**PetCare+ –>Specialized Veterinary Treatment Slot Booking System**

**PHASE 5**

1. Classes & Objects

| Code / File | Concept | Purpose |
| --- | --- | --- |
| SlotTriggerHandler | Apex Class | Handles all business logic for Slot\_\_c (validations, updates, owner assignment). |
| AppointmentHelper | Apex Class | Provides reusable helper methods for fetching booked/available slots. |
| SlotNotificationQueueable | Apex Class | Asynchronous notifications for booked slots (emails to pet owners). |
| @isTest SlotTriggerHandlerTest | Apex Class | Tests the trigger logic, double booking, owner assignment, notifications. |

2. Apex Triggers (before/after insert/update)

| Code / File | Concept | Purpose |
| --- | --- | --- |
| SlotTrigger | Apex Trigger | Runs automatically on Slot\_\_c records for before insert/update and after insert/update events. Delegates logic to SlotTriggerHandler. |

3. Trigger Design Pattern

| Code / File | Concept | Purpose |
| --- | --- | --- |
| SlotTrigger → SlotTriggerHandler | Trigger → Handler pattern | Separates trigger (event handling) from logic (validations & updates) for cleaner, maintainable code. |

4. SOQL & SOSL

| Code / File | Concept | Purpose |
| --- | --- | --- |
| SlotTriggerHandler | SOQL | Queries Slot\_\_c to check for existing booked slots (prevent double booking). |
| SlotNotificationQueueable | SOQL | Queries Slot\_\_c, Pet\_\_c, and Contact to send notifications. |
| AppointmentHelper | SOQL | Queries booked or available slots for a doctor. |

5. Collections (List, Set, Map)

| Code / File | Concept | Purpose |
| --- | --- | --- |
| SlotTriggerHandler | List,Set, Map | Bulk-safe validation: track doctors, appointment times, conflicting records. |
| SlotNotificationQueueable | List,Map, Set | Collect owners and emails for notification; prevents duplicates. |
| AppointmentHelper | List | Returns lists of booked or available slots. |

6. Control Statements (if, for, switch)

| Code / File | Concept | Purpose |
| --- | --- | --- |
| SlotTriggerHandler | if, for | Check past dates, double bookings, cancellation window. |
| SlotNotificationQueueable | if, for | Loop over slots and owners to prepare emails. |
| SlotTriggerHandlerTest | if, for, try/catch | Validate different scenarios, handle expected exceptions. |

7. Queueable Apex / Future Methods

| Code / File | Concept | Purpose |
| --- | --- | --- |
| SlotNotificationQueueable | Queueable Apex | Runs asynchronously to send appointment notifications without blocking the main transaction. |

8. Exception Handling

| Code / File | Concept | Purpose |
| --- | --- | --- |
| SlotTriggerHandlerTest | try/catch | Catches expected DMLException for double-booked slots. |
| SlotTriggerHandler | addError() | Adds user-friendly validation errors for past dates, double bookings, and cancellation rules. |

9. Test Classes

| Code / File | Concept | Purpose |
| --- | --- | --- |
| SlotTriggerHandlerTest | @isTest | Validates trigger/handler logic, double-book prevention, owner assignment, and notifications. |

| **Concept** | **Implemented By** |
| --- | --- |
| Classes & Objects | SlotTriggerHandler, AppointmentHelper, SlotNotificationQueueable, SlotTriggerHandlerTest |
| Apex Triggers | SlotTrigger |
| TriggerDesign Pattern | SlotTrigger → SlotTriggerHandler |
| SOQL | SlotTriggerHandler, SlotNotificationQueueable, AppointmentHelper |
| Collections | SlotTriggerHandler, SlotNotificationQueueable, AppointmentHelper |
| Control Statements | SlotTriggerHandler, SlotNotificationQueueable, SlotTriggerHandlerTest |
| Queueable Apex | SlotNotificationQueueable |
| Exception Handling | SlotTriggerHandler (addError), SlotTriggerHandlerTest (try/catch) |
| Test Classes | SlotTriggerHandlerTest |

**1) Trigger (SlotTrigger)**

trigger SlotTrigger on Slot\_\_c (before insert, before update, after insert) {

if (Trigger.isBefore) {

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

if (Trigger.isUpdate) SlotTriggerHandler.beforeUpdate(Trigger.new, Trigger.oldMap);

}

if (Trigger.isAfter) {

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

// if afterUpdate logic is empty, you can comment it for now

// if (Trigger.isUpdate) SlotTriggerHandler.afterUpdate(Trigger.new, Trigger.oldMap);

}

}

**2) Trigger Handler (SlotTriggerHandler)**

public with sharing class SlotTriggerHandler {

// --- BEFORE INSERT ---

public static void beforeInsert(List<Slot\_\_c> newSlots) {

validateNoPastDate(newSlots);

validateNoDoubleBooking(newSlots, null);

}

// --- BEFORE UPDATE ---

public static void beforeUpdate(List<Slot\_\_c> newSlots, Map<Id, Slot\_\_c> oldMap) {

validateNoPastDate(newSlots);

validateNoDoubleBooking(newSlots, oldMap);

// Cancellation window removed because Date\_\_c is date-only

}

// --- AFTER INSERT ---

public static void afterInsert(List<Slot\_\_c> newSlots) {

List<Id> slotIds = new List<Id>();

for (Slot\_\_c s : newSlots) slotIds.add(s.Id);

//if (!slotIds.isEmpty()) System.enqueueJob(new SlotNotificationQueueable(slotIds));

}

// --- AFTER UPDATE ---

public static void afterUpdate(List<Slot\_\_c> newSlots, Map<Id, Slot\_\_c> oldMap) {

// optional: can enqueue notifications if needed on updates

}

// --- VALIDATIONS ---

private static void validateNoPastDate(List<Slot\_\_c> slots) {

Date today = Date.today();

for (Slot\_\_c s : slots) {

if (s.Date\_\_c != null && s.Date\_\_c < today) {

s.addError('Cannot book a slot in the past.');

}

}

}

private static void validateNoDoubleBooking(List<Slot\_\_c> newSlots, Map<Id, Slot\_\_c> oldMap) {

Set<Id> userIds = new Set<Id>();

Set<Date> apptDates = new Set<Date>();

Set<Id> newIds = new Set<Id>();

Map<String, List<Slot\_\_c>> keyToNew = new Map<String, List<Slot\_\_c>>();

for (Slot\_\_c s : newSlots) {

if (s.User\_\_c == null || s.Date\_\_c == null) continue;

Boolean shouldCheck = false;

if (oldMap == null) shouldCheck = true;

else {

Slot\_\_c old = oldMap.get(s.Id);

if (old == null) shouldCheck = true;

else if (s.User\_\_c != old.User\_\_c || s.Date\_\_c != old.Date\_\_c || s.Status\_\_c != old.Status\_\_c) shouldCheck = true;

}

if (!shouldCheck) continue;

String key = String.valueOf(s.User\_\_c) + '|' + String.valueOf(s.Date\_\_c);

if (!keyToNew.containsKey(key)) keyToNew.put(key, new List<Slot\_\_c>());

keyToNew.get(key).add(s);

userIds.add(s.User\_\_c);

apptDates.add(s.Date\_\_c);

if (s.Id != null) newIds.add(s.Id);

}

if (userIds.isEmpty()) return;

List<Slot\_\_c> existing = [SELECT Id, User\_\_c, Date\_\_c, Status\_\_c

FROM Slot\_\_c

WHERE User\_\_c IN :userIds

AND Date\_\_c IN :apptDates

AND Status\_\_c = 'Booked'

AND Id NOT IN :newIds];

Set<String> existingKeys = new Set<String>();

for (Slot\_\_c e : existing) {

existingKeys.add(String.valueOf(e.User\_\_c) + '|' + String.valueOf(e.Date\_\_c));

}

for (String k : keyToNew.keySet()) {

List<Slot\_\_c> conflictGroup = keyToNew.get(k);

if (existingKeys.contains(k)) {

for (Slot\_\_c s : conflictGroup) {

s.addError('This user already has a booked slot on the selected date.');

}

} else if (conflictGroup.size() > 1) {

for (Slot\_\_c s : conflictGroup) {

s.addError('Multiple bookings in the same transaction for the same user/date.');

}

}

}

}

}

**3) Async Notifications (Queueable)**

public with sharing class SlotNotificationQueueable implements Queueable {

private List<Id> slotIds;

public SlotNotificationQueueable(List<Id> ids) { this.slotIds = ids; }

public void execute(QueueableContext context) {

List<Slot\_\_c> slots = [SELECT Id, Date\_\_c, Pet\_\_c, Pet\_\_r.Owner\_\_c, Pet\_\_r.Name, User\_\_c

FROM Slot\_\_c WHERE Id IN :slotIds];

Set<Id> ownerIds = new Set<Id>();

for (Slot\_\_c s : slots) if (s.Pet\_\_r != null && s.Pet\_\_r.Owner\_\_c != null) ownerIds.add(s.Pet\_\_r.Owner\_\_c);

Map<Id, Contact> owners = new Map<Id, Contact>();

if (!ownerIds.isEmpty()) {

for (Contact c : [SELECT Id, Email, Name FROM Contact WHERE Id IN :ownerIds]) owners.put(c.Id, c);

}

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

for (Slot\_\_c s : slots) {

if (s.Pet\_\_r == null || s.Pet\_\_r.Owner\_\_c == null) continue;

Contact owner = owners.get(s.Pet\_\_r.Owner\_\_c);

if (owner == null || String.isBlank(owner.Email)) continue;

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

mail.setToAddresses(new String[] { owner.Email });

mail.setSubject('PetCare+ Appointment Confirmation');

String body = 'Hello ' + owner.Name + ',\n\n' +

'Your appointment for pet "' + s.Pet\_\_r.Name + '" is scheduled on ' + s.Date\_\_c.format() + '.\n\n' +

'Thanks,\nPetCare+';

mail.setPlainTextBody(body);

emails.add(mail);

}

if (!emails.isEmpty()) Messaging.sendEmail(emails);

}

}

**4) Helper class (AppointmentHelper)**

public with sharing class AppointmentHelper {

public static List<Slot\_\_c> getBookedSlotsForUserAtDates(Set<Id> userIds, Set<Date> dates) {

if (userIds.isEmpty() || dates.isEmpty()) return new List<Slot\_\_c>();

return [SELECT Id, User\_\_c, Date\_\_c, Status\_\_c

FROM Slot\_\_c

WHERE User\_\_c IN :userIds

AND Date\_\_c IN :dates

AND Status\_\_c = 'Booked'];

}

public static List<Slot\_\_c> getAvailableSlotsForUser(Id userId, Date fromDate) {

return [SELECT Id, Date\_\_c FROM Slot\_\_c

WHERE User\_\_c = :userId

AND Date\_\_c >= :fromDate

AND Status\_\_c = 'Available'

ORDER BY Date\_\_c ASC

LIMIT 20];

}

}

**Test Class (SlotTriggerHandlerTest)**

private class SlotTriggerHandlerTest {

@isTest static void testDoubleBookingAndOwner() {

User user = [SELECT Id FROM User LIMIT 1];

Contact owner = new Contact(LastName = 'PetOwnerTest', Email='owner@test.com');

insert owner;

Pet\_\_c pet = new Pet\_\_c(Name='TestPet', Owner\_\_c = owner.Id);

insert pet;

Date appt = Date.today().addDays(1);

Slot\_\_c s1 = new Slot\_\_c(Pet\_\_c = pet.Id, User\_\_c = user.Id, Date\_\_c = appt, Status\_\_c = 'Booked');

insert s1;

Slot\_\_c s2 = new Slot\_\_c(Pet\_\_c = pet.Id, User\_\_c = user.Id, Date\_\_c = appt, Status\_\_c = 'Booked');

try {

insert s2;

System.assert(false, 'Expected DMLException due to double booking');

} catch (DmlException ex) {

System.assert(ex.getMessage().contains('already has a booked slot'), 'Unexpected exception: ' + ex.getMessage());

}

Date appt2 = Date.today().addDays(2);

Slot\_\_c s3 = new Slot\_\_c(Pet\_\_c = pet.Id, User\_\_c = user.Id, Date\_\_c = appt2, Status\_\_c = 'Booked');

insert s3;

s3 = [SELECT Id, Name FROM Slot\_\_c WHERE Id = :s3.Id];

System.assertEquals(user.Id, s3.Name, 'OwnerId should be set to the assigned User');

}

}