Of course. Here is a comprehensive Senior Salesforce Developer interview preparation kit, designed specifically for someone with 8 years of experience returning to a development role after a gap. This kit focuses on the depth of understanding, architectural thinking, and hands-on coding expected at a senior level.

---

Salesforce Senior Developer Interview Kit

Welcome back! A one-year gap is negligible with your 8 years of experience. This kit is designed to quickly refresh your core knowledge, update you on key modern concepts, and provide practical coding exercises. The focus for a senior role is less on syntax and more on design patterns, performance, scalability, and best practices.

Part 1: Core Concept Refresh & Modern Ecosystem (What's Changed/Hot?)

Even in a year, the ecosystem evolves. Be prepared to discuss these:

1. Salesforce Functions & Heroku Integration: Serverless compute. Know the use cases (heavy processing off-platform).

2. Event-Driven Architecture: Understand the difference between Platform Events, Change Data Capture (CDC), and Streaming API. Know when to use which.

3. Integration Patterns: Deep dive into RESTful API design, bulkification in integrations, and the use of Named Credentials.

4. Security (A MUST): Be sharp on Object-Level, Field-Level, and Record-Level security (Sharing Rules, Apex Sharing, without sharing). Be ready to explain the implications of with sharing and without sharing in different contexts.

5. Governor Limits: This is non-negotiable. You should be able to recite key limits (SOQL, DML, Heap, CPU) and, more importantly, strategies to avoid hitting them (bulkification, selective queries, efficient loops).

6. DX & DevOps: You must be comfortable talking about Salesforce DX (scratch orgs, source-driven development), version control (Git), and CI/CD pipelines (Azure DevOps, Jenkins, Copado, Gearset). Know the basics of packaging (Unlocked, Managed).

---

Part 2: Apex Hands-On Coding Examples & Solutions

Expectation: You will be asked to write code on a shared editor (CoderPad, etc.). Focus on clean, bulkified, secure code. Comment your thought process.

Problem 1: The Account Trigger & Service Layer (Senior Level) Scenario:Write a trigger on Account that, when an Account's Industry is updated to 'Banking', creates a related Opportunity for that Account. Ensure it's bulk-safe and follows best practices.

Solution Approach:

1. Use a Trigger Framework pattern. Mention this. For the interview, you can write a simplified handler.

2. Bulkify: Operate on Trigger.new, not single records.

3. Avoid SOQL/DML in loops.

4. Use a Service Class to separate business logic.

(Code Solution)

```apex

// AccountTrigger.trigger

trigger AccountTrigger on Account (after update) {

AccountTriggerHandler handler = new AccountTriggerHandler();

if (Trigger.isAfter && Trigger.isUpdate) {

handler.onAfterUpdate(Trigger.new, Trigger.oldMap);

}

}

// AccountTriggerHandler.cls

public with sharing class AccountTriggerHandler {

public void onAfterUpdate(List<Account> newAccounts, Map<Id, Account> oldMap) {

// 1. Identify which accounts need an Opportunity

List<Account> bankingAccounts = new List<Account>();

for (Account acc : newAccounts) {

if (acc.Industry == 'Banking' && oldMap.get(acc.Id).Industry != 'Banking') {

bankingAccounts.add(acc);

}

}

// 2. Delegate to a service layer for complex logic

if (!bankingAccounts.isEmpty()) {

AccountService.createOpportunitiesForBankingAccounts(bankingAccounts);

}

}

}

// AccountService.cls (The core logic - separates concerns)

public with sharing class AccountService {

public static void createOpportunitiesForBankingAccounts(List<Account> accounts) {

List<Opportunity> oppsToCreate = new List<Opportunity>();

for (Account acc : accounts) {

Opportunity newOpp = new Opportunity();

newOpp.Name = acc.Name + ' - Banking Opportunity';

newOpp.CloseDate = System.today().addDays(30);

newOpp.StageName = 'Prospecting';

newOpp.AccountId = acc.Id;

oppsToCreate.add(newOpp);

}

if (!oppsToCreate.isEmpty()) {

// Insert outside the loop - BULKIFIED

insert oppsToCreate;

}

}

}

```

Follow-up Questions:

· "How would you test this? Write a test class." (Cover positive, negative, bulk scenarios)

· "What governor limits are most relevant here?" (SOQL query limits, DML statement limits)

· "What if the service method needed to call an external system?"

---

Problem 2: Asynchronous Apex & Queueable Chaining Scenario:You need to process 10,000 records in batches, and after all batches are processed, send a single summary email. How would you architect this?

Solution Approach:

1. Use Queueable Apex for its higher limit and chaining capability.

2. Implement a pattern where each queueable job processes a batch and then chains the next job if more records exist.

3. The final job in the chain sends the email.

(Conceptual Code Snippet)

```apex

public class BatchProcessor implements Queueable, Database.AllowsCallouts {

private Integer startingIndex;

private static Integer BATCH\_SIZE = 1000;

private static Integer totalProcessed = 0;

public BatchProcessor(Integer start) {

this.startingIndex = start;

}

public void execute(QueueableContext ctx) {

// 1. Query for records using OFFSET or a more efficient WHERE clause

List<MyObject\_\_c> records = [SELECT Id, Name FROM MyObject\_\_c LIMIT :BATCH\_SIZE OFFSET :startingIndex];

// 2. Process this batch of records

totalProcessed += records.size();

processRecords(records);

// 3. Check if more records exist

if (records.size() == BATCH\_SIZE) {

// Chain the next job

System.enqueueJob(new BatchProcessor(startingIndex + BATCH\_SIZE));

} else {

// This is the final job. Send email.

Messaging.SingleEmailMessage mail = new Messaging.SingleEmailMessage();

mail.setSubject('Processing Complete');

mail.setPlainTextBody('Processed ' + totalProcessed + ' records successfully.');

mail.setToAddresses(new List<String>{'admin@company.com'});

Messaging.sendEmail(new List<Messaging.SingleEmailMessage>{mail});

}

}

private void processRecords(List<MyObject\_\_c> records) {

// Your complex processing logic here

for (MyObject\_\_c rec : records) {

rec.Processed\_\_c = true;

}

update records; // Bulk DML

}

}

// To start the chain: System.enqueueJob(new BatchProcessor(0));

```

Follow-up Questions:

· "Why did you choose Queueable over @future or Batch Apex?" (Queueable supports chaining and complex state, unlike @future. It's more granular than Batch Apex for this use case).

· "How would you handle a failure in one of the chained jobs?"

---

Part 3: LWC Hands-On Coding Examples & Solutions

Expectation: Demonstrate modern JavaScript (ES6+), understanding of the Lightning Data Service, and communication patterns.

Problem 1: Parent-to-Child and Child-to-Parent Communication Scenario:Build a simple UI. A parent component has a button "Show Details". When clicked, it should fetch data and display it in a child component. The child component should have a "Close" button that hides the details in the parent.

Solution Approach:

1. Use a public property (@api) in the child to receive data from the parent.

2. Use a custom event fired from the child to notify the parent to hide the details.

(Code Solution)

```jsx

<!-- parentComponent.html -->

<template>

<lightning-card title="Parent Component">

<lightning-button label="Show Details" onclick={handleShowDetails}></lightning-button>

<template if:true={showDetails}>

<c-child-component

details={accountDetails}

onclose={handleCloseDetails}>

</c-child-component>

</template>

</lightning-card>

</template>

```

```javascript

// parentComponent.js

import { LightningElement, track } from 'lwc';

import getAccountData from '@salesforce/apex/AccountController.getAccountData';

export default class ParentComponent extends LightningElement {

@track showDetails = false;

@track accountDetails;

handleShowDetails() {

getAccountData()

.then(data => {

this.accountDetails = data;

this.showDetails = true;

})

.catch(error => {

// Handle error

});

}

handleCloseDetails() {

this.showDetails = false;

this.accountDetails = undefined;

}

}

```

```jsx

<!-- childComponent.html -->

<template>

<lightning-card title="Details">

<p>{details}</p>

<lightning-button label="Close" onclick={handleClose}></lightning-button>

</lightning-card>

</template>

```

```javascript

// childComponent.js

import { LightningElement, api } from 'lwc';

export default class ChildComponent extends LightningElement {

@api details; // Public property to receive data from parent

handleClose() {

// Create and dispatch a custom event named 'close'

const closeEvent = new CustomEvent('close');

this.dispatchEvent(closeEvent);

}

}

```

Follow-up Questions:

· "How would you pass data from the child to the parent in the event?" (Use event.detail)

· "When would you use @wire vs. calling an Apex method imperatively?"

---

Problem 2: Using Lightning Data Service (LDS) and Wire Service Scenario:Create a component that displays the Name and Industry of an Account using the Salesforce-provided UI record API, without writing Apex.

(Code Solution)

```jsx

<!-- displayAccountRecord.html -->

<template>

<template if:true={account.data}>

<lightning-card title="Account Details" icon-name="standard:account">

<div class="slds-p-around\_medium">

<p>Account Name: {account.data.fields.Name.value}</p>

<p>Industry: {account.data.fields.Industry.value}</p>

</div>

</lightning-card>

</template>

<template if:true={account.error}>

<p>Error loading account: {account.error}</p>

</template>

</template>

```

```javascript

// displayAccountRecord.js

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

import { getRecord, getFieldValue } from 'lightning/uiRecordApi';

// Import the schema fields

import NAME\_FIELD from '@salesforce/schema/Account.Name';

import INDUSTRY\_FIELD from '@salesforce/schema/Account.Industry';

const FIELDS = [NAME\_FIELD, INDUSTRY\_FIELD];

export default class DisplayAccountRecord extends LightningElement {

@api recordId; // This component expects to be placed on an Account record page

// Wire the record adapter to the property 'account'

@wire(getRecord, { recordId: '$recordId', fields: FIELDS })

account;

}

```

Follow-up Questions:

· "What are the performance benefits of using LDS?" (Caching, fewer server calls)

· "How would you update the record from this component?" (Use updateRecord from lightning/uiRecordApi)

· "When would you not use LDS?" (When you need complex business logic or data aggregation that Apex provides)

---

Part 4: System Design & Architecture Questions (Senior Level)

Be prepared to whiteboard and discuss these scenarios:

1. Design a complex application: "Design a Vacation Request approval system for 10,000 employees. It integrates with an external HR system for final data of record. Consider UI, Apex, Flows, approval processes, and integration."

· What to discuss: Object model (Request, Day, etc.), Flow for UI vs. LWC, Apex Triggers for automation, Platform Events/Outbound Messages for integration, reporting.

2. Performance Troubleshooting: "Users complain a page is slow. How do you investigate?"

· Answer: Check browser dev tools (network tab). Check Salesforce debug logs (looking for SOQL/DML in loops, non-selective queries). Use Salesforce Optimizer. Check for large view states in Visualforce. Check indexing on filtered fields.

3. Large Data Volume (LDV): "How would you handle a report on 5 million Opportunity records that needs to be grouped by a custom field?"

· Answer: Avoid SOQL reports. Use Big Objects (if the grouping field is stable). Use an external data warehouse (Snowflake, BigQuery) with ETL tools (MuleSoft, Kafka). Use async reporting where the result is emailed later.

Part 5: Behavioral Questions

Your experience is your strength. Use the STAR method (Situation, Task, Action, Result).

· "Tell me about the most complex architecture you designed."

· "Describe a time you had a major production issue. How did you handle it?"

· "How do you mentor junior developers and ensure code quality on your team?"

· "Tell me about a time you had to disagree with an architect or product owner. How did you handle it?"

Final Tip: Practice talking through your code and design decisions aloud. A senior developer is expected to be a force multiplier, and communication is key to that. Good luck! You've got this.