I'm new to Salesforce and would like to learn writing triggers for different contexts. i should be able to provide a trigger solution with all the design patterns available. handler class should have proper logging. in the exception handling I want to send email collecting log information

Certainly! I will help you learn Salesforce trigger writing with design patterns, handler classes, proper logging, and exception handling with email notifications. The main contents of the report are as follows:

· Trigger frameworks introduction: Overview of Salesforce trigger frameworks and their benefits.

· Handler pattern implementation: Detailed explanation of the trigger handler pattern with code examples.

· Bulkification techniques: Best practices for handling bulk data operations in triggers.

· Logging framework setup: Implementing a comprehensive logging mechanism for triggers.

· Exception handling: Structured approach to error management with email notifications.

· Advanced patterns: Discussion of virtual class and interface-based trigger frameworks.

· Testing strategies: Guidelines for testing triggers and handler classes.

---

Comprehensive Guide to Salesforce Trigger Development with Advanced Patterns and Exception Handling

1 Introduction to Salesforce Trigger Frameworks

Salesforce trigger frameworks provide structured approaches to organizing trigger logic, preventing the common pitfalls of unstructured code that can lead to performance degradation, recursion issues, and maintenance challenges. Based on the search results, implementing a proper trigger framework offers numerous benefits including generic code that can be extended for any object, consistent trigger handling, simplified triggers and handlers, built-in routing logic, easy trigger bypassing capabilities, and enforced consistent behavior across different trigger contexts . For developers new to Salesforce, understanding these frameworks is crucial for building scalable and maintainable applications.

The search results indicate there are several types of trigger frameworks commonly used in Salesforce development. The most prominent patterns include: Trigger Handler Pattern, Trigger Framework using a Virtual Class, Trigger Framework using an Interface, and Architecture framework to handle triggers . Each approach has its own advantages and implementation considerations, which we will explore throughout this guide. These frameworks address the limitations of writing logic directly in triggers, which can make code difficult to test, maintain, and debug, especially as organizational complexity grows.

2 Understanding the Trigger Handler Pattern

The Trigger Handler Pattern is the most fundamental and widely adopted approach for structuring Salesforce triggers. This pattern emphasizes separation of concerns by moving business logic out of the trigger itself and into a dedicated handler class . The trigger acts only as a gateway, responsible for invoking the appropriate methods in the handler class based on the trigger context (before insert, after update, etc.), while the handler class contains the actual business logic.

According to the search results, the key advantages of using the Trigger Handler pattern include:

· Reusability: Handler methods can be called from other code or tests

· Simplified triggers: Keeps trigger code minimal and focused on routing

· Flexibility: Allows for greater adaptability to changing requirements

· Testability: Unit tests are much easier to write and maintain

· Consistent structure: Enforces standardized approach across all triggers

2.1 Basic Implementation Example

```apex

// AccountTrigger.trigger

trigger AccountTrigger on Account (

before insert, after insert,

before update, after update

) {

AccountTriggerHandler handler = new AccountTriggerHandler(

Trigger.isExecuting,

Trigger.size

);

if(Trigger.isInsert) {

if(Trigger.isBefore) {

handler.OnBeforeInsert(Trigger.new);

} else {

handler.OnAfterInsert(Trigger.new);

}

} else if(Trigger.isUpdate) {

if(Trigger.isBefore) {

handler.OnBeforeUpdate(

Trigger.new,

Trigger.old,

Trigger.newMap,

Trigger.oldMap

);

} else {

handler.OnAfterUpdate(

Trigger.new,

Trigger.old,

Trigger.newMap,

Trigger.oldMap

);

}

}

}

```

```apex

// AccountTriggerHandler.cls

public with sharing class AccountTriggerHandler {

private Boolean m\_isExecuting = false;

private Integer BatchSize = 0;

public static Boolean IsFromBatchJob;

public static Boolean isFromUploadAPI = false;

public AccountTriggerHandler(Boolean isExecuting, Integer size) {

m\_isExecuting = isExecuting;

BatchSize = size;

}

public void OnBeforeInsert(List<Account> newAccounts) {

// Logic for before insert

}

public void OnAfterInsert(List<Account> newAccounts) {

// Logic for after insert

}

public void OnBeforeUpdate(List<Account> newAccounts, List<Account> oldAccounts,

Map<ID, Account> newAccountMap, Map<ID, Account> oldAccountMap) {

// Logic for before update

}

public void OnAfterUpdate(List<Account> newAccounts, List<Account> oldAccounts,

Map<ID, Account> newAccountMap, Map<ID, Account> oldAccountMap) {

// Logic for after update

AccountActions.updateContact(newAccounts);

}

@future

public static void OnAfterUpdateAsync(Set<ID> newAccountIDs) {

// Async logic for after update

}

// Context methods

public Boolean IsTriggerContext {

get { return m\_isExecuting; }

}

public Boolean IsVisualforcePageContext {

get { return !IsTriggerContext; }

}

// Additional context methods...

}

```

3 Bulkification Techniques for Triggers

Bulkification is a critical aspect of Salesforce trigger development that ensures your code can efficiently handle multiple records simultaneously. The Salesforce platform is designed to process records in batches of 200 at a time, and triggers must be written to accommodate this bulk nature . Failure to properly bulkify code can result in performance issues and governor limit exceptions.

3.1 Bulk SOQL and DML Operations

The search results emphasize these key bulkification principles:

· Always operate on record sets: Assume triggers will process multiple records

· Move SOQL queries outside loops: Perform queries once and process results in memory

· Collect records for DML operations: Use lists to accumulate records and perform single DML calls

Table: Bulkification Patterns Comparison

Pattern Inefficient Approach Bulkified Approach

SOQL Query inside FOR loop Query outside loop with WHERE IN clause

DML DML operations inside loop Collect records in list, single DML outside loop

Processing Processing single records Processing collections of records

3.2 Bulkified Code Example

```apex

// Bulkified Trigger Example

trigger BulkifiedAccountTrigger on Account (after update) {

// Efficient SOQL query - gets all related opportunities in one query

List<Opportunity> relatedOpps = [SELECT Id, Name, Probability

FROM Opportunity

WHERE AccountId IN :Trigger.new];

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

// Iterate over related opportunities

for(Opportunity opp : relatedOpps) {

// Update description when probability is between 50% and 100%

if ((opp.Probability >= 50) && (opp.Probability < 100)) {

opp.Description = 'New description for opportunity.';

oppsToUpdate.add(opp);

}

}

// Perform bulk DML operation

if(!oppsToUpdate.isEmpty()) {

update oppsToUpdate;

}

}

```

4 Implementing a Logging Framework

Proper logging is essential for debugging triggers and understanding their behavior in different contexts. A well-implemented logging mechanism captures relevant information during trigger execution while maintaining performance and respecting platform limits.

4.1 Logging Structure Design

```apex

// Logger.cls

public with sharing class Logger {

private static List<Log\_\_c> logs = new List<Log\_\_c>();

private static Boolean isEnabled = true;

public static void debug(String source, String message) {

addLog('DEBUG', source, message, null);

}

public static void info(String source, String message) {

addLog('INFO', source, message, null);

}

public static void error(String source, String message, Exception e) {

String errorDetails = 'Message: ' + e.getMessage() +

'\nStack Trace: ' + e.getStackTraceString() +

'\nLine Number: ' + e.getLineNumber();

addLog('ERROR', source, message, errorDetails);

}

private static void addLog(String level, String source, String message, String details) {

if(!isEnabled) return;

logs.add(new Log\_\_c(

Level\_\_c = level,

Source\_\_c = source,

Message\_\_c = message.left(255),

Details\_\_c = details,

Timestamp\_\_c = System.now()

));

}

public static void flush() {

if(!logs.isEmpty()) {

insert logs;

logs.clear();

}

}

public static void enable() {

isEnabled = true;

}

public static void disable() {

isEnabled = false;

}

}

```

4.2 Integrating Logging with Trigger Handler

```apex

// Updated AccountTriggerHandler with logging

public with sharing class AccountTriggerHandler {

// ... [previous code remains] ...

public void OnAfterUpdate(List<Account> newAccounts, List<Account> oldAccounts,

Map<ID, Account> newAccountMap, Map<ID, Account> oldAccountMap) {

try {

Logger.debug('AccountTriggerHandler.OnAfterUpdate',

'Starting after update processing for ' + newAccounts.size() + ' accounts');

// Business logic here

AccountActions.updateContact(newAccounts);

Logger.info('AccountTriggerHandler.OnAfterUpdate',

'Successfully processed after update for accounts');

} catch(Exception e) {

Logger.error('AccountTriggerHandler.OnAfterUpdate',

'Failed to process after update for accounts', e);

// Exception handling logic

} finally {

Logger.flush();

}

}

// ... [other methods with similar logging] ...

}

```

5 Exception Handling and Email Notification

Robust exception handling is critical for production-grade triggers. The search results indicate that unhandled Apex exceptions automatically generate emails containing stack traces and user/org information, but there are limitations to this mechanism . Specifically, Salesforce suppresses duplicate exception emails for synchronous code and has instance-wide limits of a few hundred exception emails per hour.

5.1 Structured Exception Handling Pattern

```apex

// ExceptionHandler.cls

public with sharing class ExceptionHandler {

public static void handleException(String className, String methodName,

Exception e, List<SObject> records) {

// Create custom exception log

Exception\_Log\_\_c log = new Exception\_Log\_\_c(

Class\_Name\_\_c = className,

Method\_Name\_\_c = methodName,

Error\_Message\_\_c = e.getMessage().left(255),

Stack\_Trace\_\_c = e.getStackTraceString(),

Line\_Number\_\_c = e.getLineNumber(),

Record\_Ids\_\_c = records != null ?

String.join(new List<Id>(new Map<Id, SObject>(records).keySet()), ',') : null

);

insert log;

// Send email notification if not a duplicate exception

if(shouldSendEmail(className, methodName, e)) {

sendEmailNotification(log);

}

}

private static Boolean shouldSendEmail(String className, String methodName, Exception e) {

// Implement logic to check for duplicate exceptions

// Use timestamp tracking or platform cache to avoid email floods

return true;

}

private static void sendEmailNotification(Exception\_Log\_\_c log) {

List<Messaging.SingleEmailMessage> emails = new List<Messaging.SingleEmailMessage>();

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

// Get email addresses from custom setting or metadata

List<String> addresses = getConfiguredEmailAddresses();

email.setToAddresses(addresses);

email.setSubject('Salesforce Trigger Exception: ' + log.Class\_Name\_\_c + '.' + log.Method\_Name\_\_c);

String body = 'An exception occurred in Salesforce trigger processing:\n\n';

body += 'Class: ' + log.Class\_Name\_\_c + '\n';

body += 'Method: ' + log.Method\_Name\_\_c + '\n';

body += 'Error: ' + log.Error\_Message\_\_c + '\n';

body += 'Line: ' + log.Line\_Number\_\_c + '\n';

body += 'Time: ' + System.now() + '\n\n';

body += 'Stack Trace:\n' + log.Stack\_Trace\_\_c;

email.setPlainTextBody(body);

emails.add(email);

// Send email if not in test context

if(!Test.isRunningTest()) {

Messaging.sendEmail(emails);

}

}

private static List<String> getConfiguredEmailAddresses() {

// Retrieve email addresses from custom settings or metadata

return new List<String>{'admin@example.com'};

}

}

```

5.2 Integrating Exception Handling with Trigger Handler

```apex

// Updated AccountTriggerHandler with exception handling

public with sharing class AccountTriggerHandler {

// ... [previous code remains] ...

public void OnAfterUpdate(List<Account> newAccounts, List<Account> oldAccounts,

Map<ID, Account> newAccountMap, Map<ID, Account> oldAccountMap) {

try {

Logger.debug('AccountTriggerHandler.OnAfterUpdate',

'Starting after update processing for ' + newAccounts.size() + ' accounts');

// Business logic here

AccountActions.updateContact(newAccounts);

Logger.info('AccountTriggerHandler.OnAfterUpdate',

'Successfully processed after update for accounts');

} catch(Exception e) {

Logger.error('AccountTriggerHandler.OnAfterUpdate',

'Failed to process after update for accounts', e);

ExceptionHandler.handleException(

'AccountTriggerHandler',

'OnAfterUpdate',

e,

newAccounts

);

} finally {

Logger.flush();

}

}

}

```

6 Advanced Trigger Framework Patterns

Beyond the basic handler pattern, the search results reveal several advanced trigger frameworks that offer additional capabilities and abstraction. These include frameworks using virtual classes, interfaces, and more architectural approaches .

6.1 Virtual Class Framework Pattern

The virtual class framework utilizes inheritance to create a base trigger handler class that can be extended for specific objects. This approach provides built-in routing and context-specific methods that are automatically called during trigger execution .

```apex

// Virtual base handler class

public virtual class TriggerHandler {

// Context methods with virtual implementations

public virtual void beforeInsert(List<SObject> newRecords) {}

public virtual void afterInsert(List<SObject> newRecords) {}

public virtual void beforeUpdate(List<SObject> newRecords, List<SObject> oldRecords,

Map<ID, SObject> newMap, Map<ID, SObject> oldMap) {}

public virtual void afterUpdate(List<SObject> newRecords, List<SObject> oldRecords,

Map<ID, SObject> newMap, Map<ID, SObject> oldMap) {}

// Main execute method to handle routing

public void execute() {

if(Trigger.isBefore) {

if(Trigger.isInsert) beforeInsert(Trigger.new);

else if(Trigger.isUpdate) beforeUpdate(Trigger.new, Trigger.old, Trigger.newMap, Trigger.oldMap);

} else {

if(Trigger.isInsert) afterInsert(Trigger.new);

else if(Trigger.isUpdate) afterUpdate(Trigger.new, Trigger.old, Trigger.newMap, Trigger.oldMap);

}

}

}

```

6.2 Interface-Based Framework Pattern

The interface-based framework enforces consistency across all trigger handlers by requiring implementation of specific methods. This approach ensures all handlers have the same structure and provides multiple ways to deactivate triggers .

```apex

// Trigger handler interface

public interface ITriggerHandler {

void beforeInsert(List<SObject> newRecords);

void afterInsert(List<SObject> newRecords);

void beforeUpdate(List<SObject> newRecords, List<SObject> oldRecords,

Map<ID, SObject> newMap, Map<ID, SObject> oldMap);

void afterUpdate(List<SObject> newRecords, List<SObject> oldRecords,

Map<ID, SObject> newMap, Map<ID, SObject> oldMap);

Boolean isEnabled();

void enable();

void disable();

}

// Account trigger handler implementing interface

public class AccountTriggerHandler implements ITriggerHandler {

private Boolean enabled = true;

public Boolean isEnabled() {

return enabled;

}

public void enable() {

enabled = true;

}

public void disable() {

enabled = false;

}

public void beforeInsert(List<SObject> newRecords) {

if(!enabled) return;

List<Account> newAccounts = (List<Account>)newRecords;

// Implementation logic

}

// Other interface methods with similar structure

}

```

Table: Comparison of Advanced Trigger Frameworks

Framework Type Key Features Best For

Virtual Class Built-in routing, context methods, inheritance-based Organizations preferring convention over configuration

Interface-Based Enforcement of consistent methods, multiple deactivation options Teams needing strict consistency across handlers

Architectural Event-based handling, individual method toggling Large enterprises with complex trigger requirements

7 Testing Strategies for Triggers and Handlers

Compressive testing is essential for ensuring trigger reliability. The search results emphasize that well-structured handlers make unit tests easier to write and maintain . Proper testing should cover positive and negative scenarios, bulk operations, and different trigger contexts.

7.1 Unit Test Structure for Trigger Handlers

```apex

@isTest

private class AccountTriggerHandlerTest {

@testSetup

static void setup() {

// Create test data

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

for(Integer i = 0; i < 200; i++) {

testAccounts.add(new Account(

Name = 'Test Account ' + i,

Industry = 'Technology'

));

}

insert testAccounts;

}

@isTest

static void testBeforeInsert() {

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

for(Integer i = 0; i < 5; i++) {

newAccounts.add(new Account(

Name = 'Test Account ' + i,

Industry = null

));

}

Test.startTest();

AccountTriggerHandler handler = new AccountTriggerHandler(true, 5);

handler.OnBeforeInsert(newAccounts);

Test.stopTest();

// Verify results

for(Account acc : newAccounts) {

System.assertEquals('Other', acc.Industry,

'Default industry should be set when null');

}

}

@isTest

static void testAfterUpdateBulk() {

List<Account> existingAccounts = [SELECT Id, Name FROM Account LIMIT 200];

for(Account acc : existingAccounts) {

acc.Name = 'Updated ' + acc.Name;

}

Test.startTest();

AccountTriggerHandler handler = new AccountTriggerHandler(true, 200);

handler.OnAfterUpdate(

existingAccounts,

existingAccounts,

new Map<ID, Account>(existingAccounts),

new Map<ID, Account>(existingAccounts)

);

Test.stopTest();

// Verify bulk processing occurred without errors

System.assertEquals(200, existingAccounts.size(),

'All accounts should be processed in bulk');

}

@isTest

static void testExceptionHandling() {

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

// Create accounts that will cause exception

Test.startTest();

AccountTriggerHandler handler = new AccountTriggerHandler(true, 1);

try {

handler.OnAfterUpdate(

newAccounts,

newAccounts,

new Map<ID, Account>(),

new Map<ID, Account>()

);

System.assert(false, 'Expected exception was not thrown');

} catch(Exception e) {

// Verify exception was handled appropriately

System.assertNotEquals(null, e, 'Exception should be caught');

}

Test.stopTest();

}

}

```

8 Conclusion and Best Practices Summary

Implementing well-architected triggers with proper patterns, logging, and exception handling is essential for building scalable and maintainable Salesforce applications. Based on the search results, the key best practices include:

· One Trigger Per Object: Maintain a single trigger per object to control execution order

· Logic-less Triggers: Keep triggers minimal with only routing logic

· Context-Specific Handler Methods: Implement separate methods for each trigger context

· Bulkification: Always write code to handle multiple records efficiently

· Avoid SOQL/DML in Loops: Move queries and data operations outside loops

· Use Collections: Utilize maps and sets for efficient data processing

· Proper Exception Handling: Implement structured error handling with logging

· Comprehensive Testing: Create tests covering all scenarios including bulk operations

8.1 Implementation Roadmap

For developers new to Salesforce triggers, follow this implementation roadmap:

1. Start with the basic handler pattern for simplicity and understanding

2. Implement bulkification in all handler methods

3. Add logging to capture debug information and execution flow

4. Integrate exception handling with email notifications

5. Consider advanced frameworks as organizational complexity grows

6. Develop comprehensive tests for all trigger scenarios

By following these patterns and practices, you'll create triggers that are performant, maintainable, and resilient to errors, while providing appropriate visibility through logging and notifications. Remember that the choice of framework should be based on your organization's size and specific needs, as all approaches have their advantages and trade-offs .