Senior Salesforce Developer Interview Kit (Detailed Version)

This detailed kit expands on the previous version, providing a more in-depth set of questions, answers, and hands-on examples for interviewing a senior Salesforce developer. It covers advanced topics in Apex, Lightning Web Components (LWC), and system design.

Part 1: Advanced Apex Questions & Answers

Question 1: Explain the purpose of the WITH SECURITY_ENFORCED clause in SOQL and why it is a best practice for senior developers.

Answer: WITH SECURITY_ENFORCED is a security-oriented clause in SOQL that automatically enforces field-level security (FLS) and object-level security (OLS) on the query results. When this clause is used, the query will only return fields and objects that the running user has access to. If the user does not have access to a requested field or object, a System.QueryException is thrown.
Why it's a best practice:

- **Prevents Data Leaks:** It helps prevent accidental data exposure by ensuring that the code respects the security settings of the user. This is crucial for applications that might be used by different user profiles with varying levels of access.
- **Reduces Boilerplate Code:** Before this clause, developers had to manually check for FLS using Schema.SObject.isAccessible() and Schema.SObject.getDescribe().fields.getMap().get('FieldName').getDescribe().isAccessible(), which was verbose and error-prone.
- **Secure by Default:** It promotes a "secure by default" coding pattern, making the code more robust and less susceptible to security vulnerabilities.

Question 2: Describe a scenario where you would use Queueable Apex instead of @future methods. Provide a code example.

Answer: Queueable Apex is preferred over @future methods when you need more control and flexibility over asynchronous processes. A key advantage is the ability to chain jobs. This is essential for complex, multi-step asynchronous operations.
Scenario: You need to process a large number of Case records and, after updating them, you need to call a web service to notify an external system. This requires two distinct asynchronous operations.
Code Solution:

- **Apex Class for Queueable Job**

<!-- end list -->

```
// A Queueable class to update a batch of Cases
public class CaseUpdateQueueable implements Queueable {
    private List<Id> caseIds;

    public CaseUpdateQueueable(List<Id> caseIds) {
        this.caseIds = caseIds;
    }
    public void execute(QueueableContext context) {
        List<Case> casesToUpdate = new List<Case>();
        for (Id caseId : this.caseIds) {
            casesToUpdate.add(new Case(Id = caseId, Status =
'Escalated'));
        }

        if (!casesToUpdate.isEmpty()) {
            update casesToUpdate;
        }

        // Chain to the next job
        System.enqueueJob(new
WebServiceCalloutQueueable(this.caseIds));
    }
}

// A Queueable class for the subsequent web service callout
public class WebServiceCalloutQueueable implements Queueable,
Database.AllowsCallouts {
    private List<Id> caseIds;

    public WebServiceCalloutQueueable(List<Id> caseIds) {
        this.caseIds = caseIds;
    }

    public void execute(QueueableContext context) {
        // Assume an HTTP callout to an external service
        Http http = new Http();
        HttpRequest request = new HttpRequest();
        request.setEndpoint('https://external.api/notify');
        request.setMethod('POST');
        request.setHeader('Content-Type',
'application/json;charset=UTF-8');
        request.setBody('{"caseIds":' + JSON.serialize(this.caseIds) +
'}');

        try {
            HttpResponse response = http.send(request);
            if (response.getStatusCode() != 200) {
                // Log or handle the error
                System.debug('Error calling external service: ' +
response.getBody());
            }
        } catch (Exception e) {
            // Log or handle the exception
            System.debug('Exception during callout: ' +
e.getMessage());
        }
    }
}

// To start the process from an imperative method or trigger
// System.enqueueJob(new CaseUpdateQueueable(caseIdsToProcess));
```

Key Advantage: Unlike @future methods, Queueable allows for chaining jobs, which is crucial here to ensure the callout happens only after the case updates are committed.

Part 2: Advanced LWC Questions & Answers

Question 1: How do you handle communication between two LWC components that are not in a parent-child relationship?

Answer: You would use the Lightning Message Service (LMS). It allows communication between components across the Lightning Page, whether they are on the same page, in different containers, or even in a parent-child relationship.
Steps to implement LMS:

1. **Create a Message Channel:** Define a message channel metadata file (.messageChannel) to specify the fields of the message payload.
2. **Import:** In the components, import the message channel, along with publish, subscribe, and unsubscribe from the lightning/messageService module.
3. **Publishing Component:** Use publish() to send a message to the channel.
4. **Subscribing Component:** Use subscribe() to listen for messages and a handler function to process the received message. unsubscribe() should be used to clean up the subscription, typically in the disconnectedCallback() lifecycle hook.

Code Example:

- **myMessageChannel.messageChannel-meta.xml**

<!-- end list -->

```
<?xml version="1.0" encoding="UTF-8"?>
<LightningMessageChannel
xmlns="http://soap.sforce.com/2006/04/metadata">
    <description>This is a sample Lightning Message Channel for
communication between components.</description>
    <isExposed>true</isExposed>
    <lightningMessageFields>
        <fieldName>recordId</fieldName>
        <description>The ID of the record that has been
selected.</description>
    </lightningMessageFields>
    <lightningMessageFields>
        <fieldName>message</fieldName>
        <description>A message to display.</description>
    </lightningMessageFields>
    <masterLabel>MyMessageChannel</masterLabel>
```

```
</LightningMessageChannel>
```

- **publisherComponent.js**

<!-- end list -->

```
import { LightningElement, api } from 'lwc';
import { publish, MessageContext } from 'lightning/messageService';
import RECORD_SELECTED_CHANNEL from
'@salesforce/messageChannel/myMessageChannel__c';

export default class PublisherComponent extends LightningElement {
    @api recordId;

    @api handleRecordSelection() {
        const payload = {
            recordId: this.recordId,
            message: 'A record has been selected'
        };
        publish(this.messageContext, RECORD_SELECTED_CHANNEL,
payload);
    }
}
```

- **subscriberComponent.js**

<!-- end list -->

```
import { LightningElement, wire } from 'lwc';
import { subscribe, MessageContext, unsubscribe } from
'lightning/messageService';
import RECORD_SELECTED_CHANNEL from
'@salesforce/messageChannel/myMessageChannel__c';

export default class SubscriberComponent extends LightningElement {
    subscription = null;
    receivedMessage = '';

    @wire(MessageContext)
    messageContext;

    connectedCallback() {
        this.subscribeToMessageChannel();
    }

    disconnectedCallback() {
        unsubscribe(this.subscription);
        this.subscription = null;
    }

    subscribeToMessageChannel() {
        if (!this.subscription) {
            this.subscription = subscribe(
                this.messageContext,
                RECORD_SELECTED_CHANNEL,
                (message) => this.handleMessage(message)
            );
        }
    }

    handleMessage(message) {
        this.receivedMessage = 'Record ID: ' + message.recordId + ', 
Message: ' + message.message;
    }
}
```

Question 2: How do you handle errors and display them to the user in LWC? Provide an example.

Answer: Effective error handling is crucial for a good user experience. In LWC, errors from Apex can be caught using try...catch blocks or by checking the error property of the @wire service.
Example Scenario: A component calls an Apex method that might fail due to a DML exception or a validation rule. We need to display the error message gracefully.
Code Solution:

- **accountCreator.js**

<!-- end list -->

```
import { LightningElement } from 'lwc';
import createAccount from 
'@salesforce/apex/AccountCreatorController.createAccount';
import { ShowToastEvent } from 'lightning/platformShowToastEvent';

export default class AccountCreator extends LightningElement {
    accountName = '';
    showError = false;
    errorMessage = '';

    handleNameChange(event) {
        this.accountName = event.target.value;
    }

    async handleSave() {
        this.showError = false;
        this.errorMessage = '';

        try {
            const newAccountId = await createAccount({ accountName: 
this.accountName });
            const toastEvent = new ShowToastEvent({
                title: 'Success',
                message: 'Account created successfully!',
                variant: 'success'
            });
            this.dispatchEvent(toastEvent);
            // Optionally, reset the form or navigate
        } catch (error) {
            this.showError = true;
            this.errorMessage = 'An error occurred: ' +
this.getErrorMessage(error);
            const toastEvent = new ShowToastEvent({
                title: 'Error',
                message: this.getErrorMessage(error),
                variant: 'error'
            });
            this.dispatchEvent(toastEvent);
        }
    }

    getErrorMessage(error) {
        if (error.body && error.body.message) {
            return error.body.message;
        } else if (error.message) {
            return error.message;
        } else {
            return 'An unknown error occurred.';
        }
    }
}
```

- **accountCreator.html**

<!-- end list -->

```
<template>
    <lightning-card title="Create Account"
icon-name="standard:account">
        <div class="slds-m-around_medium">
            <lightning-input
                label="Account Name"
                value={accountName}
                onchange={handleNameChange}
            ></lightning-input>
            <lightning-button
                label="Save"
                onclick={handleSave}
                class="slds-m-top_small"
            ></lightning-button>
        </div>
        <template if:true={showError}>
            <div class="slds-m-around_medium slds-text-color_error">
                <p>{errorMessage}</p>
            </div>
        </template>
    </lightning-card>
</template>
```

Explanation: The code uses an async/await pattern with try...catch to handle the asynchronous Apex call. In the catch block, it extracts the error message from the response body and displays it both as a visible message within the component and as a ShowToastEvent for a better user experience.

Part 3: Senior-Level Design & Architecture Questions

Question 1: You are tasked with migrating a large volume of data into Salesforce. What are the key considerations and steps you would take to ensure a smooth and efficient migration?

Answer: A senior developer would approach this with a well-defined strategy:

1. **Data Assessment & Cleansing:** Analyze the source data for quality, identify duplicates, and define a data cleansing plan. This is a critical first step to avoid bringing "dirty" data into Salesforce.
2. **Migration Strategy:** Choose the right tool for the job. For large volumes, consider using data loaders like the Data Loader command-line interface, third-party tools (e.g., Informatica, MuleSoft), or custom Apex batch jobs if transformations are required.
3. **Order of Operations:** Plan the migration in the correct sequence to respect object dependencies (e.g., Accounts first, then Contacts, then Opportunities).
4. **Bulk API vs. REST/SOAP API:** Favor the Bulk API for large-scale data loads. It is optimized for high volumes, bypassing most governor limits and significantly improving performance by processing data asynchronously in batches.
5. **Disable Automation:** Temporarily disable triggers, validation rules, and workflow rules during the import to prevent performance bottlenecks and unexpected failures. Re-enable them after the migration is complete.
6. **Load in Chunks:** Break down large data sets into manageable chunks to prevent hitting governor limits and to make it easier to restart the process if a failure occurs.
7. **Error Logging & Monitoring:** Implement a robust error logging mechanism to track and report failed records.
8. **Validation & Reconciliation:** After the migration, perform a thorough validation to ensure all records were successfully imported and the data integrity is maintained. This includes reconciling record counts and key field values.

Question 2: How do you implement a robust, reusable, and testable trigger framework in a large Salesforce org?

Answer: A well-designed trigger framework is essential for managing complexity and preventing trigger-related issues.

Framework Components:

- **Trigger Handler Class:** A single trigger on each object that calls a corresponding Apex handler class. This decouples the trigger from the business logic.
- **Context-Specific Methods:** The handler class should have methods for each trigger event (before insert, before update, after insert, etc.). This allows developers to add logic to a specific context without modifying the trigger itself.
- **Recursion Control:** Use a static variable within the handler class to prevent recursive trigger execution.
- **Configuration & Bypass:** Implement a way to bypass trigger logic for specific scenarios (e.g., data loads) using Custom Metadata Types or Custom Settings.
- **Bulkification:** The framework should be designed to process collections of records, not individual records, to handle bulk operations efficiently.

Code Example (Conceptual):

```
// Trigger on Account
trigger AccountTrigger on Account (before insert, before update, after
insert, after update, before delete, after delete) {
    // Call the handler class based on the trigger context
    new AccountTriggerHandler().execute();
}

// AccountTriggerHandler.cls
public with sharing class AccountTriggerHandler {
    private static Set<Id> processedRecordIds = new Set<Id>();

    // Main method to route execution
    public void execute() {
        if (Trigger.isBefore) {
            if (Trigger.isInsert) {
                beforeInsert(Trigger.new);
            } else if (Trigger.isUpdate) {
                beforeUpdate(Trigger.new, Trigger.oldMap);
            }
            // Add other before events
        } else if (Trigger.isAfter) {
            if (Trigger.isInsert) {
                afterInsert(Trigger.new);
            }
            // Add other after events
        }
    }

    private void beforeInsert(List<Account> newAccounts) {
        // Business logic for before insert
    }

    private void afterInsert(List<Account> newAccounts) {
        // Business logic for after insert
        // Check for recursion
        for (Account acc : newAccounts) {
            if (!processedRecordIds.contains(acc.Id)) {
                // Call other methods, e.g., for async processing
                processedRecordIds.add(acc.Id);
            }
        }
    }
    // Add other methods for different contexts
}
```

This framework ensures that all business logic is in a single, well-defined class, making it easier to manage, debug, and test. It separates the "what" (the trigger event) from the "how" (the business logic).