Module Name: Advanced Programming

Module Code: CIT3009

School: Computing and Information Technology

Tutor: Christopher Panther

Assignment: Group Project

Group Members:

Jordan Bogle ID#1704233

Alanzo Harris ID#2210195

Shamori Henry ID#2100438

Elijah Muhammad ID#2005032

Kaydeen Walker ID#1902433

Due: April 6, 2025

PROJECT SYNOPSIS

The Event Scheduling System is a Java-based application designed to assist Java Entertainment, a stage equipment rental business, in managing and scheduling their assets efficiently. With the high demand for staging, lighting, power and sound equipment during the Jamaican Spring Break season, the company faces challenges in organizing and tracking their rentals. This software aims to streamline their operations by integrating key modules for asset management, scheduling, billing and invoice summary reporting.

Project Modules:

- **Asset Management:** Keeps track of all equipment in inventory, ensuring proper categorization and availability.
- **Scheduling:** Prevents double-booking by efficiently allocating equipment to upcoming events.
- **Billing:** Generates invoices and receipts for bookings and rentals.
- **Invoice Summary Reporting:** Provides consolidated reports summarizing invoice data for a specified period, including details like total invoices, paid invoices, total amounts, and payment status.

Technical Overview:

The system follows a Client/Server architecture, where the database resides on a central server and clients interact with it over a network. A graphical user interface (GUI) facilitates data entry and management across all modules. The database design incorporates Entity-Relationship Diagrams (ERDs), class diagrams and adheres to third normal form (3NF) normalization to ensure efficiency and accuracy.

This project provides an intuitive, scalable and well-structured solution to enhance Java Entertainment's scheduling and rental process, ensuring smooth operations during peak event seasons.

TECHNOLOGY STACK

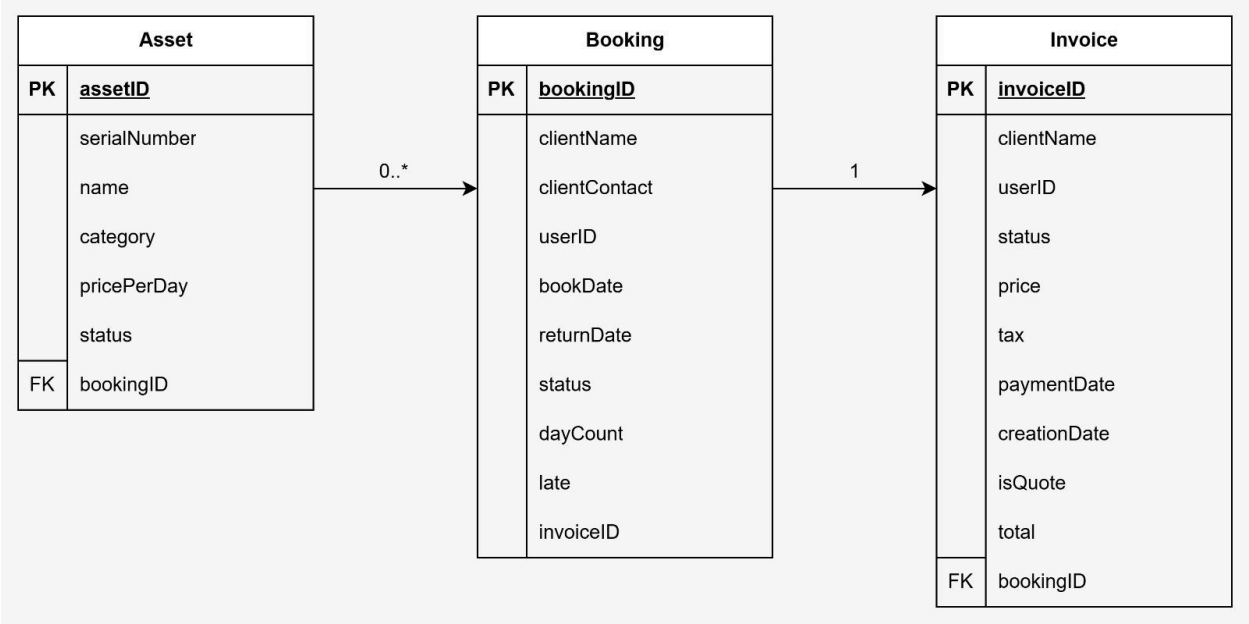
- **Java Swing:** Used to build the **Graphical User Interface (GUI)**, providing an intuitive and user-friendly platform for the client side to manage the equipment, bookings and billing.
- **MySQL:** The **database management system** used to store all data related to assets, bookings, invoices, payments and user details. It supports relational data structures and relationships between various entities (e.g. assets linked to bookings).
- **Hibernate**: An Object-Relational Mapping (ORM) framework used to handle all database interactions in place of raw JDBC. Hibernate allows for seamless mapping between Java objects and MySQL database tables, simplifying data persistence, reducing boilerplate code, and supporting advanced query capabilities.
- **Client-Server Communication:** The client and server communicate over a network. The client sends queries to the server and the server processes these requests, interacting with the MySQL database through Maven and returns results to the client.

This technology stack ensures the system is **efficient**, **scalable** and **user-friendly**, providing **real-time interaction** between the user and the underlying database while maintaining consistency and integrity in data management.

ENTITY DESIGN

Entity Relationship Diagram-ERD

The **Entity Relationship Diagram (ERD)** for the Event Scheduling System illustrates the relationships between key entities such as asset, booking and invoice ensuring efficient data management and preventing conflicts like double bookings.



DATABASE RELATIONS

The Entity Relationship Diagram (ERD) for the Event Scheduling System visualizes the core components of the system Asset, Booking and Invoice and how they interact to support smooth scheduling, billing and resource allocation for stage equipment rentals.

1. Asset Entity

This entity stores detailed information about each rentable item:

- Primary Key: assetID
- Attributes: serialNumber, name, category, pricePerDay, status
- Foreign Key: bookingID – links each asset to a booking, allowing multiple assets to be booked under one transaction (0..* relationship to Booking).

2. Booking Entity

This central entity tracks all reservations:

- Primary Key: bookingID
- Attributes: clientName, clientContact, userID, bookDate, returnDate, status, dayCount, late, invoiceID
- It connects with:
    - Asset: via bookingID, allowing one booking to include multiple assets.
    - Invoice: with a 1-to-1 relationship (each booking generates one invoice).

3. Invoice Entity

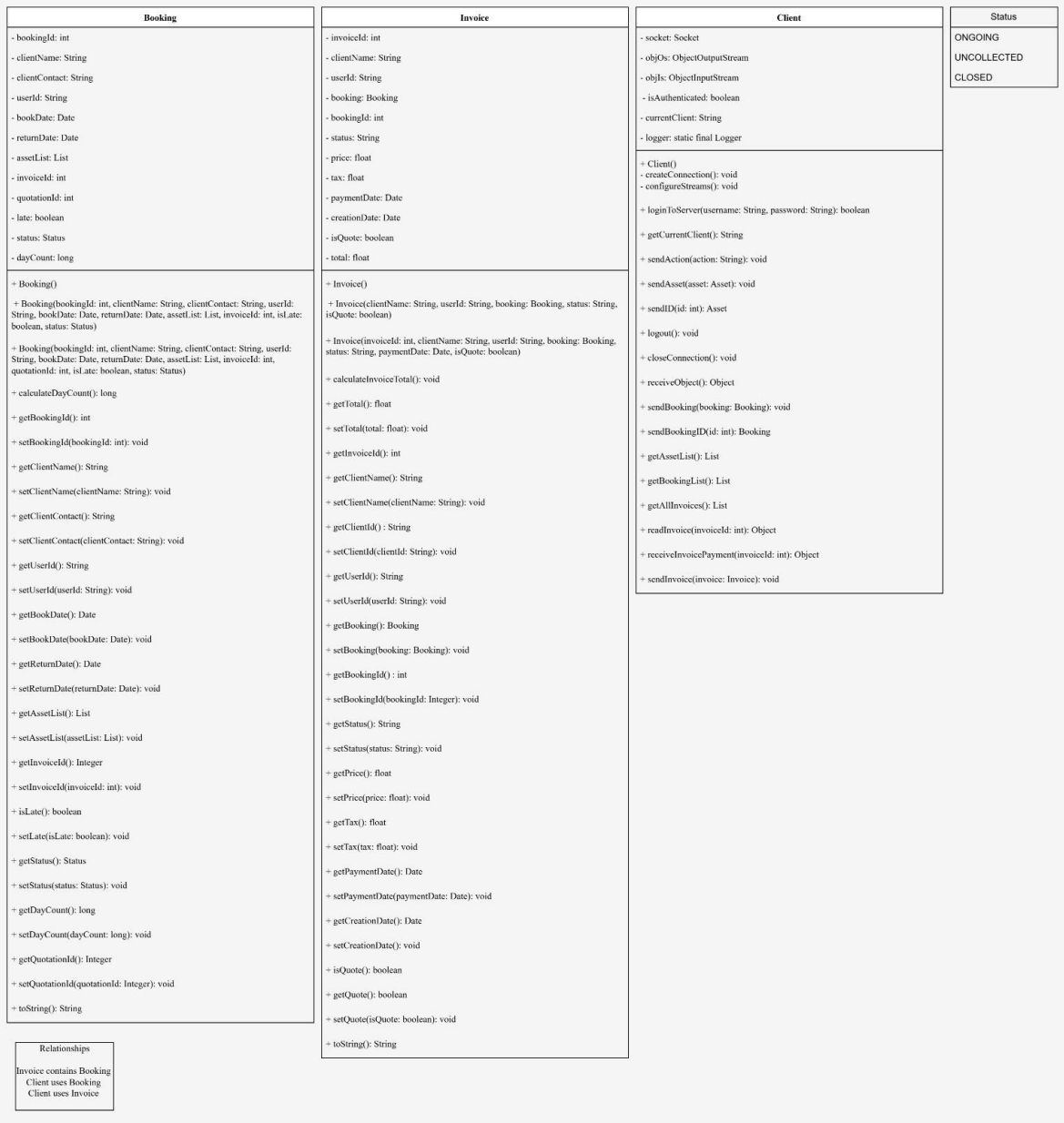
Manages payment records and quotes:

- Primary Key: invoiceID
- Attributes: clientName, userID, status, price, tax, paymentDate, creationDate, isQuote, total
- Foreign Key: bookingID – ties the invoice back to its respective booking.

CLASS DESIGN

Class Diagram

The **Class Diagram** for the Event Scheduling System visually represents the structure of the application by detailing the classes, their attributes, methods and relationships providing a blueprint for the system's functionality and how different modules interact with each other.



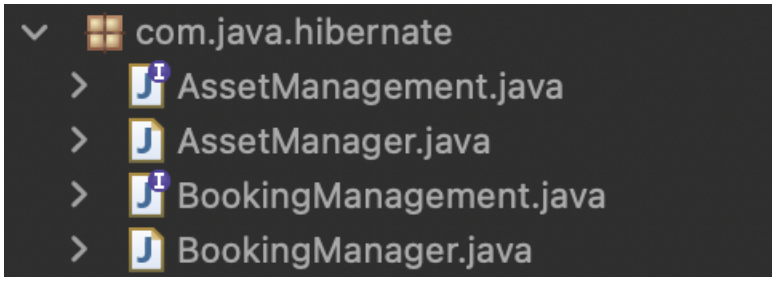
OBJECT MODEL

Abstract Classes

```java
package com.java.hibernate;

import java.util.List;
import com.java.domain.Invoice;

public abstract class InvoiceManagement {

    public abstract void exit();

    public abstract Invoice readInvoice(int invoiceId);

    public abstract void recievePayment(int invoice);

    public abstract List<?> showAllInvoices();

    public abstract void deleteInvoice(int invoiceId);

    public abstract void exportInvoice(int invoiceId, String filePath);

}
```

In the code snippet above an abstract class named InvoiceManagement is declared. Abstract classes serve as blueprints for other classes, outlining a common interface without providing a complete implementation for all methods. Here, InvoiceManagement defines several abstract methods like exit(), readInvoice(), recievePayment(), showAllInvoices(), deleteInvoice(), and exportInvoice(). These methods declare the operations that any concrete class implementing the InvoiceManagement contract must provide. By using an abstract class, the code establishes a clear structure and ensures that all invoice management implementations adhere to a specific set of functionalities, promoting consistency **and allowing for polymorphism.** The showAllInvoices() method also demonstrates the use of generics within this abstract context, indicating that concrete implementations will return a list of some specific type related to invoices.

Interfaces



The image above shows two Java interfaces within the com.java.hibernate package, denoted by the "I" icon: AssetManagement and BookingManagement. These interfaces define contracts for managing assets and bookings respectively, outlining the methods that implementing classes must provide. This design promotes modularity and allows for different concrete implementations of these core functionalities, as suggested by the presence of corresponding manager classes (e.g., AssetManager).

Eg. AssetManagement() interface:

```java
package com.java.hibernate;

import java.util.List;

import com.java.domain.Asset;

public interface AssetManagement {

    public abstract void exit();

    public abstract void create(Asset asset);

    public abstract Asset read(int assetId);

    public abstract void update(Asset asset);

    public abstract void delete(int assetId);

    public abstract List<?> showAll();

}
```

Polymorphism

Eg: the receiveobject() method:

```java
/**
 * Receives an object from the server.
 *
 * @return The object received from the server.
 */
public Object receiveObject() {
    try {
        Object receivedObject = objIs.readObject();  // Read and return the object sent by the server
        logger.info("Client {} received object: {}", currentClient, receivedObject);  // Log the received object
        return receivedObject;
    } catch (IOException | ClassNotFoundException e) {
        System.out.println("Error receiving object: " + e.getMessage());
        logger.error("Error receiving object for client {}: {}", currentClient, e.getMessage(), e);  // Log error with client info
        e.printStackTrace();
        return null;  // Return null if an error occurs
    }
}
```

The method returns a value of type Object, the **superclass of all classes in Java**.

This means it can receive and return any subclass object such as a Booking, Asset, Invoice. **without knowing the exact type at compile time**.

This is **runtime polymorphism**, where the actual object's type is determined when the program runs, not when it is written.

Inheritance

Eg: in the GUI classes:

```java
public class AssetGui extends JPanel {
    private static final long serialVersionUID = 1L;
    private static final Logger logger = LogManager.getLogger(AssetGui.class);

    private JTable assetTable;
    private JButton refreshButton, newAssetButton, deleteAssetButton;
    private JRadioButton availableAssets, bookedAssets;
    private ButtonGroup filterGroup;
    private JTextField searchField;
    private JButton searchButton;

    public AssetGui() {
        initializeUI();
```

The code snippet demonstrates inheritance through the declaration **public class AssetGui extends JPanel.** This signifies that the AssetGui class is a subclass or derived class that inherits properties and behaviors from the JPanelclass, which is its superclass or base class. By extending JPanel, AssetGui automatically gains all the functionalities of a standard panel in Swing, such as the ability to contain and manage other UI components, handle layout, and be displayed within a window. The AssetGui class then adds its own specific UI elements and logic related to asset

management (like the assetTable, buttons for actions, and filtering options), building upon the foundational capabilities provided by JPanel without needing to reimplement them. This reuse of code and hierarchical organization is a key benefit of inheritance in object-oriented programming.

Eg: in the InvoiceManager class:

```
package com.java.hibernate;

import java.sql.Date;

public class InvoiceManager extends InvoiceManagement{

    private static final Logger logger = LogManager.getLogger(InvoiceManager.class);
    private static final SessionFactory sessionFactory = buildSessionFactory(); // Singleton

    // Build session factory using Hibernate utility
    private static SessionFactory buildSessionFactory() {
        try {
            logger.info("Building session factory...");
            return new Configuration().configure().buildSessionFactory();
        } catch (Exception e) {
            logger.error("SessionFactory creation failed: {}", e.getMessage(), e);
            System.err.println("SessionFactory creation failed: " + e);
            throw new ExceptionInInitializerError(e);
        }
    }
```

Another clear example of inheritance in this project is the relationship between the InvoiceManager class and the abstract superclass InvoiceManagement. By using the extends keyword, InvoiceManager inherits all accessible methods and properties defined in InvoiceManagement, allowing for reuse of common invoice-related functionalities such as validation, processing or data formatting. This object-oriented design approach promotes modularity and maintainability by centralizing shared logic in a single abstract class, while enabling subclasses like InvoiceManager to implement or override specific behaviors as needed. Additionally, this use of inheritance supports polymorphism, allowing instances of InvoiceManager to be treated as instances of InvoiceManagement when required such as when interfacing with other components expecting the abstract type. Overall, this inheritance strategy simplifies code management and enhances the scalability of the application.

USER EXPERIENCE

The Java Entertainment System's user interface is constructed using standard **Java Swing components, including JPanel, JFrame and JTable.** JFrame serves as the main window, providing the application's frame. JPanel is used as a container within the JFrame to organize and group various UI elements, such as buttons, text fields, and tables. JTable is employed to display data in a structured, tabular format, enhancing readability and data management, as seen in the asset and booking screens.

The Java Entertainment System manual emphasizes the use of standard GUI components, parent windows, menus, message dialogs, and a consistent user experience (UX). The application adheres to these principles by utilizing familiar Swing components, ensuring ease of use for staff members. Parent windows, like the main application window (JFrame), organize the various screens (e.g., Assets, Bookings, Invoices). Message dialogs are consistently used to provide

feedback to the user, such as login confirmations or booking creation success messages, contributing to clear communication and a positive UX.

The consistent UX is further supported by the application's navigation and layout. The left-hand side menu provides easy access to different sections, and the screens maintain a similar structure with tables for data display and buttons for actions. The use of clear labels, input fields, and confirmation prompts contributes to an intuitive workflow. Overall, the UI design prioritizes functionality and usability, enabling staff to efficiently manage equipment rentals.

CLIENT/SERVER ARCHITECTURE

The Event Scheduling System is built upon a Client/Server Architecture, effectively separating the user interface and data management responsibilities. The client side provides the user with a graphical interface to interact with the system, enabling actions such as scheduling equipment, generating invoices, and viewing booking details. These client-side actions generate requests that are sent across a network to the server. The server side is responsible for processing these requests, managing and maintaining the system's data. This includes storing and retrieving information related to assets, bookings, and invoices, ensuring data consistency, and preventing conflicts like double bookings.

Communication between the client and server is achieved through the implementation of TCP/IP sockets for networking. This robust and reliable communication protocol ensures that client requests are accurately transmitted to the server and that the server's responses are correctly received by the client. The client utilizes JDBC (Java Database Connectivity) to interact with the MySQL database, sending queries and updates via the server. This architecture allows for real-time updates and seamless interaction for users, as changes made on the client side are reflected in the database through the server's processing.

The user manual provides glimpses of this architecture in action. For example, the "Server Side Interface" section details how IT staff can start and monitor the Java Rental Server, observing log messages and server status. This highlights the server's crucial role in maintaining the system. The client-side perspective is evident in the various screen descriptions, where user actions trigger data requests and updates that are handled by the server, showcasing the client-server interaction facilitated by TCP/IP sockets and JDBC.

EXCEPTION HANDLING/LOGGING

Example showing exception handling

```java
@Override
public void create(Asset asset) {
    logger.info("Creating asset: {}", asset);
    System.out.println("Creating asset: " + asset);
    Session session = sessionFactory.openSession();
    session.beginTransaction();
    try {
        session.persist(asset);
        session.getTransaction().commit();
        logger.info("Asset created successfully: {}", asset.getAssetId());
        System.out.println("Asset created successfully: " + asset.getAssetId());
    } catch (Exception e) {
        logger.error("Error creating asset: {}", e.getMessage(), e);
        System.err.println("Error creating asset: " + e.getMessage());
        if (session.getTransaction() != null) {
            session.getTransaction().rollback();
        }
    } finally {
        session.close();
    }
}
```

The system employs exception handling within the create_asset method to manage potential errors that may occur during the process of creating a new asset. This method utilizes the Session object obtained from the sessionFactory to open a database transaction and persist the new asset. The try block encapsulates the code that performs the database operation, while the catch block handles any exceptions that might arise during this process. If an exception occurs, the catch block logs the error message using the Logger object and optionally prints the error message to the console. The finally block ensures that the database session is closed, regardless of whether an exception occurred. This exception handling approach guarantees that the system remains robust and can recover from potential errors during asset creation.

Example showing Logging:

```
2025-04-06 01:36:45 [main] INFO  com.java.hibernate.AssetManager - Building session factory...
2025-04-06 01:36:45 [main] WARN  org.hibernate.engine.jdbc.connections.internal.ConnectionProviderInitiatc
2025-04-06 01:36:45 [main] WARN  org.hibernate.orm.connections.pooling - HHH10001002: Using built-in conne
2025-04-06 01:36:46 [main] WARN  org.hibernate.engine.jdbc.env.internal.JdbcEnvironmentInitiator - HHH0003
java.lang.IllegalStateException: Cannot get a connection as the driver manager is not properly initializec
    at org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl.getConnection(Dr
    at org.hibernate.engine.jdbc.env.internal.JdbcEnvironmentInitiator$ConnectionProviderJdbcConnectionAcc
    at org.hibernate.resource.transaction.backend.jdbc.internal.JdbcIsolationDelegate.delegateWork(JdbcIsc
    at org.hibernate.engine.jdbc.env.internal.JdbcEnvironmentInitiator.getJdbcEnvironmentUsingJdbcMetadata
    at org.hibernate.engine.jdbc.env.internal.JdbcEnvironmentInitiator.initiateService(JdbcEnvironmentInit
    at org.hibernate.engine.jdbc.env.internal.JdbcEnvironmentInitiator.initiateService(JdbcEnvironmentInit
    at org.hibernate.boot.registry.internal.StandardServiceRegistryImpl.initiateService(StandardServiceReg
    at org.hibernate.service.internal.AbstractServiceRegistryImpl.createService(AbstractServiceRegistryImp
    at org.hibernate.service.internal.AbstractServiceRegistryImpl.initializeService(AbstractServiceRegistr
    at org.hibernate.service.internal.AbstractServiceRegistryImpl.getService(AbstractServiceRegistryImpl.j
    at org.hibernate.engine.jdbc.internal.JdbcServicesImpl.configure(JdbcServicesImpl.java:52) [hibernate-
    at org.hibernate.boot.registry.internal.StandardServiceRegistryImpl.configureService(StandardServiceRe
    at org.hibernate.service.internal.AbstractServiceRegistryImpl.initializeService(AbstractServiceRegistr
    at org.hibernate.service.internal.AbstractServiceRegistryImpl.getService(AbstractServiceRegistryImpl.j
    at org.hibernate.engine.jdbc.connections.internal.BasicConnectionCreator.convertSqlException(BasicConr
```

The provided log excerpt from the Java Entertainment System demonstrates the use of logging for monitoring and debugging the application. The log entries, prefixed with timestamps and log levels (e.g., INFO, WARN), provide a chronological record of events occurring within the system. For instance, the INFO message indicates the successful building of the session factory by com.java.hibernate.AssetManager, a crucial step in the Hibernate framework for database interaction. Subsequently, WARN messages highlight potential configuration or connection-related issues within the JDBC environment. Critically, the java.lang.IllegalStateException: Cannot get a connection... entry signifies a severe error preventing the application from establishing a database connection. This detailed logging, as also mentioned in the user manual regarding server-side logs, is essential for IT staff to diagnose problems, understand the system's behavior, and ensure its smooth operation by providing insights into both normal activities and critical failures.

DATABASE CONNECTIVITY

Originally we had planned to work with the database with JDBC by using string SQL statements, which are the standard way to perform relational methods, to create all the necessary tables with FK constraints.

DatabaseSchema.sql

C: > Users > jorda > OneDrive > Desktop > AP PROJECT > DatabaseSchema.sql

```
17  -- Create the Bookings table
18  CREATE TABLE IF NOT EXISTS Bookings (
19      booking_id INT PRIMARY KEY NOT NULL,
20      clientId INT,
21      bookingDate DATE NOT NULL,
22      returnDate DATE NOT NULL,
23      user_id INT,
24      FOREIGN KEY (clientId) REFERENCES Clients(clientId),
25      FOREIGN KEY (user_id) REFERENCES Users(user_id) ON DELETE SET NULL
26  );
27
28  CREATE TABLE IF NOT EXISTS Assets (
29      asset_id INT PRIMARY KEY NOT NULL,
30      name VARCHAR(100) NOT NULL,
31      category VARCHAR(50) NOT NULL,
32      status VARCHAR(50) NOT NULL,
33      location VARCHAR(100) NOT NULL
34
35  );
36  -- Create the AssetInstances table
37  CREATE TABLE IF NOT EXISTS AssetInstances (
38      serial_number VARCHAR(100) PRIMARY KEY NOT NULL,
39      asset_id INT NOT NULL,
40      status ENUM('Available', 'Booked', 'Maintenance') DEFAULT 'Available',
41      booking_id INT DEFAULT NULL,
42      FOREIGN KEY (asset_id) REFERENCES Assets(asset_id) ON DELETE CASCADE,
43      FOREIGN KEY (booking_id) REFERENCES Bookings(booking_id) ON DELETE SET NULL
44  );
45
46  -- Create the Invoices table
47  CREATE TABLE IF NOT EXISTS Invoices (
48      invoice_id INT PRIMARY KEY AUTO_INCREMENT,
49      client_id INT,
50      booking_id INT,
51      user_id INT,
52      amount DECIMAL(10,2) NOT NULL,
```

However, we learned how to use Hibernate, an Object-Relational Mapping (ORM) framework, with Maven to interact with our database at lab class, and the group collectively decided to use Hibernate to create all the tables for our domain classes. We also created one more table for a many-to-one relationship for Assets and Bookings. Below is the ‘hibernate.cfg.xml’ file which connects to the database and maps domain classes to the tables, allowing us to work with objects instead of writing raw SQL.

```
-//Hibernate/Hibernate Configuration DTD 3.0//EN (doctype)
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
    "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <!-- Database connection settings -->
        <property name="connection.driver_class">com.mysql.cj.jdbc.Driver</property>
        <property name="connection.url">jdbc:mysql://127.0.0.1:3307/hibernateeventscheduler</property>
        <property name="connection.username">root</property>
        <property name="connection.password">usbw</property>

        <!-- Choose the Hibernate dialect for the specific database type -->
       <property name="hibernate.dialect">org.hibernate.dialect.MySQL8Dialect</property>

         <!-- Automatically Create Tables If Not Exist -->
        <property name="hibernate.hbm2ddl.auto">update</property>

        <property name="hibernate.hikari.maximumPoolSize">10</property>

        <property name="show_sql">true</property>

        <mapping class="com.java.domain.Asset"/>
        <mapping class="com.java.domain.Booking"/>
        <mapping class="com.java.domain.Invoice"/>
    </session-factory>
</hibernate-configuration>
```

GENERICS AND COLLECTIONS

Example showing Generics:

```
    @Override
    public List<Asset> showAll() {
        logger.info("Showing all assets...");
        System.out.println("SHOWING ASSETS...");
        try (Session session = sessionFactory.openSession()) {
            List<Asset> assets = session.createQuery("FROM Asset", Asset.class).list();
            logger.info("Total assets found: {}", assets.size());
            System.out.println("Total assets found: " + assets.size());
            return assets;
        } catch (Exception e) {
            logger.error("Error retrieving all assets: {}", e.getMessage(), e);
            System.err.println("Error retrieving all assets: " + e.getMessage());
            return null;
        }
    }
}
```

Generics are employed for example in the **showAll()** method. The declaration List<Asset> explicitly defines that the assets variable will be a list specifically containing Asset objects, ensuring type safety at compile time. Furthermore, the Hibernate **createQuery()** method utilizes generics by accepting Asset.class as an argument, instructing it to return a list (List<Asset>) where each element is guaranteed to be an Asset instance. This eliminates the need for manual casting when processing the retrieved list of assets.

Example showing Collections:

```
    public void createBooking(Booking booking) {
        logger.info("Creating booking: {}", booking);
        System.out.println("Creating booking: " + booking);
        Session session = sessionFactory.openSession();
        session.beginTransaction();

        try {
            // Merge booking to attach it to the session context
            booking = session.merge(booking);

            List<Asset> validAssets = new ArrayList<>();
            List<Asset> alreadyBookedAssets = new ArrayList<>();

            // Validate and update each asset
            for (Asset asset : booking.getAssetList()) {
                asset = session.merge(asset); // Attach asset to session

                // Asset must not be booked and must be available
                if (asset.getBookingId() == null && asset.getStatus().equals(Asset.Status.AVAILABLE)) {
                    asset.setBooking(booking); // Set reference to booking
                    asset.setBookingId(booking.getBookingId()); // Assign booking ID
                    asset.setStatus(Asset.Status.BOOKED); // Mark as booked
                    validAssets.add(asset); // Track asset to persist
                } else {
                    alreadyBookedAssets.add(asset); // Collect already booked assets
                }
            }
```

In this snippet from the **createBooking** method, collections, specifically java.util.List implemented as ArrayList, are used to manage the assets associated with a booking. Two lists, validAssets and alreadyBookedAssets are created to categorize the assets based on their availability. The code iterates through the booking.getAssetList(), which is also likely a collection, to validate each asset. Available assets are added to validAssets after being marked as booked and linked to the current booking, while unavailable assets are added to alreadyBookedAssets. This use of collections allows for efficient processing and categorization of multiple assets within the booking creation process.

THREADING

```
public static void startServer() {
    new Thread(() -> {
        try {
            new Configuration()
                    .configure("hibernate.cfg.xml")
                    .addAnnotatedClass(com.java.domain.Asset.class)
                    .buildSessionFactory();

            try (ServerSocket serverSocket = new ServerSocket(PORT)) {
                logger.info("Server started. Waiting for clients...");
                while (true) {
                    try {
                        Socket socket = serverSocket.accept();
                        new ClientHandler(socket).start();
                    } catch (IOException e) {
                        logger.error("Error accepting client connection: " + e.getMessage(), e);
                    }
                }
            }
        } catch (Exception e) {
            logger.error("Error starting the server: " + e.getMessage(), e);
        }
    }).start(); // start in a new thread so GUI doesn't freeze
}
```

The startServer method in the Java Entertainment System implements multithreading to prevent the Graphical User Interface (GUI) from freezing while the server is running and listening for client connections. This is achieved by creating a new Thread that encapsulates the server's core logic. Inside this thread's run method (implicitly defined using a lambda expression), the server initializes Hibernate, establishes a ServerSocket on a specified PORT, and enters an infinite while loop to continuously accept incoming client connections using serverSocket.accept(). For each accepted client connection, a new ClientHandler thread is created and started to manage the communication with that specific client. This approach ensures that the main GUI thread remains responsive to user interactions, while the server operations, including listening for and handling multiple client connections concurrently, are performed in separate background threads. The user manual indirectly refers to this when it mentions starting the "Java Rental Server" and observing its status and logs, implying a background process that doesn't block the UI.

FUTURE ENHANCEMENTS

The current version of the Java Entertainment System represents our **Minimum Viable Product (MVP).** An MVP is a development strategy focused on releasing a product with only the core features necessary to address the immediate needs of the users. In this initial phase, the system primarily facilitates the management of event equipment rentals, enabling staff to track inventory, schedule bookings, handle basic invoicing, print invoices and generate an invoice summary report.

However, we recognize the potential for significant expansion to provide a more comprehensive solution for Java Entertainment. Future enhancements under consideration include:

- **Customer Relationship Management (CRM):** The system could be enhanced to store detailed information about customers. This would go beyond simply recording a client's name for a booking and involve features to manage customer profiles, including contact information, booking history, preferences and communication logs. This would enable staff to provide more personalized service, track customer interactions and improve customer retention.
- **Human Resources Management (HRM):** Functionality could be added to store and manage staff information. This might include employee contact details, roles, responsibilities, schedules and permissions within the system. Implementing HRM features would streamline staff administration, improve workforce organization and enhance internal communication.

These planned enhancements reflect our commitment to evolving the Java Entertainment System from a core equipment rental tool to a more integrated platform that supports a broader range of business functions.

SOFTWARE REQUIREMENTS

1. **Java Development Kit (JDK)**

- **Version:** JDK 17 or higher (recommended for modern Java features and long-term support)
- **Purpose:** Required for compiling and running the Java application (both client and server sides).

2. **Log4j2: API & Core**

- **Version:** 2.20.0
- **Purpose:** Provides a powerful and flexible logging framework for debugging, monitoring, and auditing both client and server operations.