Phase 5: Apex Programming (Developer)

CONNECT Student Success Platform - Salesforce CRM Implementation

5.1 Classes & Objects

Use Case

Apex class to handle Student Intervention utilities, such as updating the intervention date when the status changes to "Reviewed".

Business Requirement

Ensure that when any Student Intervention record status is updated to "Reviewed", its Intervention Date field gets set to today automatically for audit and workflow tracking.

Implementation Steps

1. Go to Setup → Quick Find → Apex Classes.
2. Click "New" to create a new Apex Class.
3. In the editor, paste the following code:

Code:

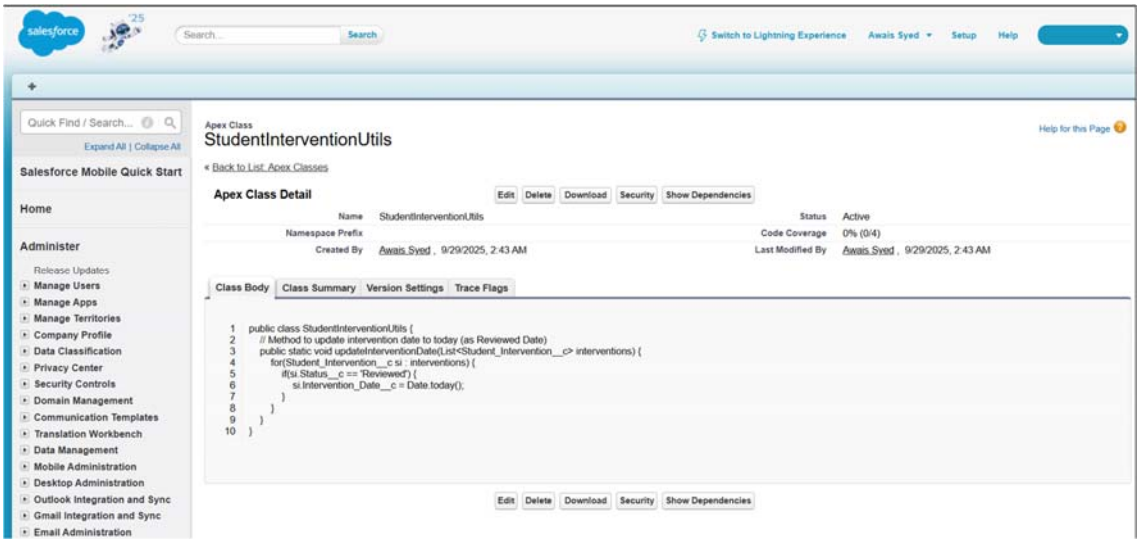
```
public class StudentInterventionUtils {
   public static void updateInterventionDate(List<Student_Intervention__c> interventions) {
      for(Student_Intervention__c si : interventions) {
         if(si.Status__c == 'Reviewed') {
            si.Intervention_Date__c = Date.today();
         }
      }
   }
}
```

4. Save.

Result

The class StudentInterventionUtils with a method updateInterventionDate now exists. This method helps automate setting the intervention date when a record is reviewed. It can be reused in triggers and other Apex logic.

Screenshot



COMMENT: Screenshot of Apex Class screen. Setup → Apex Classes → Student Intervention

5.2 Apex Triggers

Use Case: Automatically update the Intervention Date of a Student Intervention record when its status is set to "Reviewed", without requiring manual user action.

Business Requirement: Every time a Student Intervention is either created or updated, the Intervention Date should automatically be set to today's date if the status is "Reviewed". This ensures accurate tracking and minimal manual work for users.

Implementation Steps

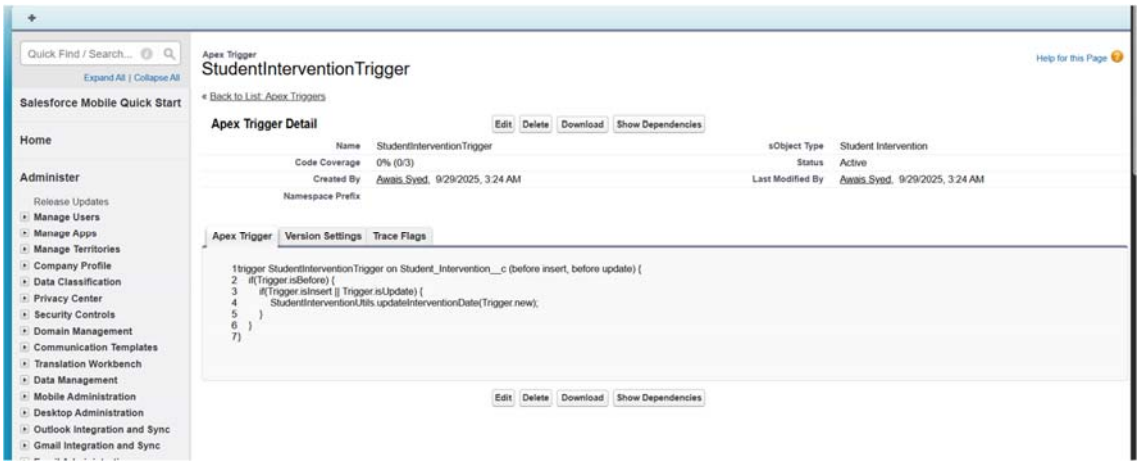
1. Go to Setup → Object Manager → Student Intervention.
2. Click on "Triggers" in the left sidebar.
3. Click "New" to create a new trigger.
4. Enter the trigger name as StudentInterventionTrigger.
5. Paste the following code:

Code

```
trigger StudentInterventionTrigger on Student_Intervention__c (before insert, before update) {
    if(Trigger.isBefore) {
        if(Trigger.isInsert || Trigger.isUpdate) {
            StudentInterventionUtils.updateInterventionDate(Trigger.new);
        }
    }
}
```

6. Save.

Result: This trigger will automatically update the Intervention Date to today's date every time a Student Intervention is inserted or updated and the status is set to "Reviewed". The trigger calls the utility method created in Phase 5.1, keeping our code modular and maintainable.



COMMENT: Screenshot of trigger code/details page. Setup → Object Manager → Student Intervention → Triggers → StudentInterventionTrigger.

5.3 Trigger Design Pattern Documentation

Use Case: To organize Apex trigger logic cleanly by separating event routing from business logic, improving maintainability and reusability in the CONNECT Student Intervention project.

Business Requirement: Implement a scalable trigger architecture where the Student Intervention trigger delegates processing to a handler class. This enables easy future extension and keeps triggers simple.

Implementation Steps

1. Create a new Apex class StudentInterventionTriggerHandler with methods for handling trigger events (beforeInsert, beforeUpdate).
2. Move business logic (e.g., invocation of utility method) from the trigger to these handler methods.
3. Modify the existing StudentInterventionTrigger to create an instance of the handler and delegate trigger context events to appropriate handler methods.

Code

Trigger:

```
trigger StudentInterventionTrigger on Student_Intervention__c (before insert, before update) {
```

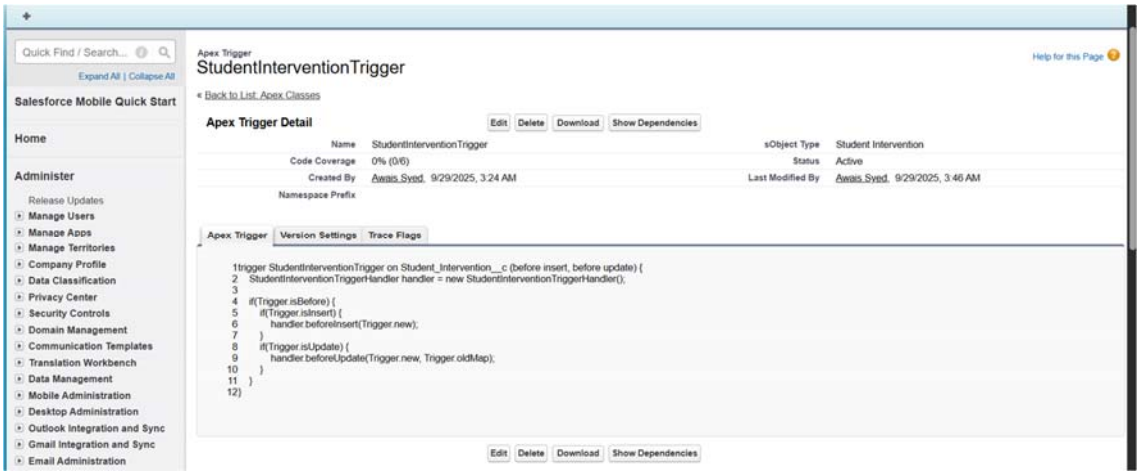
```
    StudentInterventionTriggerHandler handler = new StudentInterventionTriggerHandler();
    if(Trigger.isBefore) {
        if(Trigger.isInsert) {
            handler.beforeInsert(Trigger.new);
        }
        if(Trigger.isUpdate) {
            handler.beforeUpdate(Trigger.new, Trigger.oldMap);
        }
    }
}
```

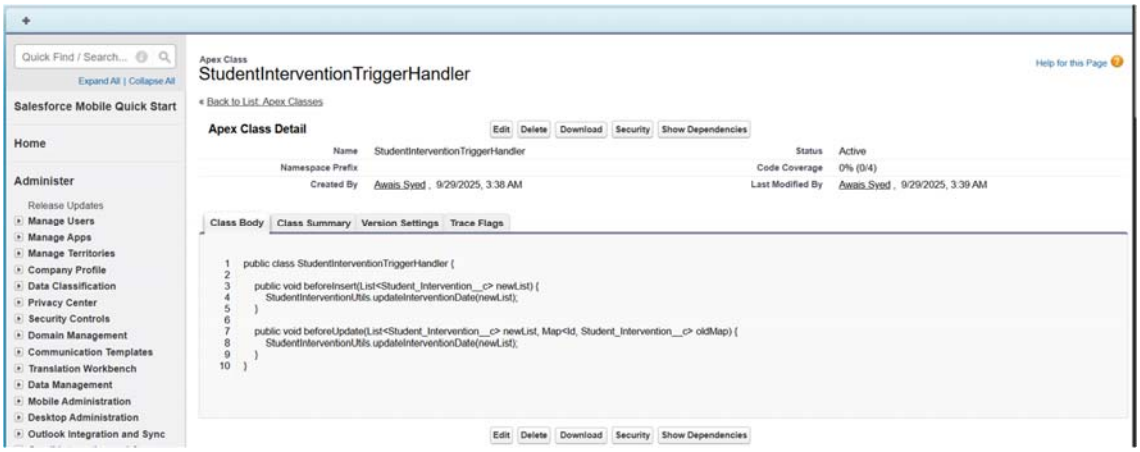
Handler Class:

```
public class StudentInterventionTriggerHandler {
    public void beforeInsert(List<Student_Intervention__c> newList) {
        StudentInterventionUtils.updateInterventionDate(newList);
    }

Public void beforeUpdate(List<Student_Intervention__c> newList, Map<Id, Student_Interve
ntion__c> oldMap) {
        StudentInterventionUtils.updateInterventionDate(newList);
    }
}
```

Result: The trigger is slim and manages only context/event routing. The handler class centralizes business logic for handling Student Intervention insert and update events, resulting in maintainable and reusable code.





COMMENT: This pattern can be expanded with more methods for other trigger events like after insert, after update, before delete, and after delete as project complexity grows.

5.4 SOQL & SOSL (Updated Documentation)

Use Case: Retrieve records related to students, interventions, or advisors dynamically from Salesforce using SOQL and SOSL queries for use in Apex code.

Business Requirement: Showcase basic querying capabilities to filter and search records based on criteria using Apex methods.

Implementation in StudentInterventionUtils Class

SOQL (Salesforce Object Query Language):

- Used to fetch records based on precise conditions.
- Only queries Salesforce objects, returns records in a List.

Code:

```
public static List<Student_Intervention__c> getRecentReviewedInterventions() {
    return [SELECT Id, Name, Status__c, Intervention_Date__c
```

```
        FROM Student_Intervention__c
        WHERE Status__c = 'Reviewed'
        ORDER BY Intervention_Date__c DESC
        LIMIT 5];
}
```

- This method returns the latest 5 reviewed interventions.

SOSL (Salesforce Object Search Language):

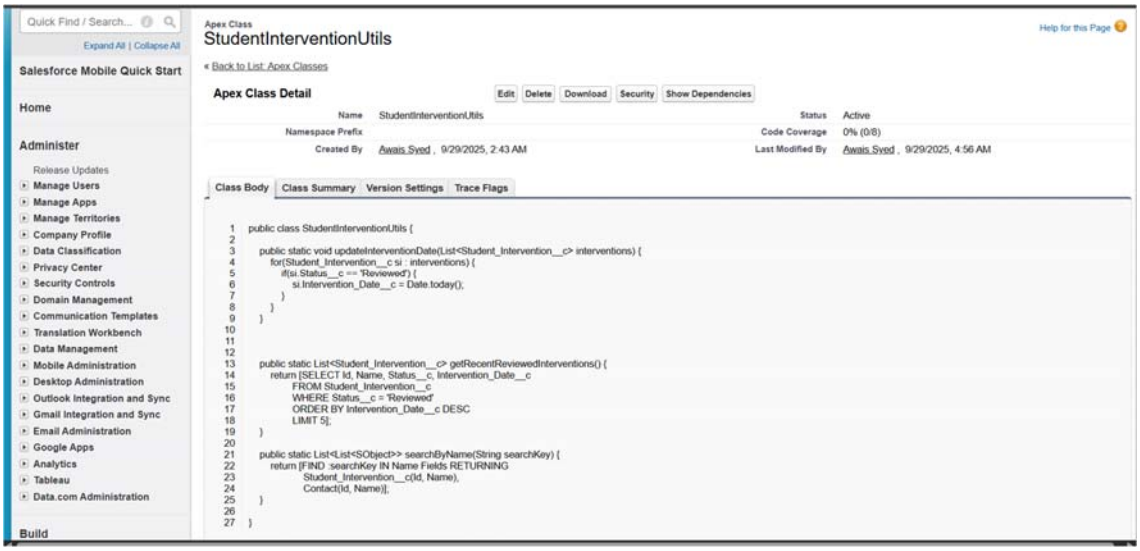
- Used for text searches across multiple objects and fields.
- Returns results grouped by object type.

Code:

```
public static List<List<SObject>> searchByName(String searchKey) {
   return [FIND :searchKey IN Name Fields RETURNING
            Student_Intervention__c(Id, Name),
            Contact(Id, Name)];
}
```

- This method searches for interventions and contacts by a partial name.

Result: Ability to programmatically retrieve and search records within custom Apex methods, making your logic dynamic and data-driven.



5.5 Collections: List, Set, Map

Use Case: Handle groups of records or data efficiently in Apex using Collections like Lists, Sets, and Maps. These allow storing, retrieving, and manipulating multiple data items in bulk operations.

Business Requirement: Use collections to improve data processing efficiency and support bulk operations for student intervention data in Salesforce Apex, ensuring performant and manageable code.

Implementation in StudentInterventionUtils Class

List: Ordered collection, allows duplicates

- Example method to extract names of interventions from a list of intervention records:

Code

```
Public static List<String> getInterventionNames(List<Student_Intervention__c> intervention
s) {
   List<String> names = new List<String>();
   for(Student_Intervention__c si : interventions) {
      names.add(si.Name);
   }
   return names;
}
```

Set: Unordered collection of unique elements, no duplicates

- Example method to get unique statuses from interventions, ensuring no duplication:

Code

```
public static Set<String> getUniqueStatuses(List<Student_Intervention__c> interventions) {
   Set<String> statuses = new Set<String>();
   for(Student_Intervention__c si : interventions) {
      statuses.add(si.Status__c);
   }
   return statuses;
}
```

Map: Key-value pairs for fast data lookup

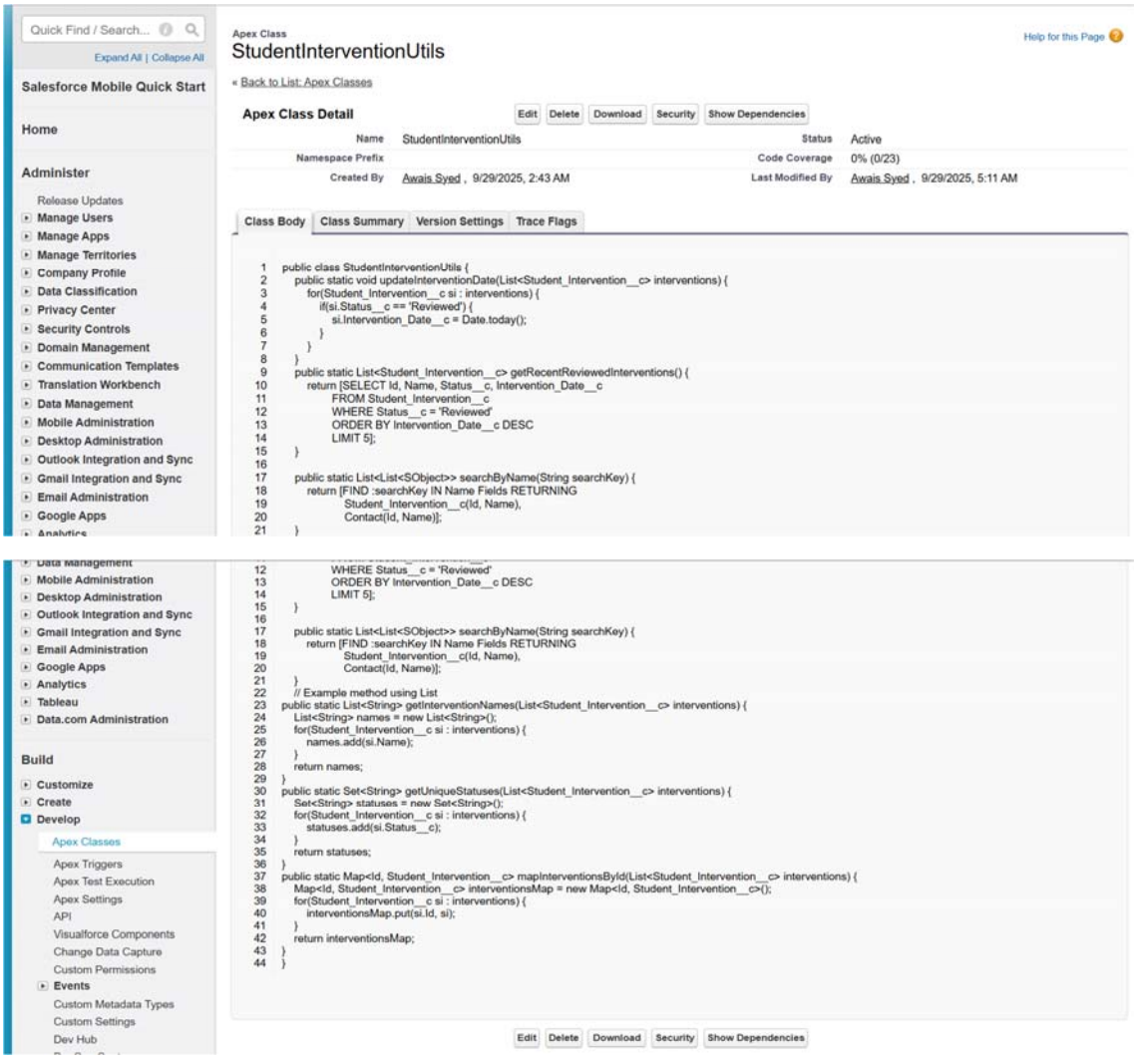
- Example method to create a map from Intervention Id to the record itself:

Code:

```
Public static Map<Id, Student_Intervention__c> mapInterventionsById(List<Student_Intervention__c> interventions) {


Map<Id, Student_Intervention__c> interventionsMap = new Map<Id,  Student_Intervention__c>();
   for(Student_Intervention__c si : interventions) {
      interventionsMap.put(si.Id, si);
   }
   return interventionsMap;
}
```

Result: Using collections efficiently supports bulk data processing, enhances code readability, and ensures faster lookup, manipulation, and retrieval of related records in your Apex logic.



5.6 Control Statements

Use Case: Control statements allow your Apex code to make decisions and repeat actions based on conditions. They guide the flow of execution in your program.

Business Requirement: Use conditional statements and loops to control logic in Salesforce Apex for efficient data manipulation and decision-making in your project.

Implementation

Code

```
public static void updateReviewedInterventions(List<Student_Intervention__c> interventions)
{
   for(Student_Intervention__c si : interventions) {
      if(si.Status__c == 'Reviewed') {
         si.Intervention_Date__c = Date.today();
      } else {
         si.Intervention_Date__c = null;
      }
   }
}
```

Result: This method uses if-else control statements within a for loop to update intervention date only if status is "Reviewed".

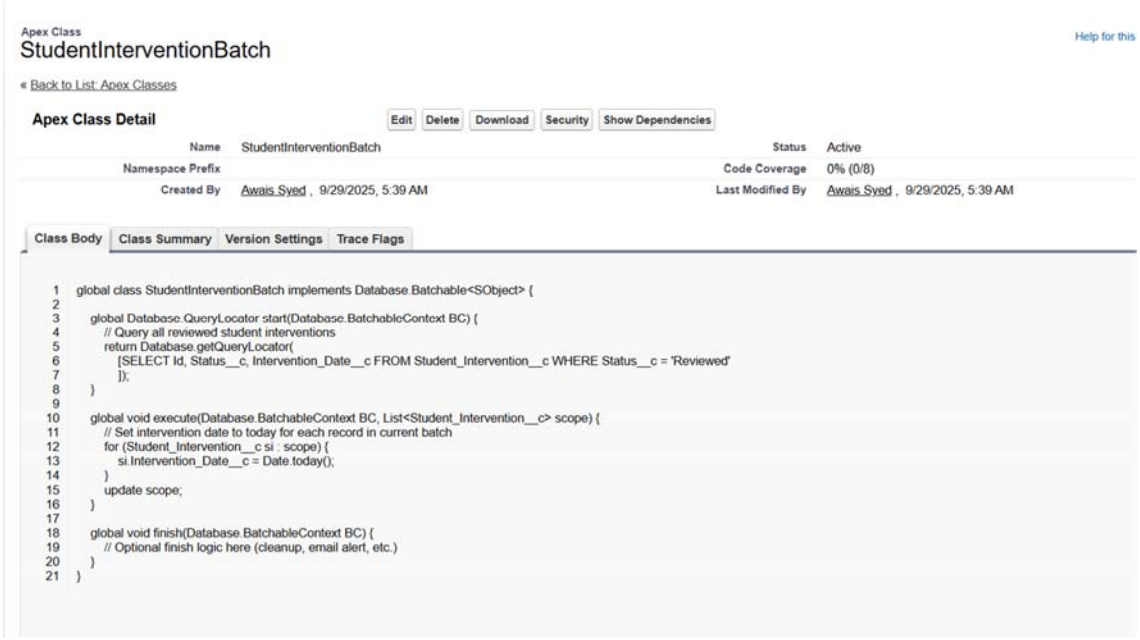
```
17      public static List<List<SObject>> searchByName(String searchKey) {
18          return [FIND :searchKey IN Name Fields RETURNING
19                      Student_Intervention__c(Id, Name),
20                      Contact(Id, Name)];
21      }
22      // Example method using List
23  public static List<String> getInterventionNames(List<Student_Intervention__c> interventions) {
24      List<String> names = new List<String>();
25      for(Student_Intervention__c si : interventions) {
26          names.add(si.Name);
27      }
28      return names;
29  }
30  public static Set<String> getUniqueStatuses(List<Student_Intervention__c> interventions) {
31      Set<String> statuses = new Set<String>();
32      for(Student_Intervention__c si : interventions) {
33          statuses.add(si.Status__c);
34      }
35      return statuses;
36  }
37  public static Map<Id, Student_Intervention__c> mapInterventionsById(List<Student_Intervention__c> interventions) {
38      Map<Id, Student_Intervention__c> interventionsMap = new Map<Id, Student_Intervention__c>();
39      for(Student_Intervention__c si : interventions) {
40          interventionsMap.put(si.Id, si);
41      }
42      return interventionsMap;
43  }
44  // Example method demonstrating if-else and loop control statements
45  public static void updateReviewedInterventions(List<Student_Intervention__c> interventions) {
46      for(Student_Intervention__c si : interventions) {
47          if(si.Status__c == 'Reviewed') {
48              si.Intervention_Date__c = Date.today();
49          } else {
50              si.Intervention_Date__c = null;
51          }
52      }
53  }
54  }
```

5.7 Batch Apex

Use Case: Perform complex, long-running, or bulk operations on large sets of records asynchronously, efficiently and within Salesforce governor limits.

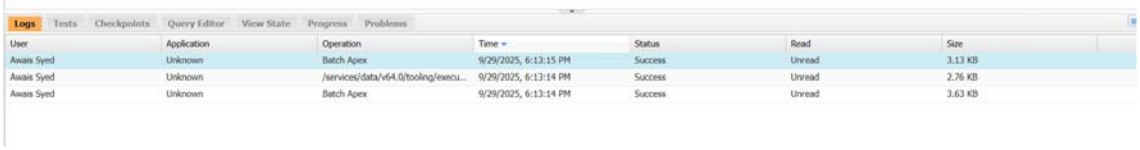
Business Requirement: Implement Batch Apex to handle tasks such as bulk data updates, data cleansing, or archiving for objects like Student Interventions when data volume is high.

Implementation Batch Apex Class



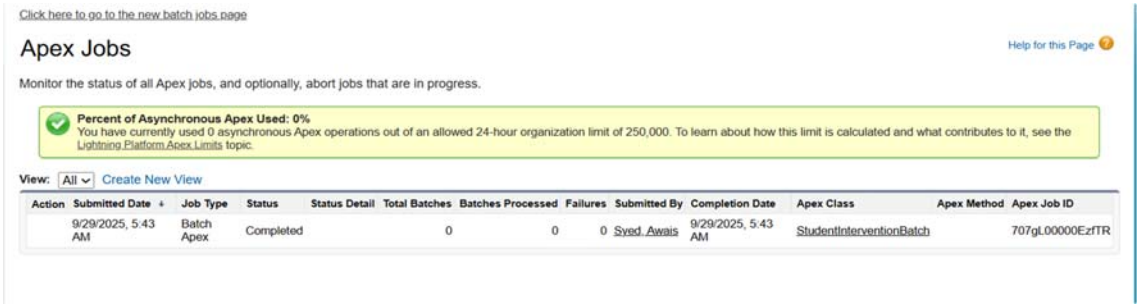
Run the Batch Job

- Go to Developer Console → Execute Anonymous window.
- Run: Database.executeBatch(new StudentInterventionBatch(), 200);



Monitor the Batch Job

- **Go to Setup → Apex Jobs**



Result: Batch Apex allows efficient, manageable processing of large datasets asynchronously, avoiding resource limits and providing robust data operations for your Salesforce project.

5.8 Queueable Apex

Use Case: Process complex asynchronous operations with more flexible job chaining than Future Methods. Queueable Apex allows for chaining jobs and offers more control over asynchronous processing.

Business Requirement: Use Queueable Apex to handle asynchronous tasks like callouts, chaining large jobs, or deferring processing without blocking the user interface.

Key Points

- Implements Queueable interface.
- Supports complex job chaining.
- Better than Future Methods with enhanced features.

5.9 Scheduled Apex

Use Case: Run Apex code at scheduled times, such as nightly batch jobs or periodic data cleanup tasks.

Business Requirement: Schedule regular automated jobs in Salesforce, such as daily student data updates, reminders, or reports.

Key Points

- Implements Schedulable interface.
- Use cron expressions to specify schedule.
- Can call batch or queueable jobs within scheduled jobs.

5.10 Future Methods

Use Case: Execute long-running operations asynchronously to improve user experience during transactions, such as making callouts.

Business Requirement: Handle deferred processing without slowing down immediate user actions, run fire-and-forget jobs.

Key Points

- Annotated with @future.
- No chaining support.
- Good for simple async tasks like callouts.

5.11 Exception Handling

Use Case: Manage errors and exceptional conditions gracefully in Apex code to avoid unhandled failures and log for traceability.

Business Requirement: Implement robust error handling to improve reliability and maintainability in Salesforce projects.

Key Points

- Use try, catch, finally blocks.
- Use custom exception classes where needed.
- Log exceptions or notify admins appropriately.

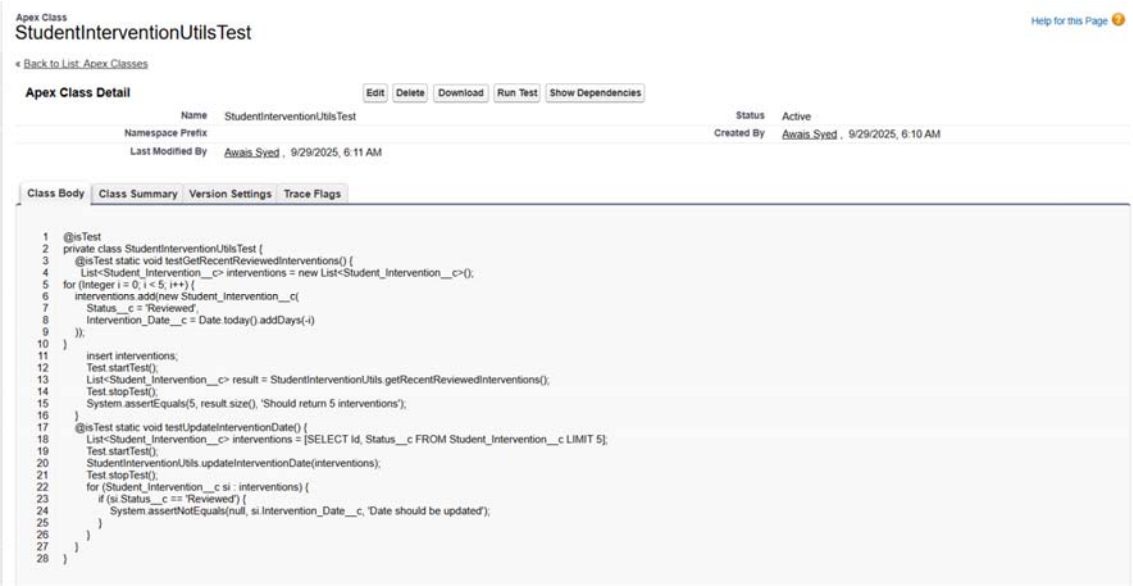
5.12 Test Classes

Use Case: Ensure code reliability and quality by writing automated test classes to verify that Apex classes and triggers work as expected.

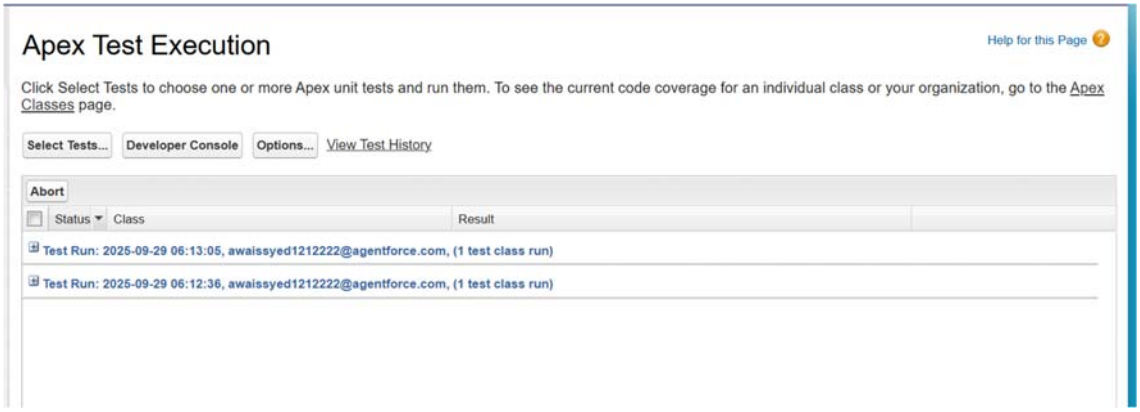
Business Requirement: Test classes must cover at least 75% of the Apex code and validate the logic correctness before deployment to production.

Implementation Understanding

- Use the @isTest annotation to define test classes and methods.
- Create test data within test methods to simulate real use cases.
- Use Test.startTest() and Test.stopTest() to wrap code execution for accurate measurement.
- Use System.assert statements to verify expected results.



Run Tests: Setup → Apex Test Execution, select your test class and run all tests.



Result: By running test classes, you ensure robust Apex code, catching errors early and meeting Salesforce deployment requirements.

5.13 Asynchronous Processing

Use Case: Handle operations asynchronously to improve system performance and user experience by processing tasks in the background without blocking user interactions.

Business Requirement: Use asynchronous Apex to efficiently manage operations such as long-running processes, bulk data updates, scheduled jobs, and chained operations based on project needs.

Type	Description	Use Case	Implement When
Batch Apex	Processes large volumes of records asynchronously in batches	Large data volume processing	Bulk operations, scheduled jobs
Queueable Apex	Flexible async jobs with chaining support	Advanced async logic requiring chaining	Complex background jobs
Scheduled Apex	Runs Apex at scheduled intervals	Regular automation tasks	Periodic data processing
Future Methods	Simple fire-and-forget async operations	Callouts, simple async without chaining	Simple async tasks

Overview of Asynchronous Apex Options

Benefits

- Improves performance by delegating heavy tasks to background processing.
- Helps maintain system limits by processing in manageable chunks.
- Enables chaining of job logic for complex workflows.
- Supports scheduling for automated maintenance and reporting.

Conclusion

This phase focused on mastering Apex programming essentials including SOQL/SOSL queries, collections, control statements, and test classes. It also introduced asynchronous processing concepts to handle complex and bulk operations efficiently in Salesforce.

Implementing these core skills ensures robust, maintainable, and performant Apex code that meets business needs and aligns with Salesforce best practices.

The foundation built in this phase significantly empowers subsequent development phases by enabling dynamic data handling, efficient logic flow, error management, and scalable background processing.