CRUD for Users: Manage users with attributes like name, password, and roles.

CRUD for Roles: Define roles like cashier, supervisor, and administrator, possibly with different permissions or views.

CRUD for Games: Manage games, including names and age restrictions (e.g., minor, teen, adult).

To implement a basic CRUD (Create, Read, Update, Delete) system in Java for managing users, roles, and games, you'd typically use a combination of a SQL-based database like MySQL for data storage and a Java application for the logic and user interface. Below is a general guide on how you could structure this system with a focus on the database and Java side.

**Database Setup:**

1. **Users Table:**
   * **UserID** (Primary Key)
   * **Name**
   * **Password** (Usually stored as a hash)
   * **RoleID** (Foreign Key from Roles Table)
2. **Roles Table:**
   * **RoleID** (Primary Key)
   * **RoleName** (e.g., cashier, supervisor, administrator)
3. **Games Table:**
   * **GameID** (Primary Key)
   * **GameName**
   * **AgeRestriction** (e.g., minor, teen, adult)

**SQL Sample Table Creation:**

Here are the SQL commands to create these tables in a MySQL database:

CREATE TABLE Roles (

RoleID INT AUTO\_INCREMENT PRIMARY KEY,

RoleName VARCHAR(255) NOT NULL

);

CREATE TABLE Users (

UserID INT AUTO\_INCREMENT PRIMARY KEY,

Name VARCHAR(255) NOT NULL,

Password VARCHAR(255) NOT NULL, -- Consider security best practices for storing passwords

RoleID INT,

FOREIGN KEY (RoleID) REFERENCES Roles(RoleID)

);

CREATE TABLE Games (

GameID INT AUTO\_INCREMENT PRIMARY KEY,

GameName VARCHAR(255) NOT NULL,

AgeRestriction ENUM('minor', 'teen', 'adult') NOT NULL

);

**Java Implementation:**

For Java, you would create classes corresponding to each entity: **User**, **Role**, and **Game**. These classes would include methods for CRUD operations. Here's a very simplified structure of what those classes might look like:

1. **User Class:**

javaCopy code

public class User { private int userID; private String name; private String password; // Consider how you'll handle password security private int roleID; // Constructors, getters, setters, and methods for CRUD operations }

1. **Role Class:**

javaCopy code

public class Role { private int roleID; private String roleName; // Constructors, getters, setters, and methods for CRUD operations }

1. **Game Class:**

javaCopy code

public class Game { private int gameID; private String gameName; private String ageRestriction; // Constructors, getters, setters, and methods for CRUD operations }

**Implementing CRUD Operations:**

For each class, implement methods to handle the four basic CRUD operations. These methods will interact with the database using JDBC to run the appropriate SQL commands.

* **Create**: Insert new records.
* **Read**: Retrieve existing records.
* **Update**: Modify existing records.
* **Delete**: Remove records.

For example, the **User** class might have these methods:

* **createUser()**
* **getUser(int userID)**
* **updateUser(User user)**
* **deleteUser(int userID)**

**Java and JDBC:**

To connect to your MySQL database and perform these operations, use JDBC with prepared statements to ensure security against SQL injection.

1. **Database Connection:** Establish a connection to your MySQL database.
2. **Prepared Statements:** Use prepared statements for all database interactions.
3. **Exception Handling:** Properly handle SQL exceptions.

**Security Considerations:**

* **Password Storage**: Never store plain-text passwords. Use a strong hashing algorithm like bcrypt, and consider including salt.
* **SQL Injection**: Always use prepared statements or other means to protect against SQL injection.
* **User Input Validation**: Always validate and sanitize user input, especially before interacting with the database.

This setup gives you a foundational structure. However, building a robust, secure, and efficient system will involve deeper considerations for architecture, security, error handling, and user interface design. As you implement, focus on one section at a time, testing thoroughly before moving on to the next to ensure each part of your system works as expected.

**Security Best Practices:**

* **HTTPS**: Always use HTTPS to protect user credentials and session data in transit.
* **Session Management**: Use secure, server-side sessions and consider token-based authentication (like JWT) for stateless authentication.
* **Input Validation**: Always validate and sanitize inputs to prevent injection attacks.
* **Password Security**: Use strong, one-way hashing algorithms like BCrypt for storing passwords. Never store plain text passwords.
* **Error Handling**: Don't reveal sensitive information in error messages. Log the details for internal use and provide generic error messages to users.
* **Regularly Update Dependencies**: Keep your security frameworks and libraries up to date.

By extending the **UserService** with authentication and authorization methods, you can provide a more secure application that ensures users are who they say they are and can only do what they're allowed to do. This setup is a basic example. Real-world applications often require more sophisticated security measures, including multi-factor authentication, more granular permissions, and regular security audits. Consider these aspects according to the specific needs and risks associated with your application.

CREATE TABLE IF NOT EXISTS Tickets ( TicketID INT AUTO\_INCREMENT PRIMARY KEY, GameID INT,

GamePricingID INT,

UserID INT,

ValidDate DATETIME,

Barcode VARCHAR(255),

FOREIGN KEY (GameID) REFERENCES Games(GameID) ,

FOREIGN KEY (GamePricingID) REFERENCES GamePricing(PricingID),

FOREIGN KEY (UserID) REFERENCES Users(UserID) );

<?php

$servername = "localhost";

$username = "username"; // Remplacez par votre nom d'utilisateur pour la base de données

$password = "password"; // Remplacez par votre mot de passe pour la base de données

$dbname = "myDB"; // Remplacez par le nom de votre base de données

// Création de la connexion

$conn = new mysqli($servername, $username, $password, $dbname);

// Vérification de la connexion

if ($conn->connect\_error) {

die("Connection failed: " . $conn->connect\_error);

}

$sql = "SELECT \* FROM Tickets";

$result = $conn->query($sql);

if ($result->num\_rows > 0) {

// Affichage des données de chaque ligne

while($row = $result->fetch\_assoc()) {

echo "TicketID: " . $row["TicketID"]. " - GameID: " . $row["GameID"]. " - GamePricingID: " . $row["GamePricingID"]. " - UserID: " . $row["UserID"]. " - ValidDate: " . $row["ValidDate"]. " - Barcode: " . $row["Barcode"]. "<br>";

}

} else {

echo "0 results";

}

$conn->close();

?>

1. **Nombre de Tickets pour le Jour en Cours** : Utilisez une requête SQL qui filtre les tickets ayant une **ValidDate** qui correspond à la date actuelle.
2. **Nombre de Tickets pour la Dernière Heure** : Utilisez une requête SQL qui filtre les tickets dont la **ValidDate** est dans l'heure précédant l'heure actuelle.

Voici un exemple de script PHP pour afficher ces informations :

<?php

$servername = "localhost";

$username = "username"; // Remplacez par votre nom d'utilisateur

$password = "password"; // Remplacez par votre mot de passe

$dbname = "myDB"; // Remplacez par le nom de votre base de données

try {

$conn = new PDO("mysql:host=$servername;dbname=$dbname", $username, $password);

$conn->setAttribute(PDO::ATTR\_ERRMODE, PDO::ERRMODE\_EXCEPTION);

// Nombre de tickets pour le jour en cours

$stmt1 = $conn->prepare("SELECT COUNT(\*) AS total\_today FROM Tickets WHERE DATE(ValidDate) = CURDATE()");

$stmt1->execute();

$result1 = $stmt1->fetch(PDO::FETCH\_ASSOC);

$totalToday = $result1['total\_today'];

// Nombre de tickets pour la dernière heure

$stmt2 = $conn->prepare("SELECT COUNT(\*) AS total\_last\_hour FROM Tickets WHERE ValidDate >= NOW() - INTERVAL 1 HOUR");

$stmt2->execute();

$result2 = $stmt2->fetch(PDO::FETCH\_ASSOC);

$totalLastHour = $result2['total\_last\_hour'];

// Affichage des résultats

echo "<div>Nombre de Tickets pour le jour en cours : $totalToday</div>";

echo "<div>Nombre de Tickets pour la dernière heure : $totalLastHour</div>";

} catch(PDOException $e) {

echo "Erreur : " . $e->getMessage();

}

$conn = null;

?>

Pour répondre à votre requête, nous devons formuler plusieurs requêtes SQL qui effectuent les calculs nécessaires à partir des tables **Users**, **GamePricing**, **Games**, et **Tickets**. Ces requêtes vont calculer :

1. **Le nombre total de tickets par utilisateur** pour le dernier jour, la dernière heure, et en total.
2. **Le total des prix par utilisateur** pour le dernier jour et la dernière heure.
3. **Le nombre de tickets par jeu** pour le dernier jour, la dernière heure, et en total.

Cela implique d'utiliser des fonctions de date et d'heure de SQL pour filtrer les enregistrements, ainsi que des jointures pour combiner les données de différentes tables. Voici des exemples de requêtes pour chacun de ces calculs :

**1. Nombre total de tickets par utilisateur**

-- Dernier jour

SELECT UserID, COUNT(\*) AS TotalTickets

FROM Tickets

WHERE DATE(ValidDate) = CURDATE() - INTERVAL 1 DAY

GROUP BY UserID;

-- Dernière heure

SELECT UserID, COUNT(\*) AS TotalTickets

FROM Tickets

WHERE ValidDate >= NOW() - INTERVAL 1 HOUR

GROUP BY UserID;

-- Total en général

SELECT UserID, COUNT(\*) AS TotalTickets

FROM Tickets

GROUP BY UserID;

### . Total des prix par utilisateur

Pour cette requête, nous devons lier les tables **Tickets**, **GamePricing**, et **Users**. Supposons que le prix d'un ticket est déterminé par le **GamePricingID** dans la table **Tickets**.

-- Dernier jour

SELECT t.UserID, SUM(gp.Price) AS TotalPrice

FROM Tickets t

JOIN GamePricing gp ON t.GamePricingID = gp.PricingID

WHERE DATE(t.ValidDate) = CURDATE() - INTERVAL 1 DAY

GROUP BY t.UserID;

-- Dernière heure

SELECT t.UserID, SUM(gp.Price) AS TotalPrice

FROM Tickets t

JOIN GamePricing gp ON t.GamePricingID = gp.PricingID

WHERE t.ValidDate >= NOW() - INTERVAL 1 HOUR

GROUP BY t.UserID;

### . Nombre de tickets par jeu

-- Dernier jour

SELECT g.GameID, COUNT(\*) AS TotalTickets

FROM Tickets t

JOIN Games g ON t.GameID = g.GameID

WHERE DATE(t.ValidDate) = CURDATE() - INTERVAL 1 DAY

GROUP BY g.GameID;

-- Dernière heure

SELECT g.GameID, COUNT(\*) AS TotalTickets

FROM Tickets t

JOIN Games g ON t.GameID = g.GameID

WHERE t.ValidDate >= NOW() - INTERVAL 1 HOUR

GROUP BY g.GameID;

-- Total en général

SELECT g.GameID, COUNT(\*) AS TotalTickets

FROM Tickets t

JOIN Games g ON t.GameID = g.GameID

GROUP BY g.GameID;