C868 – Software Capstone Project Summary

Task 2 – Section A

Capstone Proposal Project Name: TechGopher Scheduler - A Client Scheduling Application

Student Name: Evan A Smith

TechGopher Scheduler - A Client Scheduling Application

Table of Contents

TechGopher Scheduler - A Client Scheduling Application

TechGopher Scheduler - A Client Scheduling Application

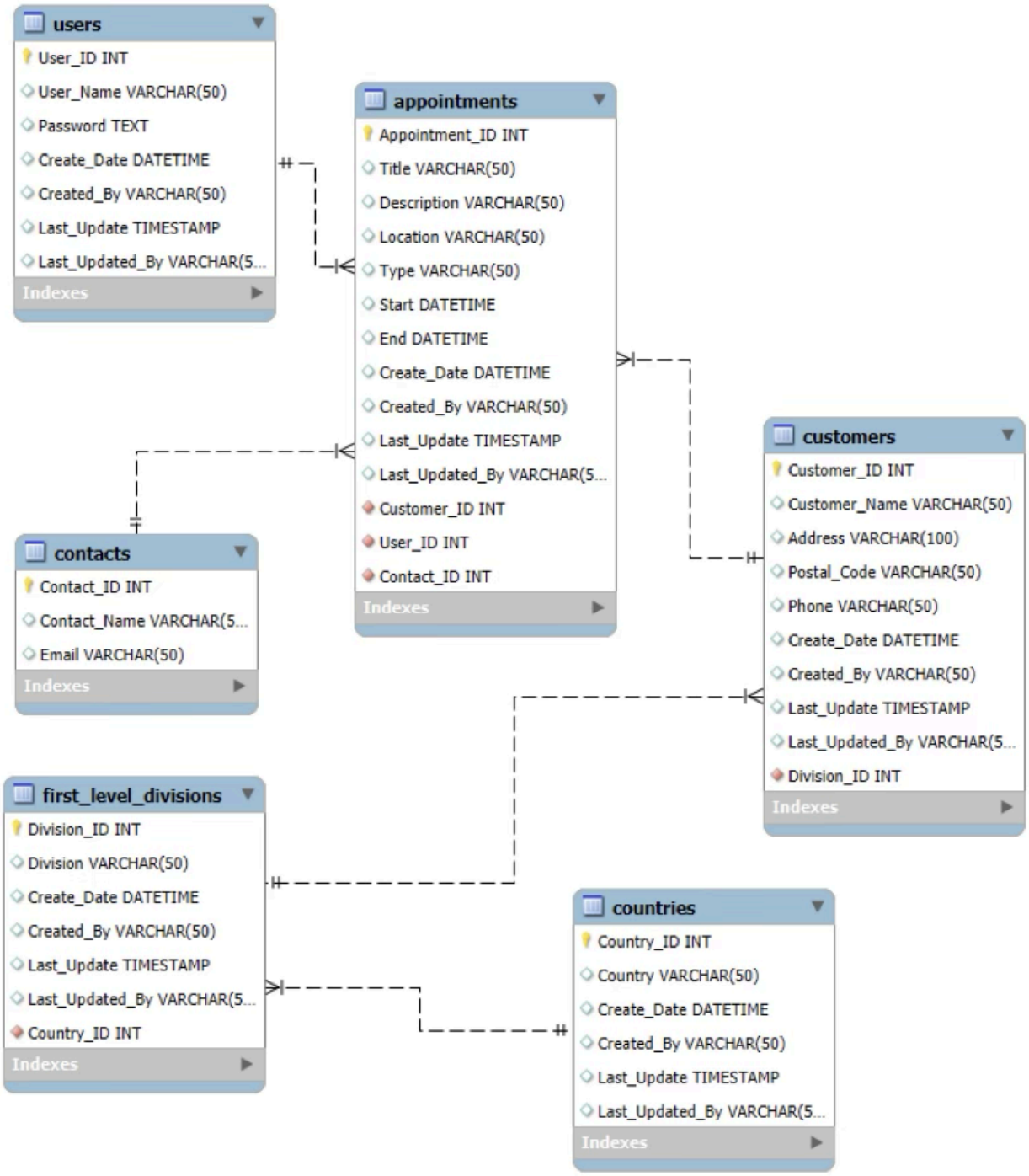
Application Design and Testing

Class Design and Database Class Design

The TechGopher Appointment Scheduling Application follows a relational class design with six primary entities: Users, Appointments, Customers, Contacts, First-Level Divisions, and Countries. Users manage appointments, which are linked to both customers and contacts. Customers belong to a first-level division, which is associated with a country, ensuring location-based organization. Appointments store key details like title, type, and scheduling times, while contacts provide communication references. First-level divisions categorize customers by region, with each division linked to a country. This structured design enforces data integrity, maintains efficient scheduling workflows, and ensures clear entity relationships within the application

The figure below depicts the relational class diagram of the MySQL Database, which interfaces with the TechGopher Scheduling app. Appointments have a many-to-one relationship with customers, users, and contacts, facilitating many users to be able to schedule many appointments and many customers/contacts (used interchangeably here) to be able to schedule many appointments. However, having a one-to-many relationship, only one customer/contact may be assigned to an appointment. A first-level division has been implemented between customers and countries.

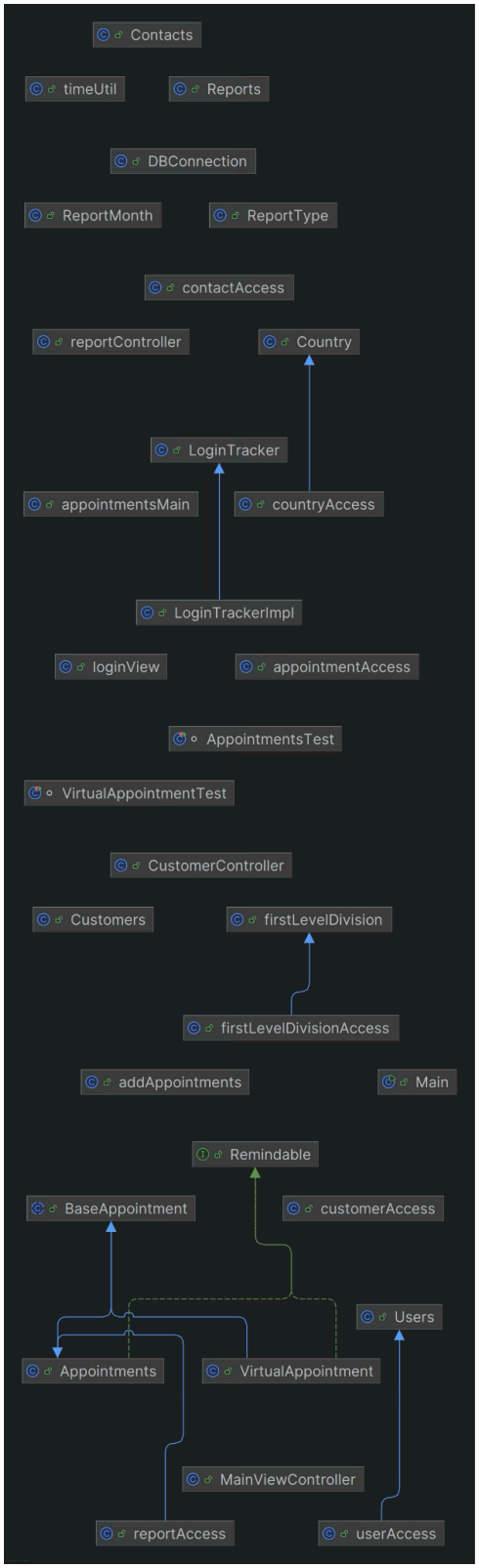
TechGopher Scheduler - A Client Scheduling Application



TechGopher Scheduler - A Client Scheduling Application

The figure below depicts the class diagram of the TechGopher Scheduler App. Several data objects (DAO) are used by the Scheduler App to facilitate the reading and manipulation of the MySQL Database. DBConnection initializes the connection to the database when the TechGopher Scheduler App is launched, and loginView calls the FXML libraries that render the login screen. After entering their credentials, users are taken to the main view, which is controlled by the MainViewController.

TechGopher Scheduler - A Client Scheduling Application



TechGopher Scheduler - A Client Scheduling Application

Application Class Design

The application class design follows a modular approach that adheres to key object-oriented principles such as encapsulation, separation of concerns, and abstraction. The project is organized into distinct layers, including entity classes, data access objects (DAOs), controllers, utility classes, and test classes, ensuring a clean and maintainable architecture.

Entity Layer

At the core of my application are the entity classes Contacts, Customers, Country, firstLevelDivision, Users, BaseAppointment, VirtualAppointment, and Appointment. To promote reusability and maintainability, I implemented BaseAppointment as a parent class for Appointments and VirtualAppointment, leveraging inheritance to share common behaviors while allowing different types of Appointments to be specified in the future.

Data Access Layer (DAO)

The DAO layer consists of classes like contactAccess, countryAccess, firstLevelDivisionAccess, appointmentAccess, customerAccess, reportAccess, and userAccess. These classes handle database interactions, abstracting the persistence logic from the rest of the application. By following the Repository Pattern, I ensure that data operations remain decoupled from business logic, improving scalability and testability.

Controller Layer

TechGopher Scheduler - A Client Scheduling Application

To manage business logic and UI interactions, I structured my application with controllers such as CustomerController, reportController, and MainViewController. These controllers follow the Model-View-Controller (MVC) pattern, acting as intermediaries between the UI and the data access layer. Controllers are used as standard for applications which use the JavaFX and FXML libraries to generate the UI.

Utility & Helper Classes

To support core functionalities of the TechGopher Scheduler App, I included utility classes such as DBConnection for managing database connectivity and timeUtil for handling time-related operations. LoginTracker and its implementation, LoginTrackerImpl, facilitate session tracking and authentication management. These classes help support potential future security enhancements, streamline operations and maintain a clean separation of concerns.

Testing Layer

To ensure software reliability, I incorporated unit testing with the jUnit libraries, using AppointmentsTest and VirtualAppointmentTest. These tests validate the behavior of the scheduling functionality by scheduling and then removing test appointments to help catch regressions early. The test is run from inside of the IntelliJ IDEA IDE by selecting the AppointmentsTest configuration, and verifying that the conditional console output matches “Reminder: Your appointment (Grooming Session) is scheduled at 2025-03-10T14:30”.

The system is designed with multiple entry points:

Main serves as the primary entry point for initializing the application.

appointmentsMain and addAppointments manage appointment scheduling.

loginView handles user authentication and login processes.

TechGopher Scheduler - A Client Scheduling Application

Potential Enhancements

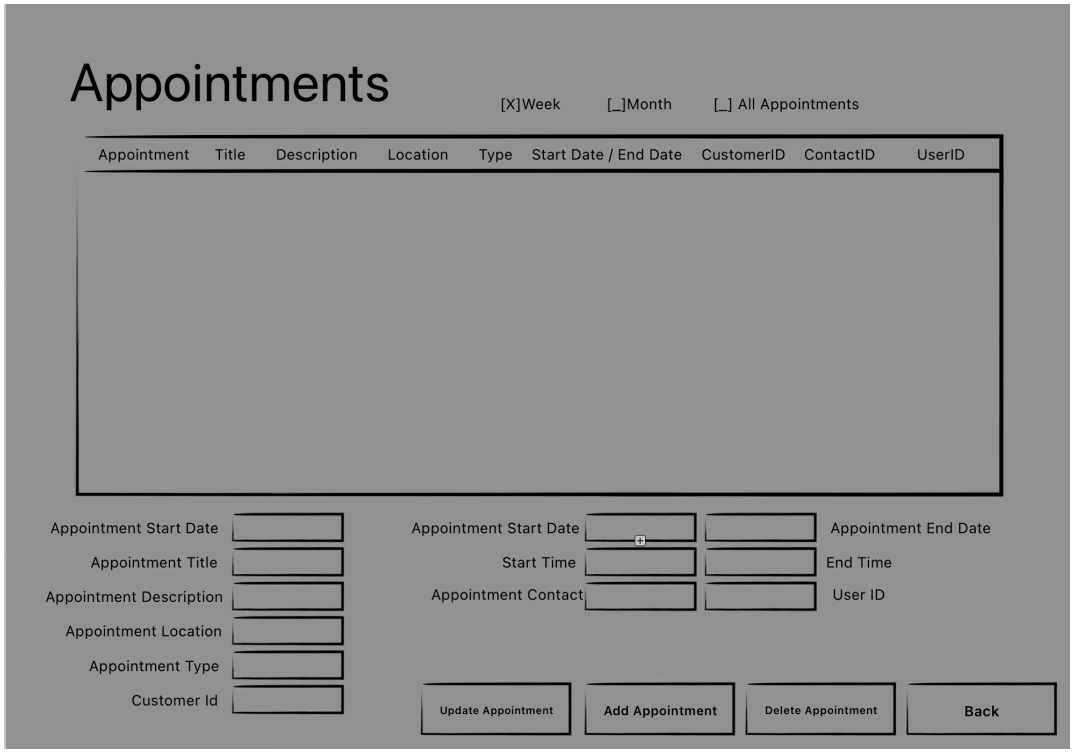
Interface-Based DAO Abstraction: Introducing interfaces for DAO classes (e.g., ICustomerDAO) to enhance flexibility and support dependency injection.

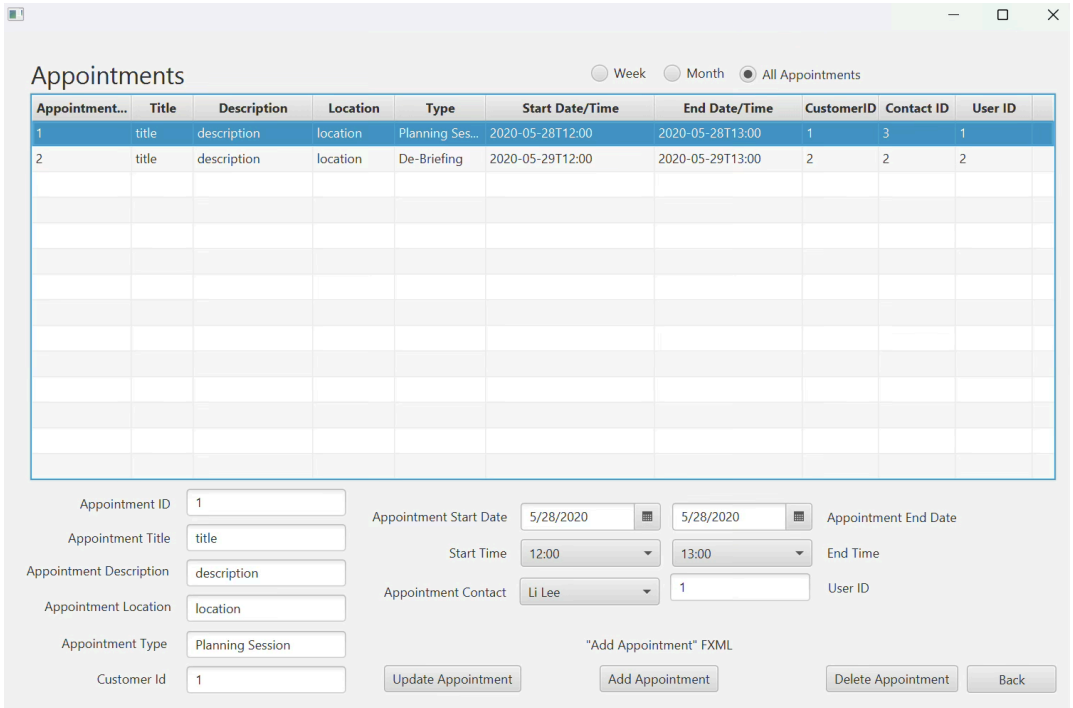
Service Layer Implementation: Logging and introducing a dedicated service layer to LoginTracker would increase the clarity and usefulness of this class if required in the future.

UI Design

Below are the low and high fidelity versions of the Appointment screen, which is the principal screen used in viewing, updating, creating, and deleting scheduled Appointments. There are a number of fields containing information that is essential to creating Appointment records with enough fidelity to meet business needs, including unique Appointment ID, associated CustomerID/ContactID, Appointment Title, Appointment Description, Appointment Location, Appointment Type, and Start and End time and date.

TechGopher Scheduler - A Client Scheduling Application





Selecting an Appointment from the table fills the non-editable, and user-editable fields with the Appointment's associated entries directly from the SQL database. User-editable fields can be altered and then committed by pressing Update, which will commit the changes to the client-schedule Database.

While the usage is somewhat technical in application, and will incur associated training and adoption costs, this is viewed by the business as an acceptable trade-off to develop a valuable new asset in the means to increase the fidelity and accuracy of customer contact records, increasing the potential for marketing tie-ins as well as generally increasing the potential customer contact experiences such a robust database will offer.

Unit Test Plan

The purpose of unit testing for this application is to reduce the total amount of regressions committed by running tests to validate the function of core methods and objects integral to the application logic and validate the integrity and accessibility of all libraries and modules required by the application. This will be accomplished by a systematic approach to testing which catalogues and records all key changes to the unit testing plan throughout development, and a robust approach to continuously run tests when changes to the application are made.

Because the appointment class inherits methods and classes from the majority of features of the application, they will be at the center of the unit test. I used jUnit libraries from Apache Maven to run test scripts that validate the function of the Appointment class each time the code is executed. The outcome of testing is verified through the console each time the application is launched/executed, but it can also be run separately by running the AppointmentsTest configuration in the IntelliJ IDEA IDE.

If a test fails, the output identifies the specific method or functionality that is not behaving as expected. Common failures include incorrect date/time conversions, validation errors, or SQL-related exceptions. When failures occur, remediation involves debugging the failing test case to analyze the root cause, modifying the Appointment class or related dependencies to resolve logic errors, and re-running the tests to confirm successful remediation. This approach ensures that the Appointment class remains stable and functional as the application evolves.

Overview of Unit Test Plan

This unit test is designed to verify two important aspects of the Appointments class. The first part of the test, testAppointmentCreation, ensures that the Appointments object is correctly initialized with the expected values for its attributes. It checks that the appointment ID, title, description, location, type, and associated IDs (customer, user, and contact) are all properly set during the creation of the appointment object. If all these values match the expected ones, it confirms that the appointment has been created as intended.

The second part of the test, testSendReminder, calls the sendReminder() method of the Appointments class. While this test does not explicitly verify any output or state change, it ensures that the method executes without throwing any exceptions. The test is intended to check that the reminder functionality runs without error, though it does not currently verify the actual content or delivery of the reminder itself. A more thorough test could involve capturing and validating the reminder output.

Test Plan Items Required for Testing

To complete the unit tests for the Appointment class, the following components are required:

JUnit 5 (Jupiter) Libraries (via Apache Maven)

TechGopher Scheduler - A Client Scheduling Application

IntelliJ IDEA (for executing the test configuration)

Appointment Class (including constructors, getter methods, and business logic)

CustomerID Class (for verifying correct customer associations)

DBConnector Class (for testing database interactions)

Test Data (sample appointment attributes for validation)

Features

The setUp() method is initializing a test fixture by creating a new Appointments object with predefined values. This method is annotated with @BeforeEach, meaning it runs before each test method in the test class. This ensures that every test starts with a fresh instance of appointment, preventing unintended interactions between tests.

By setting up this instance before each test, it ensures that tests are independent and do not modify a shared state, which could lead to unreliable test results. Each test can then safely operate on a known appointment instance without interference from other tests.

The testAppointmentCreation() method is a unit test that verifies the correct instantiation of an Appointments object by checking if its attributes match the expected values. It Uses Assertions.assertEquals(expected, actual) to compare expected values with the actual values retrieved from the Appointments object. If any assertion fails, the test will fail, indicating an issue in the constructor or getter methods. This test ensures that the Appointments constructor properly initializes all fields and that the getter methods return the expected values. If this test fails, it indicates a potential issue with either the constructor, parameter assignments, or getter methods. It serves as a foundational validation step before testing other functionalities related to appointments.

TechGopher Scheduler - A Client Scheduling Application

Tasks

To execute this unit test the following steps are required:

1. Write the code to be tested, in this case by setting up validation.
2. Create a class named “AppointmentsTest” inside the test package.
3. From inside the AppointmentsTest package, import the class under test from the correct package. In this case, that is “Appointments” from the “Model” package, or import model.Appointments using import statements. In this case, the junit-jupiter-api libraries: import org.junit.jupiter.api.Assertions;, …BeforeEach;, and …Test;.
4. Create and utilize method and operand `setUp() {`

   `appointment =` to then assign `new Appointments(1, "Grooming Session", "Dog grooming session",`

   `"New York Salon", "Grooming",`

   `LocalDateTime.of(2025, 3, 10, 14, 30),`

   `LocalDateTime.of(2025, 3, 10, 15, 30),`

   `101, 202, 303);` to set time data from the Java `LocalDateTime` library.
5. Create and utilize method and operand

   `void testAppointmentCreation() {`

   to fill with test data:

   ```
   Assertions.assertEquals(1, appointment.getAppointmentID());
   Assertions.assertEquals("Grooming Session",
   appointment.getAppointmentTitle());
   Assertions.assertEquals("Dog grooming session",
   appointment.getAppointmentDescription());
   ```

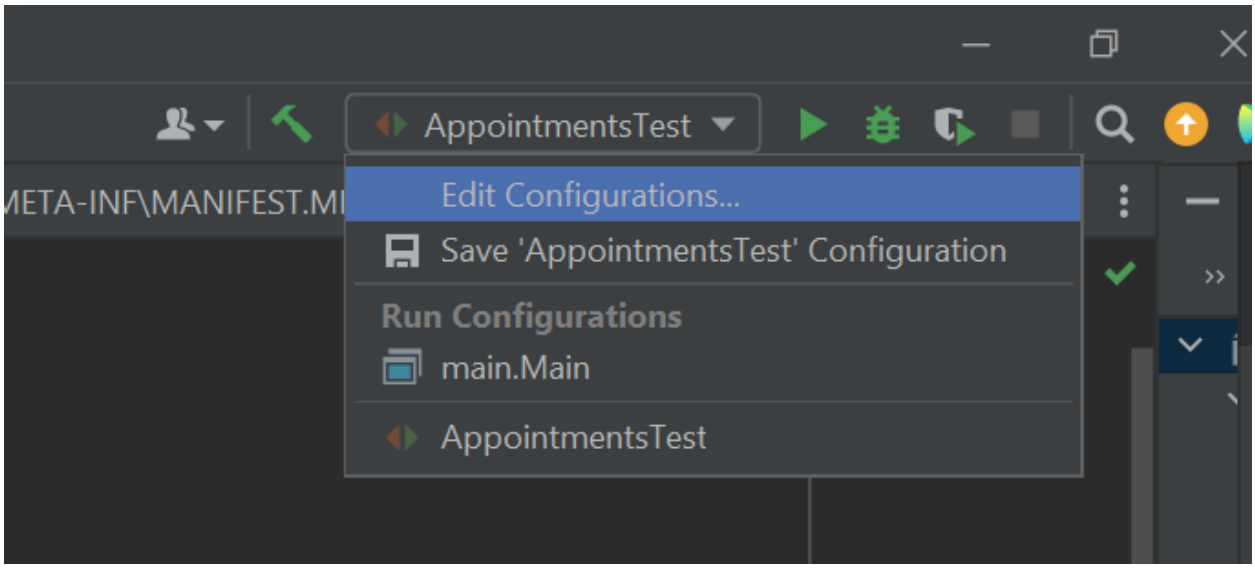
```
Assertions.assertEquals("New York Salon",
appointment.getAppointmentLocation());
Assertions.assertEquals("Grooming", appointment.getAppointmentType());
Assertions.assertEquals(101, appointment.getCustomerID());
Assertions.assertEquals(202, appointment.getUserID());
Assertions.assertEquals(303, appointment.getContactID());
```

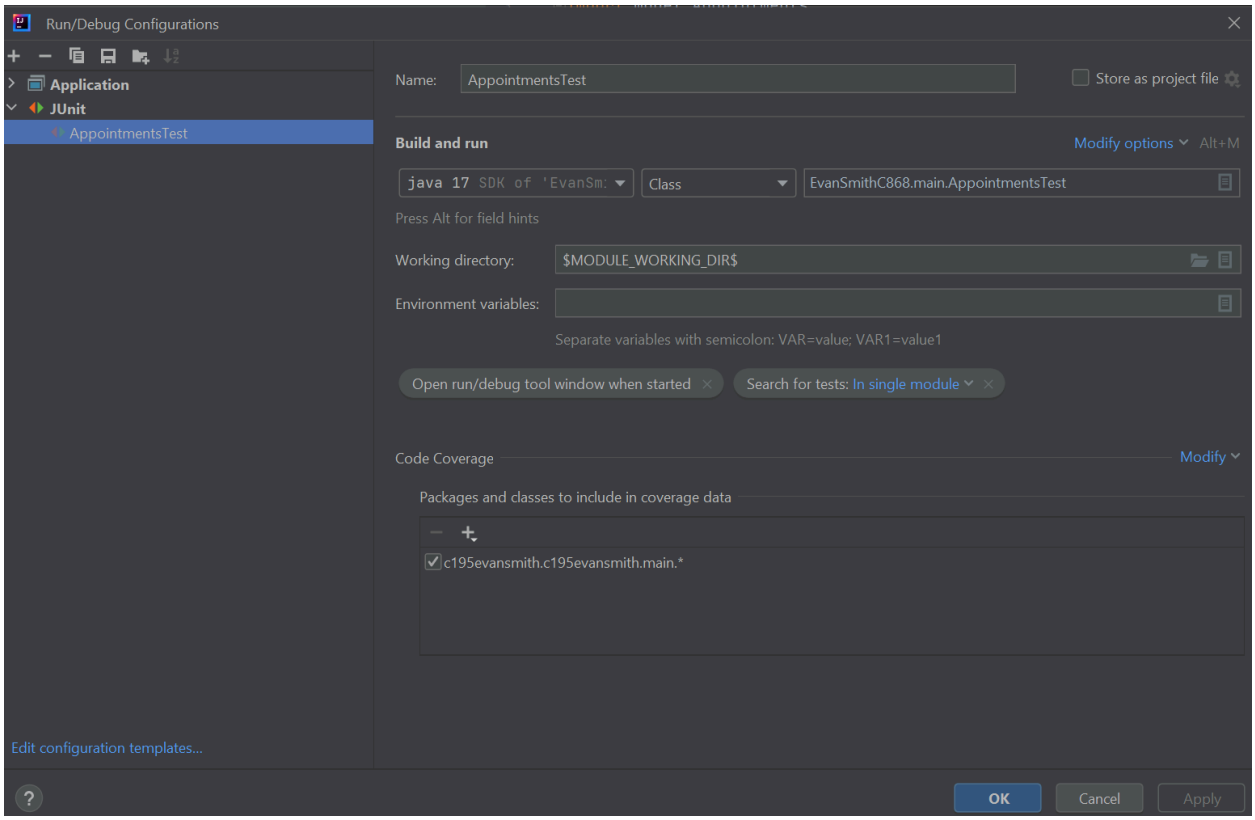
6. Create and utilize method and operand `void testSendReminder() {`

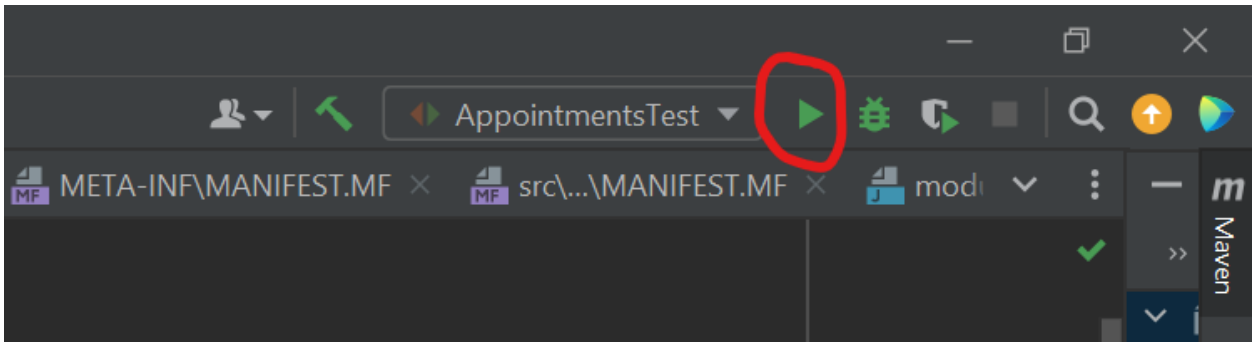
   ```
   // Capture console output
   appointment.sendReminder();
   ```

   to generate console output.

7. Run the test by selecting the AppointmentsTest Configuration in IntelliJ Idea.







8. Examine the output to determine pass or fail.

Test Needs

Supporting the application itself is MySQL Server running on Windows. This will be running in the Software II virtual machine, and it contains the client_schedule database that the TechGopher

Application uses to create and edit its data. The jUnit unit testing library is automatically included in the Apache Maven library used in developing the source code of this application.

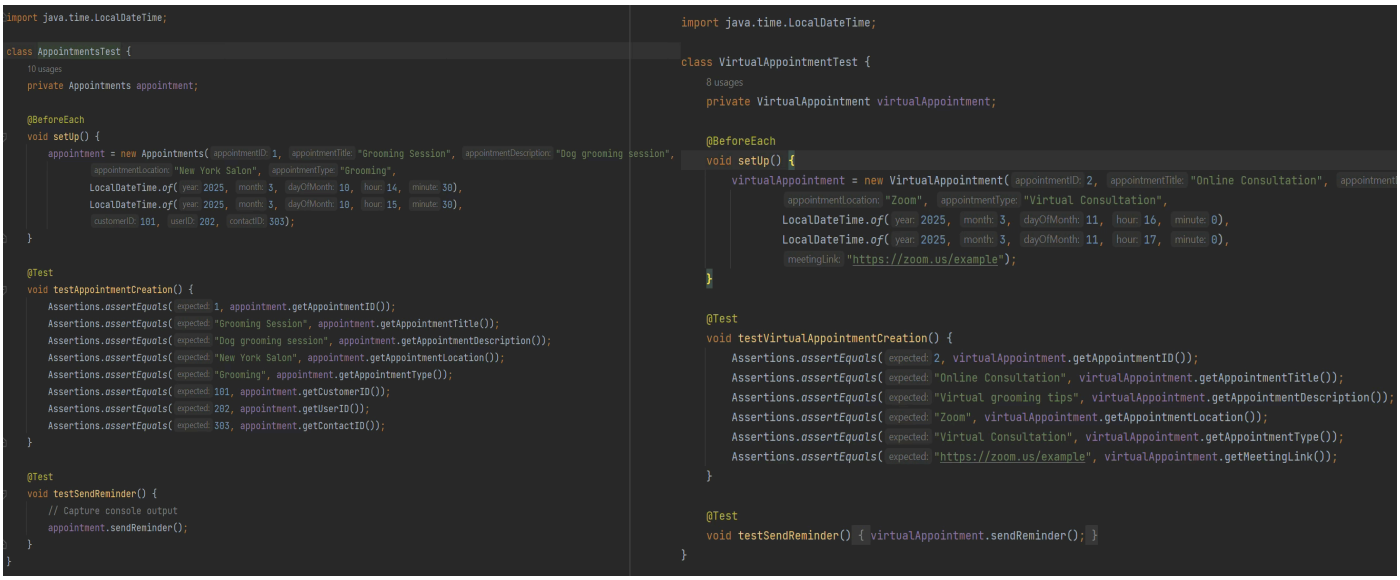
Pass/Fail Criteria

The success of the testAppointmentCreation() test was determined by verifying that all assertions passed, meaning the actual values retrieved from the Appointments object matched the expected values. Additionally, the test had to execute without exceptions, ensuring that the constructor and getter methods functioned correctly. If JUnit reported a successful test run with no failures, it confirmed that the Appointments object was properly instantiated and its attributes were accurately assigned and retrievable. A positive test result followed a protocol where the test was marked as successful, confirming the correctness of the implementation. This allowed development to proceed with additional testing for other functionalities, such as modifying appointments, handling database interactions, and sending reminder notifications.

If the test failed, the next step was to identify the cause of the failure. JUnit provided error messages that indicated whether an assertion failed or if an exception was thrown, which helped pinpoint the issue. Debugging efforts focused on checking the constructor for incorrect parameter assignments, ensuring getter methods properly retrieved stored values, and addressing any data type mismatches, such as issues with date and time formatting. After making the necessary corrections, the test was rerun to confirm the fix. Proper documentation was required for each failure, including a summary of the issue, an analysis of the root cause, the remediation steps taken, and confirmation that the updated test passed successfully. This documentation ensures transparency in the testing process and helps prevent similar issues in the future.

TechGopher Scheduler - A Client Scheduling Application

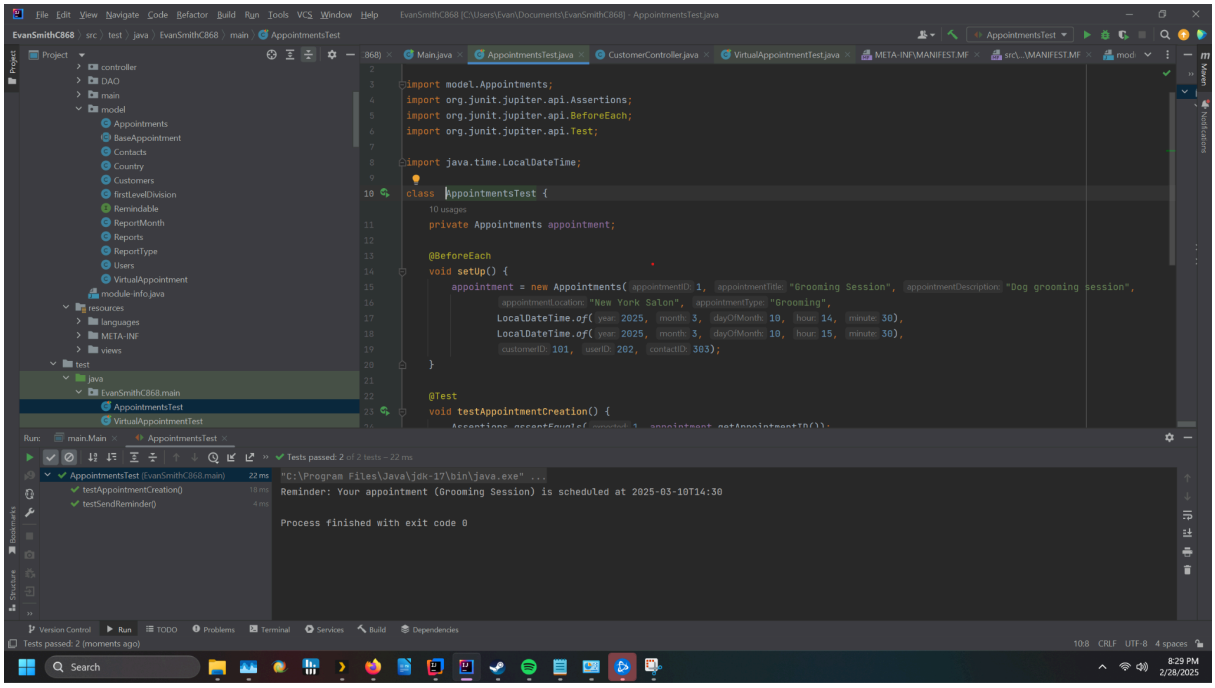
Specifications



```java
import java.time.LocalDateTime;

class AppointmentsTest {
    10 usages
    private Appointments appointment;

    @BeforeEach
    void setUp() {
        appointment = new Appointments( appointmentID: 1, appointmentTitle: "Grooming Session", appointmentDescription: "Dog grooming session",
                appointmentLocation: "New York Salon", appointmentType: "Grooming",
                LocalDateTime.of( year: 2025, month: 3, dayOfMonth: 10, hour: 14, minute: 30),
                LocalDateTime.of( year: 2025, month: 3, dayOfMonth: 10, hour: 15, minute: 30),
                customerID: 101, userID: 202, contactID: 303);
    }

    @Test
    void testAppointmentCreation() {
        Assertions.assertEquals( expected: 1, appointment.getAppointmentID());
        Assertions.assertEquals( expected: "Grooming Session", appointment.getAppointmentTitle());
        Assertions.assertEquals( expected: "Dog grooming session", appointment.getAppointmentDescription());
        Assertions.assertEquals( expected: "New York Salon", appointment.getAppointmentLocation());
        Assertions.assertEquals( expected: "Grooming", appointment.getAppointmentType());
        Assertions.assertEquals( expected: 101, appointment.getCustomerID());
        Assertions.assertEquals( expected: 202, appointment.getUserID());
        Assertions.assertEquals( expected: 303, appointment.getContactID());
    }

    @Test
    void testSendReminder() {
        // Capture console output
        appointment.sendReminder();
    }
}
```

```java
import java.time.LocalDateTime;

class VirtualAppointmentTest {
    8 usages
    private VirtualAppointment virtualAppointment;

    @BeforeEach
    void setUp() {
        virtualAppointment = new VirtualAppointment( appointmentID: 2, appointmentTitle: "Online Consultation", appointment
                appointmentLocation: "Zoom", appointmentType: "Virtual Consultation",
                LocalDateTime.of( year: 2025, month: 3, dayOfMonth: 11, hour: 16, minute: 0),
                LocalDateTime.of( year: 2025, month: 3, dayOfMonth: 11, hour: 17, minute: 0),
                meetingLink: "https://zoom.us/example");
    }

    @Test
    void testVirtualAppointmentCreation() {
        Assertions.assertEquals( expected: 2, virtualAppointment.getAppointmentID());
        Assertions.assertEquals( expected: "Online Consultation", virtualAppointment.getAppointmentTitle());
        Assertions.assertEquals( expected: "Virtual grooming tips", virtualAppointment.getAppointmentDescription());
        Assertions.assertEquals( expected: "Zoom", virtualAppointment.getAppointmentLocation());
        Assertions.assertEquals( expected: "Virtual Consultation", virtualAppointment.getAppointmentType());
        Assertions.assertEquals( expected: "https://zoom.us/example", virtualAppointment.getMeetingLink());
    }

    @Test
    void testSendReminder() { virtualAppointment.sendReminder(); }
}
```

This is a screenshot of the test code described above. It is located in the AppointmentsTest and VirtualAppointment classes respectively. It utilizes import statements to import the JUnit testing library, and JavaDateTime to validate time information for interface with the client schedule database.



Results

"Reminder: Your appointment (Grooming Session) is scheduled at 2025-03-10T14:30

Process finished with exit code 0" indicates that tests have passed.

C4. Source Code

Source code is included in the EvanSmithC868 zip archive submitted.

User Guide

User guide is submitted "User Guide for TechGopher Scheduler.pdf" also located in the EvanSmithC868 zip archive.