PHASE-5

Developer Part

Use Case Scenario: When a new Appointment is created, check if the Doctor is Available.
Admin flows/validation can't easily cross-check multiple records for time overlap.
Trigger ensures: If the doctor has another appointment at the same Date/Time → throw an error.

We implemented a **before insert/update trigger** on the Appointment__c object to **prevent doctors from being double-booked**. The trigger calls a handler method that checks if another appointment already exists for the same doctor at the same date and time. If a conflict is found, the trigger throws an error and stops the record from being saved. This ensures that no doctor can have two appointments scheduled at the same time.

For testing, we created test data using our **TestDataFactory** (Clinic, Doctor, and Patient) and then inserted one valid appointment. Next, we attempted to insert another appointment with the same doctor and time slot. As expected, the trigger prevented the second appointment and returned the correct error message. We also tested a valid scenario where appointments at different times were allowed to be created successfully.

I followed the best practice that obeys governor limit and trigger handler Pattern

AppointmentTrigger.apxt | MissedAppointmentsBatch.apxc | AppointmentNotificationTest.apxc | AppointmentTriggerTest.apxc

Code Coverage: None | API Version: 64

```
trigger AppointmentTrigger on Appointment__c (before insert,before update) {
    if (Trigger.isBefore &&(Trigger.isInsert || Trigger.isUpdate)){
        AppointmentTriggerHandler.preventDoubleBooking(Trigger.new);

    }
    if (Trigger.isAfter && Trigger.isInsert) {
    for (Appointment__c app : Trigger.new) {
        if (app.Patient__c != null) {
    AppointmentNotification.sendNotification(app.Patient__c, app.Id);
}


}
}
}
```

This is the AppointmentTrigger that executes before insert and before update, which calls the appointment Trigger handler

```
public class AppointmentTriggerHandler {
    public static void preventDoubleBooking(List<Appointment__c> newApps){
        //creating sets for doctor and its date from incoming appointment
        Set<Id> doctorIds = new Set<Id>();
        Set<DateTime> times = new Set<DateTime>();

        for (Appointment__c app:newApps){
            if (app.Doctor__c != null && app.Appointment_DateTime__c != null){
                doctorIds.add(app.Doctor__c);
                times.add(app.Appointment_DateTime__c);
            }
        }
         // retriving existing appointments for these doctors and times
          List<Appointment__c> conflicts = new List<Appointment__c>();
          if(!doctorIds.isEmpty() && !times.isEmpty()){
            conflicts = [
                SELECT Id, Doctor__c, Appointment_DateTime__c
                FROM Appointment__c
                WHERE Doctor__c IN :doctorIds
                AND Appointment_DateTime__c IN :times
            ];
        }
        // creatning map for doctors and their bookings
        Map<Id,Set<DateTime>> doctorSchedule = new Map<Id,Set<DateTime>>();
        for(Appointment__c c : conflicts){
            if(!doctorSchedule.containsKey(c.Doctor__c)){
                doctorSchedule.put(c.Doctor__c, new Set<DateTime>());
            }
            doctorSchedule.get(c.Doctor__c).add(c.Appointment_DateTime__c);
        }
        // check that new appointments must obey doctorSchedule
        for(Appointment__c app : newApps){
            if(app.Doctor__c != null && app.Appointment_DateTime__c != null){
                if(doctorSchedule.containsKey(app.Doctor__c) &&
                   doctorSchedule.get(app.Doctor__c).contains(app.Appointment_DateTime__c)){
                    app.addError('This doctor already has an appointment at this time.');
```

```
        }
        // creatning map for doctors and their bookings
        Map<Id,Set<DateTime>> doctorSchedule = new Map<Id,Set<DateTime>>();
        for(Appointment__c c : conflicts){
            if(!doctorSchedule.containsKey(c.Doctor__c)){
                doctorSchedule.put(c.Doctor__c, new Set<DateTime>());
            }
            doctorSchedule.get(c.Doctor__c).add(c.Appointment_DateTime__c);
        }
        // check that new appointments must obey doctorSchedule
        for(Appointment__c app : newApps){
            if(app.Doctor__c != null && app.Appointment_DateTime__c != null){
                if(doctorSchedule.containsKey(app.Doctor__c) &&
                   doctorSchedule.get(app.Doctor__c).contains(app.Appointment_DateTime__c)){
                    app.addError('This doctor already has an appointment at this time.');
                }

            }
        }
    }
}
```

Then, I tested this trigger using the first Test Data Factory and then run the test in the test class.

```
@isTest
public class TestDataFactory2 {

    public static Doctor__c createDoctor(String name, String specialty, Integer yearsOfExp, Id clinicId){
        // Auto-create Account if clinicId is null
        if (clinicId == null) {
        // Generate dummy Clinic Registration ID
        String regId = 'REG-' + String.valueOf(Math.abs(Crypto.getRandomInteger())).substring(0,6);

        Account clinic = new Account(
            Name = name + ' Clinic',
            Clinic_Registration_ID__c = regId // populate required field
        );
        insert clinic;
        clinicId = clinic.Id;
    }

        Doctor__c doc = new Doctor__c(
            Full_Name__c = name,
            Specialty__c = specialty,
            Phone__c = '9999999999',
            Email__c = 'testdoctor@example.com',
            Years_of_Experience__c = yearsOfExp,
            Clinic__c = clinicId
        );
        insert doc;
        return doc;
    }

    public static Contact createPatient(String lastName){
        Contact pat = new Contact(
            LastName = lastName,
            Date_of_Birth__c = Date.newInstance(1990,1,1),
            Gender__c = 'Male',        // required field
            Blood_Group__c = 'A+'
        );
```

```
        );
        insert pat;
        return pat;
    }

    public static Appointment__c createAppointment(Id doctorId, Id patientId, DateTime appTime){
        Appointment__c app = new Appointment__c(
            Doctor__c = doctorId,
            Patient__c = patientId,
            Appointment_DateTime__c = appTime,
            Status__c = 'Scheduled'
        );
        insert app;
        return app;
    }
}
```

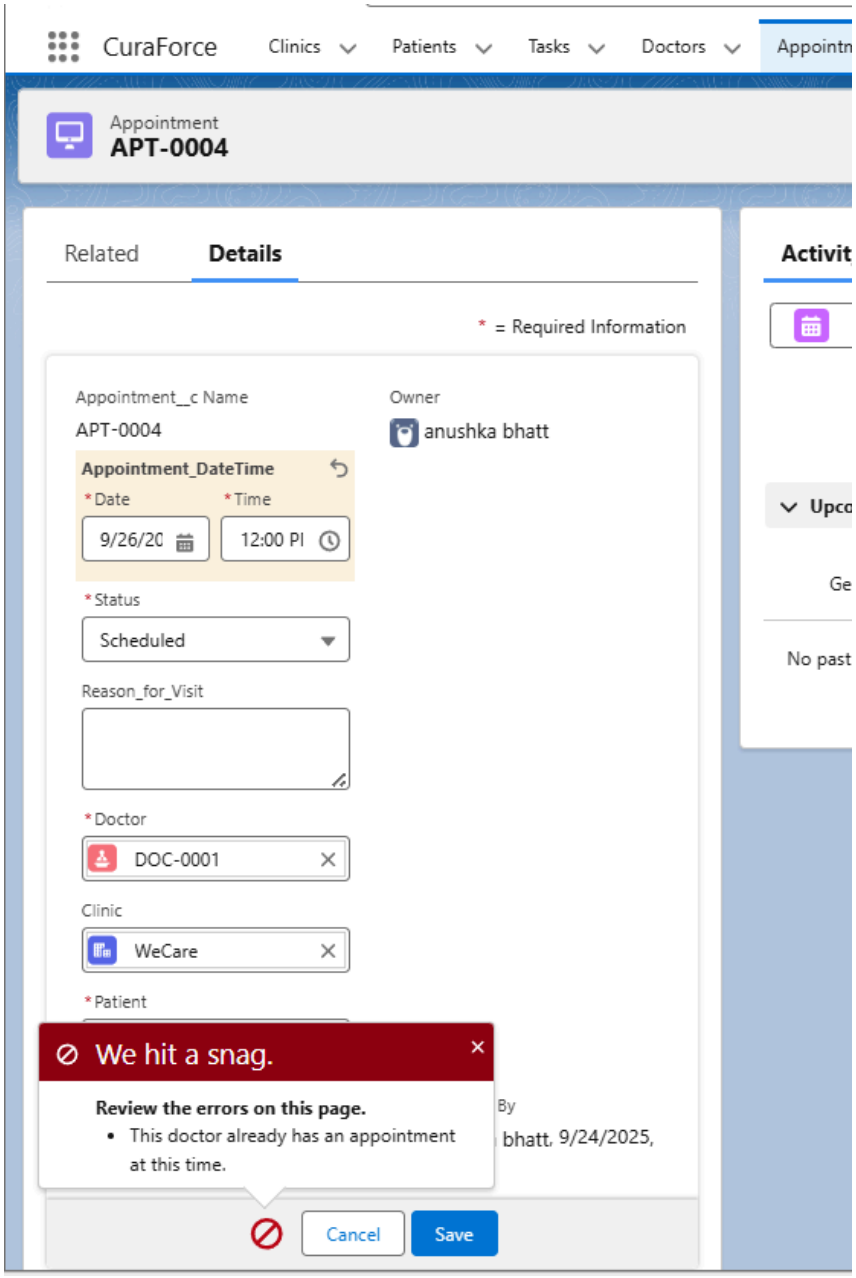
Code Coverage: AppointmentTriggerTest.testDoubleBookingPrevention 100% API Version: 64

```
12          }
13           // retriving existing appointments for these doctors and times
14           List<Appointment__c> conflicts = new List<Appointment__c>();
15           if(!doctorIds.isEmpty() && !times.isEmpty()){
16             conflicts = [
17                 SELECT Id, Doctor__c, Appointment_DateTime__c
18                 FROM Appointment__c
19                 WHERE Doctor__c IN :doctorIds
20                 AND Appointment_DateTime__c IN :times
21             ];
22          }
23          // creatning map for doctors and their bookings
24          Map<Id,Set<DateTime>> doctorSchedule = new Map<Id,Set<DateTime>>();
25          for(Appointment__c c : conflicts){
26              if(!doctorSchedule.containsKey(c.Doctor__c)){
27                  doctorSchedule.put(c.Doctor__c, new Set<DateTime>());
28              }
29              doctorSchedule.get(c.Doctor__c).add(c.Appointment_DateTime__c);
30          }
31          // check that new appointments must obey doctorSchedule
32          for(Appointment__c app : newApps){
33              if(app.Doctor__c != null && app.Appointment_DateTime__c != null){
34                  if(doctorSchedule.containsKey(app.Doctor__c) &&
35                      doctorSchedule.get(app.Doctor__c).contains(app.Appointment_DateTime__c)){
36                       app.addError('This doctor already has an appointment at this time.');
37                  }
```

Logs | Tests | Checkpoints | Query Editor | View State | Progress | Problems

Status	Test Run	Enqueued Time	Duration
✗	TestRun @ 6:28:16 pm		
✗	AppointmentNotificationTest		
✗	testSendNotification		0:00
✓	TestRun @ 6:31:32 pm		
✓	TestRun @ 7:01:34 pm		
✓	AppointmentTriggerTest		
✓	testDoubleBookingPrevention		0:00
✓	testValidAppointment		0:00

Test case passed successfully



I tested the functionality in my org also.

Appointment Notification Trigger (Future Method)

In addition to the double-booking check, system extended the Appointment__c trigger with an **after insert** context to send **appointment notifications asynchronously**. After a new appointment is successfully created, the trigger calls the AppointmentNotification.sendNotification method, which is defined as a @future method. This ensures the notification logic runs in the background without delaying the main transaction. For

demo purposes, the method simulates sending an email to the patient by logging a debug message with the patient's email and appointment details.

To test this functionality,I inserted a valid appointment in my test class and wrapped the process in Test.startTest() and Test.stopTest() to ensure the asynchronous method executed. Verified that the future method ran without errors, confirming that notifications are properly enqueued whenever a new appointment is created.

```
    public static Appointment__c createAppointment(Id doctorId, Id patientId, DateTime appTime){
        Appointment__c app = new Appointment__c(
            Doctor__c = doctorId,
            Patient__c = patientId,
            Appointment_DateTime__c = appTime,
            Status__c = 'Scheduled'
        );
        insert app;
        return app;
    }
}
```

Made changes to this test data factory to create appointment with scheduled status

```
@isTest
private class AppointmentNotificationTest {
    @isTest
    static void testSendNotification() {

        Doctor__c doc = TestDataFactory2.createDoctor('Dr. Demo', 'General Physician', 8, null);
        Contact pat = TestDataFactory2.createPatient('Watson');
        Appointment__c app = TestDataFactory2.createAppointment(doc.Id, pat.Id, System.now().addDays(1));

        Test.startTest();
        AppointmentNotification.sendNotification(pat.Id, app.Id);
        Test.stopTest();

        // No exception means success
        System.assert(true, 'Future method executed without error');
    }
}
```

```
public class MissedAppointmentsBatch implements Database.Batchable<SObject> {
    public Database.QueryLocator start(Database.BatchableContext bc) {
        return Database.getQueryLocator(
            'SELECT Id, Status__c, Appointment_DateTime__c ' +
            'FROM Appointment__c WHERE Status__c = \'Scheduled\' ' +
            'AND Appointment_DateTime__c < :System.now()'
        );
    }

    public void execute(Database.BatchableContext bc, List<Appointment__c> scope) {
        for (Appointment__c app : scope) {
            app.Status__c = 'Missed';
        }
        update scope;
    }

    public void finish(Database.BatchableContext bc) {
        System.debug('Batch completed: Missed appointments updated.');
    }
```

```
ublic class AppointmentNotification {
future
    public static void sendNotification(Id contactId, Id appointmentId) {
        Contact c = [SELECT Id, LastName, Email FROM Contact WHERE Id = :contactId LIMIT 1];
        Appointment__c app = [SELECT Id, Appointment_DateTime__c FROM Appointment__c WHERE Id = :appointmentId LIMIT 1]

        // For demo: just log instead of actual email
        System.debug('Sending email to ' + c.Email +
                     ' for appointment on ' + app.Appointment_DateTime__c);
    }
```