Salesforce Developer Interview Kit - v3
System Design, Advanced Topics & Recent Updates
For developers with 8+ years experience returning to development roles

---

Table of Contents

Hands-on Coding Challenges

Q15: Design a trigger framework that prevents recursive calls and supports multiple handlers.
Answer:
```apex
// Trigger Framework Handler
public abstract class TriggerHandler {

    // Static map to prevent recursion
    private static Map<String, LoopCount> loopCountMap;
    private static Set<String> bypassedHandlers;

    static {
        loopCountMap = new Map<String, LoopCount>();
        bypassedHandlers = new Set<String>();
    }

    // Current trigger context
    protected TriggerContext context;

    // Constructor
    public TriggerHandler() {
        this.setTriggerContext();
    }

    // Main entry point for triggers
    public void run() {
```

```
        // Check if this handler should be bypassed
        if (bypassedHandlers.contains(getHandlerName())) {
            return;
        }

        // Check for recursion
        if (this.context != null && this.context.isExecuting) {
            this.addToLoopCount();

            // Check max loop count
            if (this.context.isBefore && this.getMaxLoopCount() >= 0 && this.getLoopCount() >
this.getMaxLoopCount()) {
                String message = 'Maximum loop count of ' + String.valueOf(this.getMaxLoopCount())
+ ' reached in ' + getHandlerName();
                throw new TriggerException(message);
            }

            if (this.context.isAfter && this.getMaxLoopCount() >= 0 && this.getLoopCount() >
this.getMaxLoopCount()) {
                String message = 'Maximum loop count of ' + String.valueOf(this.getMaxLoopCount())
+ ' reached in ' + getHandlerName();
                throw new TriggerException(message);
            }

            // Execute the appropriate method
            if (this.context.isBefore) {
                this.beforeInsert();
                this.beforeUpdate();
                this.beforeDelete();
            }

            if (this.context.isAfter) {
                this.afterInsert();
                this.afterUpdate();
                this.afterDelete();
                this.afterUndelete();
            }
        }
    }

    // Bypass methods
    public static void bypass(String handlerName) {
        bypassedHandlers.add(handlerName);
    }
```

```
public static void clearBypass(String handlerName) {
    bypassedHandlers.remove(handlerName);
}

public static Boolean isBypassed(String handlerName) {
    return bypassedHandlers.contains(handlerName);
}

public static void clearAllBypasses() {
    bypassedHandlers.clear();
}

// Virtual methods to be overridden
protected virtual void beforeInsert() {}
protected virtual void beforeUpdate() {}
protected virtual void beforeDelete() {}
protected virtual void afterInsert() {}
protected virtual void afterUpdate() {}
protected virtual void afterDelete() {}
protected virtual void afterUndelete() {}

// Override this to set max loop count (default is 5)
protected virtual Integer getMaxLoopCount() {
    return 5;
}

// Private methods
private void setTriggerContext() {
    this.context = new TriggerContext(
        Trigger.isExecuting,
        Trigger.isInsert,
        Trigger.isUpdate,
        Trigger.isDelete,
        Trigger.isUndelete,
        Trigger.isBefore,
        Trigger.isAfter,
        Trigger.size
    );
}

private void addToLoopCount() {
    String handlerName = getHandlerName();
    if (loopCountMap.containsKey(handlerName)) {
```

```
            Boolean exceeded = loopCountMap.get(handlerName).increment();
            if (exceeded) {
                Integer max = this.getMaxLoopCount();
                throw new TriggerException('Maximum number of trigger loops exceeded: ' +
handlerName + ' (max: ' + max + ')');
            }
        } else {
            loopCountMap.put(handlerName, new LoopCount(this.getMaxLoopCount()));
        }
    }

    private Integer getLoopCount() {
        String handlerName = getHandlerName();
        if (loopCountMap.containsKey(handlerName)) {
            return loopCountMap.get(handlerName).getCount();
        } else {
            return 0;
        }
    }

    private String getHandlerName() {
        return String.valueOf(this).substring(0, String.valueOf(this).indexOf(':'));
    }

    // Inner classes
    private class LoopCount {
        private Integer max;
        private Integer count;

        public LoopCount() {
            this.max = 5;
            this.count = 0;
        }

        public LoopCount(Integer max) {
            this.max = max;
            this.count = 0;
        }

        public Boolean increment() {
            this.count++;
            return this.exceeded();
        }
```

```
        public Boolean exceeded() {
            return this.max >= 0 && this.count > this.max;
        }

        public Integer getCount() {
            return this.count;
        }

        public Integer getMax() {
            return this.max;
        }
    }

    private class TriggerContext {
        public Boolean isExecuting { get; set; }
        public Boolean isInsert { get; set; }
        public Boolean isUpdate { get; set; }
        public Boolean isDelete { get; set; }
        public Boolean isUndelete { get; set; }
        public Boolean isBefore { get; set; }
        public Boolean isAfter { get; set; }
        public Integer size { get; set; }

        public TriggerContext(Boolean isExecuting, Boolean isInsert, Boolean isUpdate,
                              Boolean isDelete, Boolean isUndelete, Boolean isBefore,
                              Boolean isAfter, Integer size) {
            this.isExecuting = isExecuting;
            this.isInsert = isInsert;
            this.isUpdate = isUpdate;
            this.isDelete = isDelete;
            this.isUndelete = isUndelete;
            this.isBefore = isBefore;
            this.isAfter = isAfter;
            this.size = size;
        }
    }

    public class TriggerException extends Exception {}
}

// Example implementation for Account
public class AccountTriggerHandler extends TriggerHandler {

    private List<Account> newAccounts;
```

```
    private List<Account> oldAccounts;
    private Map<Id, Account> newAccountMap;
    private Map<Id, Account> oldAccountMap;

    public AccountTriggerHandler() {
        super();
        this.newAccounts = (List<Account>) Trigger.new;
        this.oldAccounts = (List<Account>) Trigger.old;
        this.newAccountMap = (Map<Id, Account>) Trigger.newMap;
        this.oldAccountMap = (Map<Id, Account>) Trigger.oldMap;
    }

    protected override void beforeInsert() {
        AccountService.validateRequiredFields(this.newAccounts);
        AccountService.setDefaultValues(this.newAccounts);
    }

    protected override void beforeUpdate() {
        AccountService.validateBusinessRules(this.newAccounts, this.oldAccountMap);
    }

    protected override void afterInsert() {
        AccountService.createDefaultContacts(this.newAccounts);
        AccountService.sendWelcomeEmails(this.newAccounts);
    }

    protected override void afterUpdate() {
        AccountService.syncRelatedRecords(this.newAccountMap, this.oldAccountMap);
    }

    protected override Integer getMaxLoopCount() {
        return 3; // Custom max loop count for Account triggers
    }
}

// Trigger implementation
trigger AccountTrigger on Account (before insert, before update, after insert, after update) {
    new AccountTriggerHandler().run();
}
```

Q16: Create a batch job with email notifications and error handling.
Answer:
```apex
```

```
public class AccountUpdateBatch implements Database.Batchable<SObject>,
Database.Stateful {

    private String query;
    private String updateField;
    private Object updateValue;
    private List<String> errorMessages;
    private Integer totalProcessed;
    private Integer totalErrors;

    public AccountUpdateBatch(String updateField, Object updateValue) {
        this.updateField = updateField;
        this.updateValue = updateValue;
        this.errorMessages = new List<String>();
        this.totalProcessed = 0;
        this.totalErrors = 0;

        // Build dynamic query
        this.query = 'SELECT Id, Name, ' + updateField + ' FROM Account WHERE ' + updateField
+ ' = null';
    }

    public Database.QueryLocator start(Database.BatchableContext context) {
        System.debug('Starting AccountUpdateBatch with query: ' + this.query);
        return Database.getQueryLocator(this.query);
    }

    public void execute(Database.BatchableContext context, List<Account> accounts) {
        List<Account> accountsToUpdate = new List<Account>();

        for (Account acc : accounts) {
            acc.put(this.updateField, this.updateValue);
            accountsToUpdate.add(acc);
        }

        // Perform DML with partial success allowed
        Database.SaveResult[] results = Database.update(accountsToUpdate, false);

        // Process results
        for (Integer i = 0; i < results.size(); i++) {
            Database.SaveResult result = results[i];
            Account acc = accountsToUpdate[i];

            if (result.isSuccess()) {
```

```
            this.totalProcessed++;
        } else {
            this.totalErrors++;
            String errorMsg = 'Failed to update Account ' + acc.Name + ' (ID: ' + acc.Id + '): ';

            for (Database.Error error : result.getErrors()) {
                errorMsg += error.getMessage() + '; ';
            }

            this.errorMessages.add(errorMsg);
            System.debug('Error updating account: ' + errorMsg);
        }
    }
}

public void finish(Database.BatchableContext context) {
    System.debug('AccountUpdateBatch completed. Processed: ' + this.totalProcessed + ',
Errors: ' + this.totalErrors);

    // Send completion email
    sendCompletionEmail(context.getJobId());

    // Log batch execution
    logBatchExecution(context);
}

private void sendCompletionEmail(Id jobId) {
    AsyncApexJob job = [
        SELECT Id, Status, NumberOfErrors, JobItemsProcessed, TotalJobItems, CreatedDate,
CompletedDate
        FROM AsyncApexJob
        WHERE Id = :jobId
    ];

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

    // Set email properties
    email.setSubject('Account Update Batch Job Completion - ' + job.Status);
    email.setToAddresses(new List<String>{'admin@company.com'});

    // Build email body
    String emailBody = buildEmailBody(job);
    email.setHtmlBody(emailBody);
```

```
        // Send email
        try {
            Messaging.sendEmail(new List<Messaging.SingleEmailMessage>{email});
        } catch (Exception e) {
            System.debug('Failed to send completion email: ' + e.getMessage());
        }
    }

    private String buildEmailBody(AsyncApexJob job) {
        String body = '<html><body>';
        body += '<h2>Account Update Batch Job Results</h2>';
        body += '<table border="1" style="border-collapse: collapse;">';
        body += '<tr><td><strong>Job ID:</strong></td><td>' + job.Id + '</td></tr>';
        body += '<tr><td><strong>Status:</strong></td><td>' + job.Status + '</td></tr>';
        body += '<tr><td><strong>Started:</strong></td><td>' + job.CreatedDate + '</td></tr>';
        body += '<tr><td><strong>Completed:</strong></td><td>' + job.CompletedDate +
'</td></tr>';
        body += '<tr><td><strong>Total Items:</strong></td><td>' + job.TotalJobItems +
'</td></tr>';
        body += '<tr><td><strong>Processed:</strong></td><td>' + job.JobItemsProcessed +
'</td></tr>';
        body += '<tr><td><strong>Errors:</strong></td><td>' + job.NumberOfErrors + '</td></tr>';
        body += '<tr><td><strong>Update Field:</strong></td><td>' + this.updateField +
'</td></tr>';
        body += '<tr><td><strong>Update Value:</strong></td><td>' + this.updateValue +
'</td></tr>';
        body += '</table>';

        if (!this.errorMessages.isEmpty()) {
            body += '<h3>Error Details:</h3><ul>';
            for (String error : this.errorMessages) {
                body += '<li>' + error + '</li>';
            }
            body += '</ul>';
        }

        body += '</body></html>';
        return body;
    }

    private void logBatchExecution(Database.BatchableContext context) {
        // Create custom log record (assuming you have a Batch_Log__c custom object)
        try {
            Batch_Log__c logRecord = new Batch_Log__c();
```

```
            logRecord.Job_Id__c = String.valueOf(context.getJobId());
            logRecord.Batch_Class__c = 'AccountUpdateBatch';
            logRecord.Total_Processed__c = this.totalProcessed;
            logRecord.Total_Errors__c = this.totalErrors;
            logRecord.Execution_Date__c = System.now();
            logRecord.Status__c = this.totalErrors > 0 ? 'Completed with Errors' : 'Completed
Successfully';

            if (!this.errorMessages.isEmpty()) {
                String errorSummary = String.join(this.errorMessages, '\n');
                logRecord.Error_Details__c = errorSummary.length() > 32000 ?
                                    errorSummary.substring(0, 32000) : errorSummary;
            }

            insert logRecord;

        } catch (Exception e) {
            System.debug('Failed to create batch log record: ' + e.getMessage());
        }
    }
}

// Scheduler class for the batch job
public class AccountUpdateScheduler implements Schedulable {

    private String updateField;
    private Object updateValue;
    private Integer batchSize;

    public AccountUpdateScheduler(String updateField, Object updateValue, Integer batchSize) {
        this.updateField = updateField;
        this.updateValue = updateValue;
        this.batchSize = batchSize != null ? batchSize : 200;
    }

    public void execute(SchedulableContext context) {
        AccountUpdateBatch batch = new AccountUpdateBatch(this.updateField,
this.updateValue);
        Database.executeBatch(batch, this.batchSize);
    }
}

// Usage examples:
// Execute immediately:
```

```
// AccountUpdateBatch batch = new AccountUpdateBatch('Industry', 'Technology');
// Database.executeBatch(batch, 100);

// Schedule for later:
// String cronExp = '0 0 2 * * ?'; // Every day at 2 AM
// AccountUpdateScheduler scheduler = new AccountUpdateScheduler('Industry', 'Technology',
100);
// System.schedule('Account Update Job', cronExp, scheduler);
```

System Design Questions

Q17: Design a solution for handling high-volume data integration with external systems.
Answer:

Architecture Components:

1. **Data Ingestion Layer**
   - Platform Events for real-time data streaming
   - Bulk API 2.0 for large data loads
   - REST APIs for transactional updates

2. **Processing Layer**
   - Queueable jobs for asynchronous processing
   - Batch jobs