Salesforce Developer Interview Kit - v1
Core Salesforce Concepts & Apex Programming
For developers with 8+ years experience returning to development roles

---

Table of Contents

Core Salesforce Concepts

Q1: Explain the different types of relationships in Salesforce and their limits.
Answer:
- **Master-Detail**: Child inherits security and sharing from parent. Roll-up summary fields possible. Max 2 master-detail relationships per object.
- **Lookup**: Independent objects, no inheritance. Can be optional or required.
- **Many-to-Many**: Implemented via junction objects with 2 master-detail relationships.
- **Hierarchical**: Self-referencing lookup (only on User object).
- **External Lookup**: Links to external data sources.

Limits: Max 40 custom relationships per object, 25 total master-detail relationships in an org.

Q2: What are the different types of Flows and when would you use each?
Answer:
- **Screen Flow**: User interaction with screens, forms, and input validation
- **Autolaunched Flow**: Background processes triggered by automation
- **Record-Triggered Flow**: Replaces workflow rules and process builder
- **Platform Event-Triggered Flow**: Responds to platform events
- **Schedule-Triggered Flow**: Time-based automation

Use Screen Flows for guided user experiences, Record-Triggered for real-time automation, and Autolaunched for complex business logic.

Q3: Explain Governor Limits and how to optimize against them.
Answer:
Key limits per transaction:
- 100 SOQL queries
- 50,000 records from SOQL
- 150 DML statements

- 10,000 DML rows
- 6MB heap size
- 10 minutes CPU time

Optimization strategies:
- Bulkify code (process collections, not single records)
- Use selective SOQL queries with proper indexing
- Implement efficient exception handling
- Use asynchronous processing for heavy operations
- Leverage platform cache when appropriate

Apex Programming Fundamentals

Q4: What are the different execution contexts in Apex?
Answer:
- **Anonymous**: Developer Console, Data Loader
- **Trigger**: Before/After DML operations
- **Web Service**: REST/SOAP callouts
- **Visualforce**: Page controllers and extensions
- **Lightning**: Component controllers
- **Batch**: Asynchronous processing
- **Scheduled**: Time-based execution
- **Future**: Asynchronous callouts
- **Queueable**: Enhanced asynchronous processing

Q5: Explain Trigger Design Patterns and best practices.
Answer:
One Trigger Per Object Pattern:
```apex
trigger AccountTrigger on Account (before insert, before update, after insert, after update) {
    AccountTriggerHandler.handleTrigger();
}
```

Handler Class:
```apex
public class AccountTriggerHandler {
    public static void handleTrigger() {
        if (Trigger.isBefore) {
            if (Trigger.isInsert) beforeInsert();
            if (Trigger.isUpdate) beforeUpdate();
        }
```

```
        if (Trigger.isAfter) {
            if (Trigger.isInsert) afterInsert();
            if (Trigger.isUpdate) afterUpdate();
        }
    }

    private static void beforeInsert() {
        AccountService.validateAccounts(Trigger.new);
    }

    private static void beforeUpdate() {
        AccountService.updateRelatedRecords(Trigger.newMap, Trigger.oldMap);
    }
}
```

Q6: Implement a bulk-safe trigger for updating related Contact records.
Answer:
```apex
// Trigger
trigger AccountTrigger on Account (after update) {
    if (Trigger.isAfter && Trigger.isUpdate) {
        AccountTriggerHandler.updateRelatedContacts(Trigger.newMap, Trigger.oldMap);
    }
}

// Handler
public class AccountTriggerHandler {
    public static void updateRelatedContacts(Map<Id, Account> newMap, Map<Id, Account>
oldMap) {
        Set<Id> accountIds = new Set<Id>();

        // Collect accounts where Name changed
        for (Id accId : newMap.keySet()) {
            Account newAcc = newMap.get(accId);
            Account oldAcc = oldMap.get(accId);

            if (newAcc.Name != oldAcc.Name) {
                accountIds.add(accId);
            }
        }

        if (!accountIds.isEmpty()) {
            updateContactDescriptions(accountIds, newMap);
```

```
        }
    }

    private static void updateContactDescriptions(Set<Id> accountIds, Map<Id, Account>
accountMap) {
        List<Contact> contactsToUpdate = new List<Contact>();

        for (Contact con : [SELECT Id, AccountId, Description FROM Contact WHERE AccountId
IN :accountIds]) {
            con.Description = 'Updated for account: ' + accountMap.get(con.AccountId).Name;
            contactsToUpdate.add(con);
        }

        if (!contactsToUpdate.isEmpty()) {
            update contactsToUpdate;
        }
    }
}
```

Q7: Create a utility class for dynamic SOQL with proper error handling.
Answer:
```apex
public class DynamicSOQLUtil {

    public static List<SObject> queryRecords(String objectName, Set<String> fields, String
whereClause, Integer limitCount) {
        try {
            validateInputs(objectName, fields);

            String query = buildQuery(objectName, fields, whereClause, limitCount);
            System.debug('Executing query: ' + query);

            return Database.query(query);

        } catch (Exception e) {
            System.debug('Error in queryRecords: ' + e.getMessage());
            throw new DynamicSOQLException('Failed to execute dynamic query: ' +
e.getMessage());
        }
    }

    private static void validateInputs(String objectName, Set<String> fields) {
        if (String.isBlank(objectName)) {
```

```
            throw new DynamicSOQLException('Object name cannot be blank');
        }

        if (fields == null || fields.isEmpty()) {
            throw new DynamicSOQLException('Field set cannot be empty');
        }

        // Check object accessibility
        Schema.DescribeSObjectResult objDescribe =
Schema.getGlobalDescribe().get(objectName)?.getDescribe();
        if (objDescribe == null || !objDescribe.isAccessible()) {
            throw new DynamicSOQLException('Object not accessible: ' + objectName);
        }

        // Check field accessibility
        Map<String, Schema.SObjectField> fieldMap = objDescribe.fields.getMap();
        for (String field : fields) {
            Schema.SObjectField fieldToken = fieldMap.get(field.toLowerCase());
            if (fieldToken == null || !fieldToken.getDescribe().isAccessible()) {
                throw new DynamicSOQLException('Field not accessible: ' + field);
            }
        }
    }

    private static String buildQuery(String objectName, Set<String> fields, String whereClause,
Integer limitCount) {
        String query = 'SELECT ' + String.join(new List<String>(fields), ', ') +
                       ' FROM ' + objectName;

        if (String.isNotBlank(whereClause)) {
            query += ' WHERE ' + whereClause;
        }

        if (limitCount != null && limitCount > 0) {
            query += ' LIMIT ' + limitCount;
        }

        return query;
    }

    public class DynamicSOQLException extends Exception {}
}
```

Advanced Apex Patterns

Q8: Implement a Queueable class for processing large datasets.
Answer:
```apex
public class DataProcessorQueueable implements Queueable, Database.AllowsCallouts {

    private List<Id> recordIds;
    private Integer batchSize;
    private Integer currentIndex;
    private String processType;

    public DataProcessorQueueable(List<Id> recordIds, String processType) {
        this.recordIds = recordIds;
        this.processType = processType;
        this.batchSize = 100;
        this.currentIndex = 0;
    }

    public DataProcessorQueueable(List<Id> recordIds, String processType, Integer
currentIndex) {
        this.recordIds = recordIds;
        this.processType = processType;
        this.batchSize = 100;
        this.currentIndex = currentIndex;
    }

    public void execute(QueueableContext context) {
        try {
            Integer endIndex = Math.min(currentIndex + batchSize, recordIds.size());
            List<Id> currentBatch = new List<Id>();

            for (Integer i = currentIndex; i < endIndex; i++) {
                currentBatch.add(recordIds[i]);
            }

            processBatch(currentBatch);

            // Chain next batch if there are more records
            if (endIndex < recordIds.size()) {
                System.enqueueJob(new DataProcessorQueueable(recordIds, processType,
endIndex));
```

```
            }

        } catch (Exception e) {
            System.debug('Error in DataProcessorQueueable: ' + e.getMessage());
            // Log error or send notification
            logError(e.getMessage(), context.getJobId());
        }
    }

    private void processBatch(List<Id> batchIds) {
        switch on processType {
            when 'ACCOUNT_UPDATE' {
                processAccounts(batchIds);
            }
            when 'CONTACT_SYNC' {
                syncContacts(batchIds);
            }
            when 'EXTERNAL_CALLOUT' {
                makeExternalCallouts(batchIds);
            }
            when else {
                throw new ProcessorException('Unknown process type: ' + processType);
            }
        }
    }

    private void processAccounts(List<Id> accountIds) {
        List<Account> accountsToUpdate = [SELECT Id, Name, Description FROM Account
WHERE Id IN :accountIds];

        for (Account acc : accountsToUpdate) {
            acc.Description = 'Processed on ' + System.now();
        }

        update accountsToUpdate;
    }

    private void syncContacts(List<Id> contactIds) {
        // Implementation for contact sync
        List<Contact> contacts = [SELECT Id, Email, Phone FROM Contact WHERE Id IN
:contactIds];
        // Sync logic here
    }
```

```
    private void makeExternalCallouts(List<Id> recordIds) {
        for (Id recordId : recordIds) {
            HttpRequest req = new HttpRequest();
            req.setEndpoint('https://api.example.com/sync/' + recordId);
            req.setMethod('POST');

            Http http = new Http();
            HttpResponse res = http.send(req);

            // Process response
            if (res.getStatusCode() != 200) {
                System.debug('Callout failed for record: ' + recordId);
            }
        }
    }

    private void logError(String errorMessage, Id jobId) {
        // Custom error logging implementation
        System.debug('Job failed: ' + jobId + ', Error: ' + errorMessage);
    }

    public class ProcessorException extends Exception {}
}

// Usage example:
// List<Id> accountIds = new List<Id>{'001...', '001...'};
// System.enqueueJob(new DataProcessorQueueable(accountIds, 'ACCOUNT_UPDATE'));
```

Trigger Design & Best Practices

Key Principles for Senior Developers:

1. **One Trigger Per Object**: Avoid multiple triggers on the same object
2. **Handler Pattern**: Separate business logic from trigger context
3. **Bulkification**: Always process collections, never single records
4. **Recursion Prevention**: Implement static flags or trigger framework
5. **Exception Handling**: Graceful error handling with proper logging
6. **Testing**: Comprehensive test coverage with bulk scenarios

Common Anti-Patterns to Avoid:
- SOQL/DML inside loops

- Hardcoded IDs or values
- Lack of null checks
- Poor exception handling
- Non-bulkified logic
- Direct DML in triggers (use service classes)

Modern Best Practices:
- Use Record-Triggered Flows where appropriate
- Implement proper logging and monitoring
- Follow security best practices (with sharing)
- Use Custom Metadata for configuration
- Implement proper unit testing patterns

Next Steps

This is Part 1 of the comprehensive interview kit. Continue with:
- **v2**: Lightning Web Components & Integration Patterns
- **v3**: System Design, Advanced Topics & Recent Updates

Last Updated: August 2025
Author: Senior Salesforce Developer Interview Preparation