Available online www.jsaer.com

***Journal of Scientific and Engineering Research*, 2020, 7(1):296-300**

Research Article

ISSN: 2394-2630
CODEN(USA): JSERBR

Implementing a Trigger Framework in Salesforce

Chirag Amrutlal Pethad

PetSmart.com, LLC, Stores and Services
Phoenix, Arizona, USA
ChiragPethad@live.com, ChiragPethad@gmail.com, Cpethad@petsmart.com

Abstract: This document provides a guide to implementing a robust Trigger Framework in Salesforce Apex. It explains the importance of triggers, their execution context, and the need for a trigger framework. The document outlines the core components of a trigger framework and discusses the benefits of using one, such as reusability, maintainability, and scalability. It also provides an example implementation and best practices for trigger development in Salesforce.

Keywords: Execution Context, Context Variables, Bulk, Modularity, Scalability, Governor Limits, Testability

Introduction

Salesforce Apex Triggers are essential tools for automating business processes and ensuring data integrity. Triggers are powerful mechanisms that enable developers to execute custom logic before or after records are inserted, updated, deleted, or undeleted in the database. They are vital components in customizing and extending the functionality of Salesforce applications. However, as projects grow in complexity and scale, managing triggers efficiently becomes essential. This white paper presents a structured approach to handling triggers through a Trigger Framework, improving code maintainability, readability, and scalability.

Overview Of Salesforce Triggers

Apex Triggers are server-side scripts that execute automatically based on specific events in Salesforce, such as inserting, updating, or deleting records. Triggers can operate before or after the record is saved to the database. They provide way for applications perform additional actions, validations, or business logic in response to these database operations.

A. Trigger Execution Context

• **Before Triggers:** These triggers execute before the records are committed to the database. They are commonly used for validation purposes or to modify record values before they are saved.

• **After Triggers:** These triggers execute after the records are committed to the database. They are often used for tasks such as sending notifications, updating related records, or invoking external services.

• **Bulk Triggers:** Triggers in Salesforce are designed to handle bulk operations, meaning they should be written to process multiple records efficiently. Bulk triggers are essential for ensuring performance and scalability of your application.

B. Trigger Events and Context Variables

Salesforce provides trigger context variables that allow code logic to access information about the records being processed and the operation being performed. Some of the commonly used trigger context variables include:

- **Trigger.new**: Contains the list of new records being inserted or updated.
- **Trigger.old:** Contains the list of old records before they were updated or deleted.

- **Trigger.newMap:** A map of IDs to the new versions of the records being processed.
- **Trigger.oldMap:** A map of IDs to the old versions of the records being processed.
- **Trigger.isInsert, Trigger.isUpdate, Trigger.isDelete, Trigger.isUndelete:** Boolean variables indicating the type of DML operation being performed.

What Is a Framework

A framework is a highly optimized, reusable structure that serves as a building block. These building blocks provide common functionality that developers can override or specialize for their own needs. Reusable frameworks increase the speed of development, improve the clarity and efficiency of your code, and simplify code reviews and debugging.

Need For a Trigger Framework

As the complexity of a Salesforce org increases, so does the need for a robust mechanism to manage triggers efficiently. The primary goal of this framework is to avoid common pitfalls such as recursion, governor limit exceptions, and unmanageable code by providing a systematic way to structure trigger logic. Some challenges without a framework include:

- **Code Duplication:** Similar logic across multiple triggers.
- **Unorganized Code:** Hard to maintain and read.
- **Performance Issues:** Unoptimized trigger execution.
- **Lack of Reusability:** Difficulty in reusing common logic.

A Trigger Framework addresses these issues by providing a structured approach to trigger management.

Core Components of a Trigger Framework

A Trigger Framework typically consists of the following components:

- **Trigger Handler Class**: Contains the business logic for different trigger events.
- **Helper Classes:** Encapsulate reusable logic.
- **Utility Classes**: Provide common functionalities.
- **Custom Settings/Metadata:** Control trigger execution dynamically.

Benefits of Using a Trigger Framework

Implementing a Trigger Framework offers several advantages:

- **Reusability:** Trigger framework allows to implement shared logic that can be reused across multiple triggers.
- **Maintainability**: Trigger frameworks enable to maintain cleaner and more organized code.
- **Control:** Trigger frameworks provide ability to manage trigger execution order and context.
- **Modularity:** Trigger frameworks promote modular design by separating concerns and encapsulating trigger logic within handler classes.
- **Scalability:** Trigger frameworks enable to easily add, remove, or modify trigger logic without affecting existing functionality.
- **Testability**: Trigger logic encapsulated within handler classes can be easily unit tested, leading to improved code quality and reliability.
- **Governor Limits Management:** Trigger framework help in managing Salesforce governor limits more efficiently by optimizing the execution of trigger logic.
- **Version Control and Change Management:** Trigger framework encourage version control and change management procedures. This is especially significant in larger development teams or projects when several developers are working on the same codebase.
- **Collaboration:** Using a shared framework generates coding norms and patterns that facilitate developer collaboration. It encourages consistent code style and structure across the development team.

Implementing a Trigger Framework

A. Context Variables

We start by creating an Apex class to automatically populate the Trigger Context variables. This is much cleaner than having methods with 5 to 10 parameters while still giving full access to all standard functionality.

```
public class TriggerContext {
    private System.TriggerOperation operation {
        get; set{ operation = System.Trigger.operationType;}
    }
    private List<SObject> triggerNew {
        get; set{ triggerNew = System.Trigger.new;}
    }
    private List<SObject> triggerOld {
      get; set{ triggerOld = System.Trigger.old;}
    }
    private Map<Id, SObject> newMap {
      get; set{ newMap = System.Trigger.newMap;}
    }
    private Map<Id, SObject> oldMap {
        get; set{ oldMap = System.Trigger.oldMap;}
    }
}
```

B. Interface to declare Trigger Handler Blueprint

The Trigger Handler Pattern involves creating a base handler class and extending it for specific objects. The base class provides the default implementation for all the different event (Before / After, Insert/Update/Delete and Undelete), while the derived classes implement the actual business logic only for what the trigger is responsible for.

Implement this interface to ensure consistency across different handler classes.

```
public interface ITrigger {
  void beforeInsert(List<SObject> newItems);
  void beforeUpdate(Map<Id, SObject> oldItems, Map<Id, SObject> newItems);
  void beforeDelete(Map<Id, SObject> oldItems);
  void afterInsert(Map<Id, SObject> newItems);
  void afterUpdate(Map<Id, SObject> oldItems, Map<Id, SObject> newItems);
  void afterDelete(Map<Id, SObject> oldItems);
  void afterUndelete(Map<Id, SObject> oldItems);
}
```

C. Base Implementation of the Trigger Handler Interface

Base Class provides the default implementation for the Trigger Interface so that all the child handlers do not need to implement the functionality that is not required. Developers can focus only on the business functionality.

```
public virtual class TriggerHandler {
  public virtual void beforeInsert(List<SObject> newItems) {}
  public virtual void beforeUpdate(
      Map<Id, SObject> oldItems, Map<Id, SObject> newItems) {}
    public virtual void beforeDelete(Map<Id, SObject> oldItems) {}
  public virtual void afterInsert(Map<Id, SObject> newItems) {}
  public virtual void afterInsert(Map<String, SObject> newItems) {}
  public virtual void afterUpdate(
      Map<Id, SObject> oldItems, Map<Id, SObject> newItems) {}
  public virtual void afterDelete(Map<Id, SObject> oldItems) {}
  public virtual void afterUndelete(Map<Id, SObject> oldItems) {}
}
```

D. Tigger Handler Metadata

Create a Helper Class that will retrieve the SObject trigger metadata which includes all the handlers associated with the SObject, Order of Execution, etc.

```
public class TriggerSettings {
  private List<ITrigger> handlers = new List<ITrigger>();
  public TriggerSettings(Schema.SObjectType objType) {
      //Write logic to get the handler Names from Custom Metadata
      // or Custom Settings. And then loop over the list
      // and Instantiate instances of those Trigger Handlers for SObjectType
      //for (TriggerHandlerMetadata_mdt triggerHandler : triggerHandlers) {
      //      Type t = Type.forName(handlerSetting.DeveloperName);
      //    this.handlers.add((ITrigger)t.newInstance());
      //}
  }
  public List<ITrigger> getHandlers(){ return handlers; }
}
```

E. Tigger Dispatcher / Controller

Next, we implement a dispatcher class that will orchestrate all the execution of various handlers and logic associated with an SObject.

```
public class TriggerDispatcher {
  public static void dispatch(Schema.SObjectType objType) {
    // Set the TriggerContext
    TriggerContext  ctx = new TriggerContext();
    // Get the Handlers Lis
    List<ITrigger> handlers = new TriggerSettings(objType).getHandlers();
    for (ITrigger handler : handlers) {
      switch on ctx.getOperation() {
        when BEFORE_INSERT {
          handler.beforeInsert(ctx.getTriggerNew());
        } when BEFORE_UPDATE {
          handler.beforeUpdate(ctx.getOldMap(), ctx.getNewMap());
        } when BEFORE_DELETE {
          handler.beforeDelete(ctx.getOldMap());
        } when AFTER_INSERT {
          handler.afterInsert(ctx.getNewMap());
        } when AFTER_UPDATE {
          handler.afterUpdate(ctx.getOldMap(), ctx.getNewMap());
        } when AFTER_DELETE {
          handler.afterDelete(ctx.getOldMap());
        } when AFTER_UNDELETE {
          handler.afterUndelete(ctx.getOldMap());
        }
      }
    }
  }
}
```

Example Implementation

A. Trigger

Time and effort in using the framework reduces the time and effort of the developers so that they can focus on the business logic.

```
trigger ContactTrigger on Contact (before insert, before update,
                                   before delete, after insert,
                                   after update, after delete,
                                   after undelete) {
    TriggerDispatcher.dispatch(Contact.sObjectType);
}
```

B. Handler

Developers just need to override and implement the business logic for the one or more of the event types and skip the ones that are not required.

```
public class ContactTriggerHandler extends TriggerHandler {
    override public void beforeInsert() {
        // Insert logic here
    }
    override public void beforeUpdate() {
        // Update logic here
    }
    override public void beforeDelete() {
        // Delete logic here
    }
    override public void afterInsert() {
        // After insert logic here
    }
    override public void afterUpdate() {
        // After update logic here
    }
    override public void afterDelete() {
        // After delete logic here
    }
    override public void afterUndelete() {
        // After undelete logic here
    }
}
```

Best Practices

- **Bulkification:** Ensure all trigger operations handle bulk data efficiently.
- **Single Trigger Per Object**: Consolidate logic into a single trigger per object.

- **Limit DML Statements:** Minimize the number of DML statements and queries within triggers.
- **Error Handling**: Implement robust error handling and logging mechanisms.

Conclusion

Triggers are powerful tools in Salesforce development, but they require careful management to ensure scalability, maintainability, and performance. Implementing a Trigger Framework in Salesforce Apex significantly enhances code quality and maintainability. By organizing and structuring trigger logic, developers can ensure scalable and efficient trigger management, ultimately leading to a more robust Salesforce environment.

References

[1]. Salesforce Apex Developer Guide - https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_dev_guide.htm
[2]. Salesforce Trigger Best Practices- https://trailhead.salesforce.com/content/learn/modules/apex_triggers/apex_triggers_intro.
[3]. Fflib-apex-common - https://github.com/apex-enterprise-patterns/fflib-apex-common
[4]. https://github.com/kevinohara80/sfdc-trigger-framework