workflow rule on the Account also fires and updates a field. Explain the order of execution and what potential issues you might face.

**Answer:** This question tests your understanding of the **Salesforce Order of Execution**.
The simplified order for this scenario is:

1. Original record is loaded.
2. before triggers execute.
3. System validation rules run.
4. Record is saved to the database (but not yet committed).
5. after triggers execute.
6. Workflow rules execute.
7. **If a workflow rule updates a field**, the entire process (before and after triggers) is fired **again**. However, it does not fire workflow rules again.
8. Roll-up summary fields are updated.
9. Transaction is committed.

**Potential Issue:** The biggest risk is hitting a **governor limit**, especially "Maximum trigger depth

exceeded" if the trigger logic itself causes another update that refires the trigger. A good trigger framework with recursion control (e.g., using a static boolean variable) is essential to prevent this. You also need to ensure your trigger logic is idempotent, meaning it can run multiple times without causing incorrect data changes.

### Q2: What's the difference between a Custom Setting and Custom Metadata? When would you use one over the other?

**Answer:** Both are used to store application configuration data, but Custom Metadata is the modern and more powerful choice.

- **Custom Settings:** They are similar to custom objects. There are two types: **List** (a reusable set of static data) and **Hierarchy** (allows you to personalize settings for specific profiles or users). The data in custom settings is treated as **data**, not metadata. This means you have to use DML to update it and it can't be deployed directly in a change set (you need a post-deployment script).
- **Custom Metadata Types:** This is the preferred method. The records are treated as **metadata**. This means they are deployable directly via change sets or other metadata API tools. You can query them in Apex without a SOQL limit penalty. They are also great for mapping data, like mapping a country code to a country name, or storing constants that might need to be updated by an admin without a code change.

**Rule of thumb:** Use **Custom Metadata Types** by default for all application configuration. Only use Hierarchy Custom Settings if you have a specific need to vary the configuration data by user or profile.

### Q3: You need to make a callout to an external REST API from a trigger. How do you do it and what are the limitations?

**Answer:** You cannot make a direct synchronous callout from a trigger because it would hold the database transaction open for an indeterminate amount of time, which is a major performance risk.

The correct approach is to **decouple the callout from the trigger transaction** by using asynchronous Apex.

1. The trigger identifies the records that need processing.
2. It then calls an asynchronous method (typically a **Queueable** class, but a **@future** method also works) and passes the IDs of the records.
3. The asynchronous method then performs the REST callout. This runs in a separate transaction, so it doesn't block the initial save operation.

**Code Example Snippet (in the trigger handler):**

```
// In the After Insert/Update block of the trigger handler
Set<Id> accountIdsForCallout = new Set<Id>();
for (Account acc : newRecords) {
    if (acc.Needs_Sync__c) {
        accountIdsForCallout.add(acc.Id);
    }
}

if (!accountIdsForCallout.isEmpty()) {
    // Enqueue the job to handle the callout
```

```
    System.enqueueJob(new
AccountCalloutQueueable(accountIdsForCallout));
}
```

Using a Queueable is better than a Future method because you can pass the Set<Id> directly and can monitor the job.

# 💻 Hands-On Apex Challenge 1: Trigger Framework

**Scenario:** When an Opportunity is updated to 'Closed Won', create a new Project__c custom object record linked to the Account. Additionally, update the Description field on all related Contacts of that Account to "Part of a winning project: [Opportunity Name]". The solution must be bulk-safe.

## Solution

This uses a simple trigger handler pattern.
**1. The Trigger (OpportunityTrigger.trigger)** This stays clean and simple, just delegating the work.

```
trigger OpportunityTrigger on Opportunity (after update) {
    if (Trigger.isAfter && Trigger.isUpdate) {
        OpportunityTriggerHandler.handleAfterUpdate(Trigger.new,
Trigger.oldMap);
    }
}
```

**2. The Handler Class (OpportunityTriggerHandler.cls)** This is where all the logic lives.

```
public class OpportunityTriggerHandler {
    public static void handleAfterUpdate(List<Opportunity> newOpps,
Map<Id, Opportunity> oldOppsMap) {
        Set<Id> accountIds = new Set<Id>();
        List<Project__c> projectsToCreate = new List<Project__c>();

        for (Opportunity opp : newOpps) {
            // Check if the stage changed to 'Closed Won'
            if (opp.StageName == 'Closed Won' &&
oldOppsMap.get(opp.Id).StageName != 'Closed Won') {
                // Prepare the new project for creation
                projectsToCreate.add(new Project__c(
                    Name = opp.Name + ' Project',
                    Account__c = opp.AccountId,
                    Opportunity__c = opp.Id,
                    Status__c = 'New'
                ));
                // Collect the Account ID for updating contacts
                if (opp.AccountId != null) {
                    accountIds.add(opp.AccountId);
```

```
                }
            }
        }

        // 1. DML Operation: Insert new Projects
        if (!projectsToCreate.isEmpty()) {
            insert projectsToCreate;
        }

        // 2. Bulk-update related contacts
        if (!accountIds.isEmpty()) {
            updateRelatedContacts(accountIds, newOpps);
        }
    }

    private static void updateRelatedContacts(Set<Id> accountIds,
List<Opportunity> closedWonOpps) {
        // Create a map of AccountId -> Opportunity Name for easy
lookup
        Map<Id, String> accountIdToOppNameMap = new Map<Id, String>();
        for(Opportunity opp : closedWonOpps) {
            if(opp.StageName == 'Closed Won' && opp.AccountId != null)
{
                accountIdToOppNameMap.put(opp.AccountId, opp.Name);
            }
        }

        List<Contact> contactsToUpdate = new List<Contact>();
        // Query for all contacts related to the winning
opportunities' accounts in one go
        for (Contact c : [SELECT Id, Description, AccountId FROM
Contact WHERE AccountId IN :accountIds]) {
            String oppName = accountIdToOppNameMap.get(c.AccountId);
            c.Description = 'Part of a winning project: ' + oppName;
            contactsToUpdate.add(c);
        }

        // DML Operation: Update all contacts at once
        if (!contactsToUpdate.isEmpty()) {
            update contactsToUpdate;
        }
    }
}
```

**3. The Test Class (OpportunityTriggerHandlerTest.cls)** This is critical. You must test your logic and ensure it passes.

```
@isTest
public class OpportunityTriggerHandlerTest {
```

```
    @isTest
    static void testOppClosedWonCreatesProjectAndUpdatesContacts() {
        // Setup test data
        Account testAcc = new Account(Name = 'Test Account');
        insert testAcc;

        Contact c1 = new Contact(LastName = 'Test', AccountId =
testAcc.Id);
        Contact c2 = new Contact(LastName = 'Contact', AccountId =
testAcc.Id);
        insert new List<Contact>{c1, c2};

        Opportunity opp = new Opportunity(
            Name = 'Big Deal',
            StageName = 'Prospecting',
            CloseDate = Date.today().addMonths(1),
            AccountId = testAcc.Id
        );
        insert opp;

        // Start the test context
        Test.startTest();

        // Perform the action that fires the trigger
        opp.StageName = 'Closed Won';
        update opp;

        Test.stopTest();
        // End the test context

        // Assertions: Verify the results
        // 1. Verify a Project was created
        List<Project__c> createdProjects = [SELECT Id FROM Project__c
WHERE Opportunity__c = :opp.Id];
        System.assertEquals(1, createdProjects.size(), 'A project
should have been created.');

        // 2. Verify contacts were updated
        List<Contact> updatedContacts = [SELECT Id, Description FROM
Contact WHERE AccountId = :testAcc.Id];
        for (Contact c : updatedContacts) {
            System.assertEquals('Part of a winning project: Big Deal',
c.Description, 'Contact description was not updated correctly.');
        }
    }
}
```

# 🚀 Hands-On LWC Challenge: Data Display & Imperative Apex

**Scenario:** Create an LWC that can be placed on an Account record page. It should display a list of related contacts. Each contact should have a "Send Welcome Email" button next to it. Clicking this button should call an Apex method to send a (mock) welcome email and then display a toast notification on the screen showing success.

## Solution

This solution demonstrates @api for getting the recordId, @wire for fetching data, and imperative Apex for performing the action.
**1. Apex Controller (ContactController.cls)** This provides the data and the action method.

```
public with sharing class ContactController {

    // @wire uses this method. It must be cacheable.
    @AuraEnabled(cacheable=true)
    public static List<Contact> getContacts(Id accountId) {
        return [
            SELECT Id, Name, Email
            FROM Contact
            WHERE AccountId = :accountId
            ORDER BY Name
            LIMIT 10
        ];
    }

    // Imperative method for the button click. Not cacheable as it
performs an action.
    @AuraEnabled
    public static String sendWelcomeEmail(Id contactId) {
        try {
            // In a real scenario, you would perform email sending
logic here.
            // For this example, we just simulate success.
            Contact c = [SELECT Name FROM Contact WHERE Id =
:contactId];
            System.debug('Sending welcome email to ' + c.Name);
            return 'Success';
        } catch (Exception e) {
            throw new AuraHandledException('Error sending email: ' +
e.getMessage());
        }
    }
}
```

**2. LWC HTML (contactList.html)** The template for displaying the list and buttons.

```
<template>
    <lightning-card title="Related Contacts"
icon-name="standard:contact">
        <div class="slds-m-around_medium">
            <template if:true={contacts.data}>
                <template for:each={contacts.data} for:item="contact">
                    <div key={contact.Id} class="slds-grid
slds-gutters slds-m-bottom_small">
                        <div class="slds-col slds-size_3-of-4">
                            <p>{contact.Name}</p>
                            <p
class="slds-text-body_small">{contact.Email}</p>
                        </div>
                        <div class="slds-col slds-size_1-of-4">
                             <lightning-button
                                label="Send Welcome Email"
                                variant="brand"
                                onclick={handleSendEmail}
                                data-contactid={contact.Id}>
                            </lightning-button>
                        </div>
                    </div>
                </template>
            </template>
            <template if:true={contacts.error}>
                <p>Error loading contacts.</p>
            </template>
             <template if:false={contacts.data}>
                <p>No contacts found.</p>
            </template>
        </div>
    </lightning-card>
</template>
```

**3. LWC JavaScript (contactList.js)** The component's logic.

```
import { LightningElement, api, wire } from 'lwc';
import getContacts from
'@salesforce/apex/ContactController.getContacts';
import sendWelcomeEmail from
'@salesforce/apex/ContactController.sendWelcomeEmail';
import { ShowToastEvent } from 'lightning/platformShowToastEvent';

export default class ContactList extends LightningElement {
    // Get the Account record ID from the page
    @api recordId;

    // Use @wire to get the contact data declaratively
```

```
    @wire(getContacts, { accountId: '$recordId' })
    contacts;

    // Handler for the button click
    handleSendEmail(event) {
        // Get the contactId from the data attribute on the button
        const contactId = event.target.dataset.contactid;

        // Call the Apex method imperatively
        sendWelcomeEmail({ contactId: contactId })
            .then(result => {
                this.dispatchEvent(
                    new ShowToastEvent({
                        title: 'Success',
                        message: 'Welcome email sent successfully!',
                        variant: 'success',
                    }),
                );
            })
            .catch(error => {
                this.dispatchEvent(
                    new ShowToastEvent({
                        title: 'Error',
                        message: error.body.message,
                        variant: 'error',
                    }),
                );
            });
    }
}
```

**4. LWC Meta XML (contactList.js-meta.xml)** This exposes the component to be used on a record page.

```
<?xml version="1.0" encoding="UTF-8"?>
<LightningComponentBundle
xmlns="http://soap.sforce.com/2006/04/metadata">
    <apiVersion>59.0</apiVersion>
    <isExposed>true</isExposed>
    <targets>
        <target>lightning__RecordPage</target>
    </targets>
    <targetConfigs>
        <targetConfig targets="lightning__RecordPage">
            <objects>
                <object>Account</object>
            </objects>
        </targetConfig>
    </targetConfigs>
```

```
</LightningComponentBundle>
```

# ✅ Final Tips for Interview Success

- **Talk About Your Gap Positively:** You weren't idle; you took a strategic break for personal reasons/professional development/etc. Frame it as a planned event. Mention that you've been actively refreshing your skills on Trailhead and personal dev orgs to stay current.
- **Acknowledge New Features:** Show you're up-to-date. Casually mention things like **Dynamic Forms**, the **@AuraEnabled(continuation=true)** for long-running callouts, or recent updates to Flow that reduce the need for Apex in some scenarios. This shows you're engaged with the platform's evolution.
- **Think Aloud:** During the coding challenge, explain your thought process. "Okay, first I need to get the records. I'll use a Set for the IDs to make sure my query is efficient. I'll use a Map to process the results to avoid nested loops. This approach is bulk-safe because..."
- **Ask Insightful Questions:** When they ask if you have questions, ask about their development process. "What does your code review process look like?", "How do you handle CI/CD?", or "What's the most interesting technical challenge the team has solved recently?". This shows you think like a senior developer.

Good luck! Your experience is highly valuable, and this refresher should give you the confidence to ace your interviews.