Java Class Structure (Camera and IP Camera)

1. Java Class Structure (Camera and IP Camera)

- **Camera (Abstract Class)**: Define shared attributes and methods that all camera types will have. For example, methods like startRecording(), stopRecording(), and getStream().

```
public abstract class Camera {
    protected String model;
    protected String ipAddress;

    // Constructor
    public Camera(String model, String ipAddress) {
        this.model = model;
        this.ipAddress = ipAddress;
    }

    // Abstract method to get video stream
    public abstract void getStream();

    // Concrete method
    public void startRecording() {
        System.out.println("Recording started...");
    }

    public void stopRecording() {
        System.out.println("Recording stopped...");
    }
}
```

- **IP Camera (Concrete Class)**: This class extends `Camera` and implements any specific behavior or details related to IP cameras.

```
public class IPCamera extends Camera {

    // Constructor
    public IPCamera(String model, String ipAddress) {
        super(model, ipAddress);
    }

    @Override
    public void getStream() {
        // Logic to connect to the IP camera and fetch the video stream
        System.out.println("Streaming from IP Camera at: " + ipAddress);
    }

    // You can also add IP-camera specific methods here, like authentication, etc.
}
```

2. Web Application (home.jsp and Dashboard)

- You can create a DashboardServlet that interacts with the camera objects (like IPCamera) and pass the relevant data to your JSP pages for rendering.

Example Servlet code:

java
Αντιγραφή κώδικα
```
@WebServlet("/dashboard")
public class DashboardServlet extends HttpServlet {
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
        Camera camera = new IPCamera("Model XYZ", "192.168.1.1");
        request.setAttribute("camera", camera);

        RequestDispatcher dispatcher =
request.getRequestDispatcher("/dashboard.jsp");
        dispatcher.forward(request, response);
    }
}
```

In dashboard.jsp, you can display the camera info or its streaming status:

```
<%@ page import="yourpackage.Camera" %>
<%@ page import="yourpackage.IPCamera" %>
<html>
    <body>
        <h2>Camera Dashboard</h2>
        <p>Camera Model: ${camera.model}</p>
        <p>IP Address: ${camera.ipAddress}</p>
        <p>Status: Streaming from camera...</p>
    </body>
</html>
```

3. Modularity and Extensibility

- For **modularity**, ensure that each class is responsible for a specific task. The `Camera` class handles common logic, and the `IPCamera` class handles specific logic for IP cameras.
- You can add more camera types by creating new classes (like `AnalogCamera` or `WirelessCamera`) that extend `Camera`.

4. SQL Integration

- If you want to store camera details in a database, you can create a `CameraDAO` class that handles database operations like saving or fetching camera data.

```
public class CameraDAO {
    public Camera getCameraDetails(int cameraId) {
        // SQL code to fetch camera details from the database
    }
    public void saveCamera(Camera camera) {
        // SQL code to save camera details to the database
    }
}
```

This approach will ensure that your system remains modular and easily extensible. Let me know if you'd like further details or examples!

It sounds like you're aiming for a hybrid version of `home.jsp` and `dashboard.jsp`, with encapsulation of the `Camera` class (and its extension `IPCamera`) that interacts with the user data via the existing `UserDAO`.

Here's a suggested approach to integrating everything into a single flow:

1. Encapsulation in the Combined Model

You can use the `Camera` and `IPCamera` classes as part of the business logic passed between the `dashboard.jsp` and `home.jsp`. The hybrid approach could involve defining a model or DTO (Data Transfer Object) that encapsulates both user and camera information.

Camera Class (abstract)
java
Αντιγραφή κώδικα
```
public abstract class Camera {
    protected String model;
    protected String ipAddress;

    public Camera(String model, String ipAddress) {
        this.model = model;
        this.ipAddress = ipAddress;
    }

    public abstract void startStream();

    public void stopStream() {
        System.out.println("Stream stopped");
    }

    public String getModel() {
        return model;
    }

    public String getIpAddress() {
        return ipAddress;
    }
}
```

IP Camera Class (concrete)

```
public class IPCamera extends Camera {

    public IPCamera(String model, String ipAddress) {
        super(model, ipAddress);
    }

    @Override
    public void startStream() {
        System.out.println("Streaming from IP Camera at: " + ipAddress);
    }
}
```

2. User and Camera Model Integration

You can create a class that integrates both user and camera information, for example, `DashboardModel`.

```
public class DashboardModel {
    private User user;
    private Camera camera;

    public DashboardModel(User user, Camera camera) {
        this.user = user;
        this.camera = camera;
    }

    public User getUser() {
        return user;
    }

    public Camera getCamera() {
        return camera;
    }
}
```

This model can be populated by the `DashboardServlet` and passed to both `home.jsp` and `dashboard.jsp`.

3. Dashboard Servlet (Controller)

In your `DashboardServlet`, you can retrieve user data using `UserDAO` and create a `DashboardModel` that includes both the user and the camera details.

java
Αντιγραφή κώδικα

```java
@WebServlet("/dashboard")
public class DashboardServlet extends HttpServlet {
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
        // Retrieve user from UserDAO (existing code)
        UserDAO userDao = new UserDAO();
        User user = userDao.getUserById(1);  // Assume user ID = 1 for simplicity

        // Create Camera object (for example, IPCamera)
        Camera camera = new IPCamera("Model XYZ", "192.168.1.1");

        // Encapsulate user and camera in DashboardModel
        DashboardModel dashboardModel = new DashboardModel(user, camera);

        // Pass the model to JSP
        request.setAttribute("dashboardModel", dashboardModel);
        RequestDispatcher dispatcher = request.getRequestDispatcher("/home.jsp");
        dispatcher.forward(request, response);
    }
}
```

4. Home JSP (View)

In *home.jsp*, you can access the *DashboardModel* and display both user and camera details.

jsp
Αντιγραφή κώδικα

```
<%@ page import="yourpackage.User" %>
<%@ page import="yourpackage.Camera" %>
<%@ page import="yourpackage.DashboardModel" %>
<html>
    <body>
        <h2>Welcome, ${dashboardModel.user.name}</h2>
        <h3>Camera Model: ${dashboardModel.camera.model}</h3>
        <p>IP Address: ${dashboardModel.camera.ipAddress}</p>
        <p>Status: Streaming...</p>
    </body>
</html>
```

5. Modular Integration

- **Encapsulation**: The *DashboardModel* encapsulates both user and camera objects, ensuring that both pieces of data can be passed around efficiently.
- **Modular**: You maintain modularity by keeping *Camera* abstract, *IPCamera* concrete, and the business logic in a separate servlet or controller. The *DashboardModel* can evolve as you add more functionality, like additional camera types or user data.

6. UserDAO Integration

Since you already have UserDAO, you can easily retrieve and pass user details along with the camera object to the JSP page, providing a dynamic and integrated view.

If you'd like, I can help further customize or refine this approach based on specific needs (e.g., adding more camera types, handling form submissions, etc.).

You're building a complex and robust system, and I can definitely help you structure the database and plan out how to implement features like time-limited footage storage and the integration of **MotionEyeOS** with your setup. Here's how I would approach it:

1. Database Structure for Your Use Case

Given that you have users, cameras, footage, settings, and logs, we need to extend your current database design to support **MotionEyeOS integration**, **camera settings**, and **dynamic storage management** (like rotating storage). Let's break it down.

Entities in the Database

- **User Accounts (user_accounts)**: Stores the user credentials and roles (e.g., admin, regular user).
- **Camera Settings (camera_settings)**: Stores settings related to the cameras being used.
- **Recorded Footage (recorded_footage)**: Tracks the footage files, including their timestamps and duration.
- **Log Data (log_data)**: Stores logs related to user actions (e.g., camera settings updated).
- **Storage Management (storage_management)**: Manages rotating footage storage (e.g., storageFootageOne, storageFootageTwo).

Here's how we can extend your database to support these:

Camera Settings Table

This table links cameras to users. We can add fields to hold the camera's streaming URL, model, or other camera-specific details.

sql
Αντιγραφή κώδικα
```
CREATE TABLE `camera_settings` (
  `camera_id` INT(11) NOT NULL AUTO_INCREMENT,
  `user_id` INT(11) NOT NULL,
  `streaming_url` VARCHAR(255) NOT NULL,  -- For MotionEyeOS stream URL or
similar
  `camera_model` VARCHAR(50),
  `camera_name` VARCHAR(255),  -- User's custom camera name (from
user_subtitles)
  PRIMARY KEY (`camera_id`),
  FOREIGN KEY (`user_id`) REFERENCES `user_accounts`(`user_id`) ON DELETE
CASCADE
);
```

Storage Management Table

For the rotating storage system (`storageFootageOne`, `storageFootageTwo`), we create a new table to track which storage is active, and when the footage should be swapped:

sql
Αντιγραφή κώδικα
```
CREATE TABLE `storage_management` (
  `storage_id` INT(11) NOT NULL AUTO_INCREMENT,
  `storage_name` VARCHAR(50) NOT NULL,  -- e.g., "storageFootageOne",
"storageFootageTwo"
  `is_active` TINYINT(1) NOT NULL,  -- 1 if active, 0 if inactive
  `start_time` TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
  `end_time` TIMESTAMP,  -- When the storage should be swapped out (1 week
after start)
```

```
  PRIMARY KEY (`storage_id`)
);
```

Recorded Footage Table

You can continue using the `recorded_footage` table, but now we'll include the `storage_id` so that we can associate recorded footage with the correct storage.

sql
Αντιγραφή κώδικα

```
CREATE TABLE `recorded_footage` (
  `footage_id` INT(11) NOT NULL AUTO_INCREMENT,
  `timestamp` TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
  `camera_id` INT(11) NOT NULL,
  `file_path` VARCHAR(255) NOT NULL,
  `duration` DOUBLE DEFAULT NULL,
  `storage_id` INT(11) NOT NULL,  -- Reference to storage management
  PRIMARY KEY (`footage_id`),
  FOREIGN KEY (`camera_id`) REFERENCES `camera_settings`(`camera_id`),
  FOREIGN KEY (`storage_id`) REFERENCES `storage_management`(`storage_id`)
);
```

Log Data for Camera Changes

This is the same as what you already have. This log table tracks all changes related to camera settings or footage.

sql
Αντιγραφή κώδικα

```
CREATE TABLE `log_data` (
  `log_id` INT(11) NOT NULL AUTO_INCREMENT,
  `timestamp` TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
  `user_id` INT(11) NOT NULL,
  `event_type` VARCHAR(50) NOT NULL,
  `details` TEXT DEFAULT NULL,
  PRIMARY KEY (`log_id`),
  FOREIGN KEY (`user_id`) REFERENCES `user_accounts`(`user_id`)
);
```

2. Time-Limited Footage and Storage Swap Logic

You can implement this in two ways:

1. **Using triggers** to handle storage rotation.
2. **Using an external scheduler or cron job** to manage the footage rotation.

Rotating Storage Logic with Triggers

To rotate storage every week, you could create a **BEFORE INSERT** trigger that checks whether the footage should be inserted into the active storage. If it's past the defined time limit (one week), it should rotate.

sql
Αντιγραφή κώδικα
```
DELIMITER $$

CREATE TRIGGER `rotate_storage_footage` BEFORE INSERT ON `recorded_footage`
FOR EACH ROW BEGIN
    -- Check if current storage is past its time limit and needs to be swapped
    DECLARE new_storage_id INT;
    DECLARE active_storage INT;

    -- Get the active storage (i.e., where footage is being recorded)
    SELECT `storage_id` INTO active_storage
    FROM `storage_management`
    WHERE `is_active` = 1
    LIMIT 1;
    -- If the active storage is older than a week, swap to the next storage
    SELECT `storage_id` INTO new_storage_id
    FROM `storage_management`
    WHERE `is_active` = 0
    LIMIT 1;

    -- Swap the storage
    UPDATE `storage_management` SET `is_active` = 1 WHERE `storage_id` =
new_storage_id;
    UPDATE `storage_management` SET `is_active` = 0 WHERE `storage_id` =
active_storage;

    -- Assign the new storage ID to the footage
    SET NEW.storage_id = new_storage_id;

END
$$

DELIMITER ;
```

External Scheduler (Cron Job)

Alternatively, you could write a script to run every day (or week) using cron jobs or a scheduled task. This script would check the `storage_management` table to see which storage has been active for over a week and swap the storage accordingly.

3. MotionEyeOS Integration

MotionEyeOS can be integrated into your system by configuring it to stream to a URL that your cameras will access. In the database, you can store the streaming URL in the `camera_settings` table, which can be updated dynamically via your web interface.

- **Camera Streaming URL**: You can store the IP camera's RTSP/MJPEG URL (e.g., `http://192.168.1.100:8081`) for each camera in the `camera_settings` table.
- You can create a **MotionEyeOS web app** to allow users to configure their camera streams directly from the user interface.

4. Overall Workflow

- **User Login**: User logs in via `user_accounts`.
- **Camera Setup**: The user configures their camera in `camera_settings`, including setting a custom name from `user_subtitles`.
- **Streaming**: MotionEyeOS streams the video to the configured URL, which is recorded in the `recorded_footage` table.
- **Storage Rotation**: Every week, a trigger or scheduled task swaps the active storage, moving footage from one storage set to the next.
- **Log Events**: Every time a user interacts with the system (e.g., changes settings, uploads footage), log data is stored in the `log_data` table.

5. Suggestions and Forward Thinking

- **Data Redundancy**: Consider adding redundancy or backup mechanisms for storing footage, especially if you plan to scale. Cloud storage or offsite backups can be integrated as secondary storage.
- **User Permissions**: You could extend the `user_accounts` table to support finer-grained permissions, allowing users to manage their own footage but not interfere with other users' footage.
- **Security**: Make sure to encrypt sensitive data such as `hashed_password` and any potential camera credentials.

This approach will give you a modular, scalable solution that can evolve as you add new features like multi-camera setups, enhanced footage management, or cloud-based storage. Let me know if you'd like to explore specific parts of this design in more detail!

To design a robust database for your CCTV system incorporating all the features and considerations you mentioned, including user accounts, camera settings, storage management, and a week-based footage rotation system, we need to organize the schema carefully. Here's a detailed approach that outlines the database structure, tables, foreign keys, triggers, and other features based on your requirements:

Database Structure Overview

1. **User Management**:
   - **user_accounts**: For login and registration.
   - **user_subtitles**: To store custom camera names (subtitles).
2. **CCTV Management**:
   - **camera_settings**: Stores settings for each camera, including which user owns it.
   - **recorded_footage**: Stores metadata for each recorded footage (with timestamps).
   - **storage_management**: Manages different storage units for video footage (storageFootageOne, storageFootageTwo).
   - **log_data**: Logs events like camera settings changes, footage creation, user actions.
3. **Camera Interaction**:

Your current database schema for **CCTV_DB_Management** looks well-organized and comprehensive, but there are some areas where improvement, optimization, and attention to best practices could help enhance the design. Here's a detailed analysis, comparison with your previous model, and suggestions for improvements:

1. User Accounts Table (`user_accounts`)

-

Password Complexity Check:

-
  - The `hashed_password` complexity check using `REGEXP` for special characters is a good security measure, but it might not fully address password strength (e.g., ensuring upper/lowercase, numbers, and symbols).
  - **Recommendation:** Consider using a more robust password hashing function (e.g., bcrypt, Argon2) at the application layer. Also, ensure that password hashes are being salted and not directly stored as plain hashes.
-

Admin Role:

-
  - The `admin_role` column is a boolean flag, which is fine, but it could be extended for better flexibility if your application scales.
  - **Recommendation:** A `role_id` that links to a separate `roles` table (for more granular role management) could be more scalable. The `roles` table would define the different roles (e.g., `admin`, `moderator`, `user`), and the `user_accounts` table would reference this `role_id`.

2. Camera Settings Table (`camera_settings`)

- **Foreign Key to User Accounts:**
  - The `camera_settings` table has a foreign key to `user_accounts`, which makes sense for associating users with specific cameras.
  - **Recommendation:** If multiple users can manage the same camera, consider creating a junction table (e.g., `camera_user_associations`) that would allow many-to-many relationships between users and cameras.

3. Recorded Footage Table (`recorded_footage`)

-

Data Retention Policy:

-

  - You're implementing a trigger to delete footage older than one year. This is a great way to manage storage and ensure compliance with data retention policies.
  - **Recommendation:** Consider storing a `retention_period` or `expiration_date` for each footage record so that you can manage data more dynamically. Having a field like `archived BOOLEAN` or `deleted_at TIMESTAMP` would allow for logical deletion (soft delete).

-

Indexes:

-

  - You've added indexes on `camera_id` and `timestamp`, which is excellent for query performance, especially if you're frequently querying footage based on these columns.
  - **Recommendation:** Depending on your query patterns, you might want to consider additional indexing on `user_id` or composite indexes (e.g., `camera_id, timestamp`) if you often query footage by both.

4. Log Data Table (`log_data`)

-

Logging User Account Modifications:

-
  - You're logging user account modifications using a trigger, which is a good auditing approach.
  - **Recommendation:** You might want to extend the trigger to capture updates or deletions on the `user_accounts` table as well, not just insertions. Consider creating separate event types for `UPDATE` and `DELETE`.

-

Additional Audit Data:

-
  - You might want to add more specific details for audit logs, such as IP address, user agent (browser/device info), or session ID for traceability.
  - **Recommendation:** Add optional fields like `ip_address`, `user_agent`, or `session_id` to `log_data` to enhance audit information.

5. Remote Assistance Logs Table (`remote_assistance_logs`)

- **Foreign Key Relationships:**
  - You've added foreign key relationships for both `user_id` and `admin_id`, which is great for ensuring data integrity.
  - **Recommendation:** Ensure that the `admin_id` is indeed representing an admin user. You may also want to create a `roles` table for better tracking of user roles in both `user_accounts` and `remote_assistance_logs`.

6. Triggers:

-

Automatic Deletion of Outdated Footage:

-
  - The trigger for deleting footage older than 1 year is a good idea, but it could be more flexible.
  - **Recommendation:** Consider adding a `retention_period` column to `recorded_footage` to specify when footage should be deleted (instead of a fixed one-year policy). This way, you can handle footage differently for different cameras or users.

-

Trigger Logging for Camera Settings and User Accounts:

-
    - Triggers are implemented for logging actions on `user_accounts` and `camera_settings`, which is a great way to ensure auditing.
    - **Recommendation:** Similar to the `log_data` table, consider adding a trigger for `recorded_footage` modifications (e.g., adding new footage, updates). For instance, you might want to log when footage is flagged for deletion or retention.

7. Database Optimizations and Best Practices:

-

Indexes:

-
    - You're indexing `camera_id` and `timestamp` on `recorded_footage`, which is excellent for querying footage quickly.
    - **Recommendation:** As mentioned earlier, consider composite indexes on frequently queried columns. For example, `camera_id` and `timestamp` together, if often used together in queries.

-

Normalization:

-
    - The schema is relatively normalized, but in certain cases, like logging and remote assistance, it may be worth checking whether some columns (e.g., `details` in log tables) are too large for efficient storage.
    - **Recommendation:** You might want to split larger text-based fields (like `details`) into separate tables if they grow large or if you need to keep logs for a long period.

-

Backup and Disaster Recovery:

-

  - You may want to ensure that important tables (such as `user_accounts`, `log_data`, `recorded_footage`) have a backup strategy, especially in a production environment.
  - **Recommendation:** Set up regular database backups and ensure transactional integrity for critical operations.

8. Security Considerations:

-

Password Storage:

-

  - You're storing hashed passwords, which is a good practice, but ensure you're using a secure hashing algorithm (bcrypt, Argon2) that can protect against brute-force attacks.
  - **Recommendation:** Ensure that the password is hashed at the application layer before storing in the database (don't store plain text or weak hashes like MD5 or SHA1).

-

Access Control:

-

  - The `permissions` field in `user_accounts` is a string field. You might want to consider normalizing this into a separate table for more granular access control (e.g., different permission levels per user).
  - **Recommendation:** Implement a **permissions** table and a **user_permissions** join table to give more flexibility in managing access rights across the system.

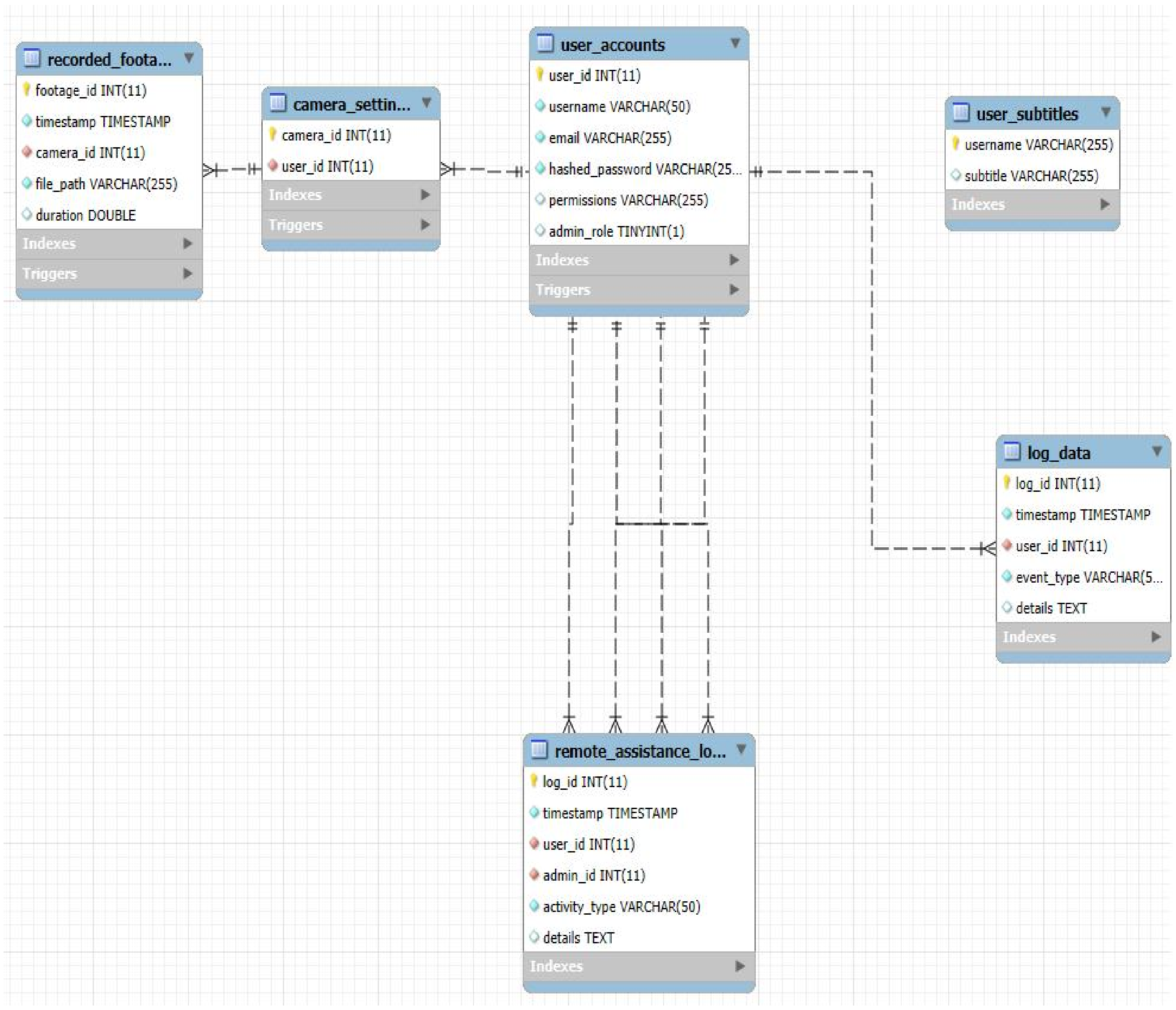
9. Data Privacy & Compliance:

- **Retention and Access Control:**
  - Depending on the jurisdiction and use case (e.g., GDPR or CCPA), you might need to ensure that personal data (e.g., footage, logs) is handled in compliance with data privacy laws.
  - **Recommendation:** You can add features for **data anonymization**, **data encryption**, and **explicit user consent** for footage storage, if applicable.

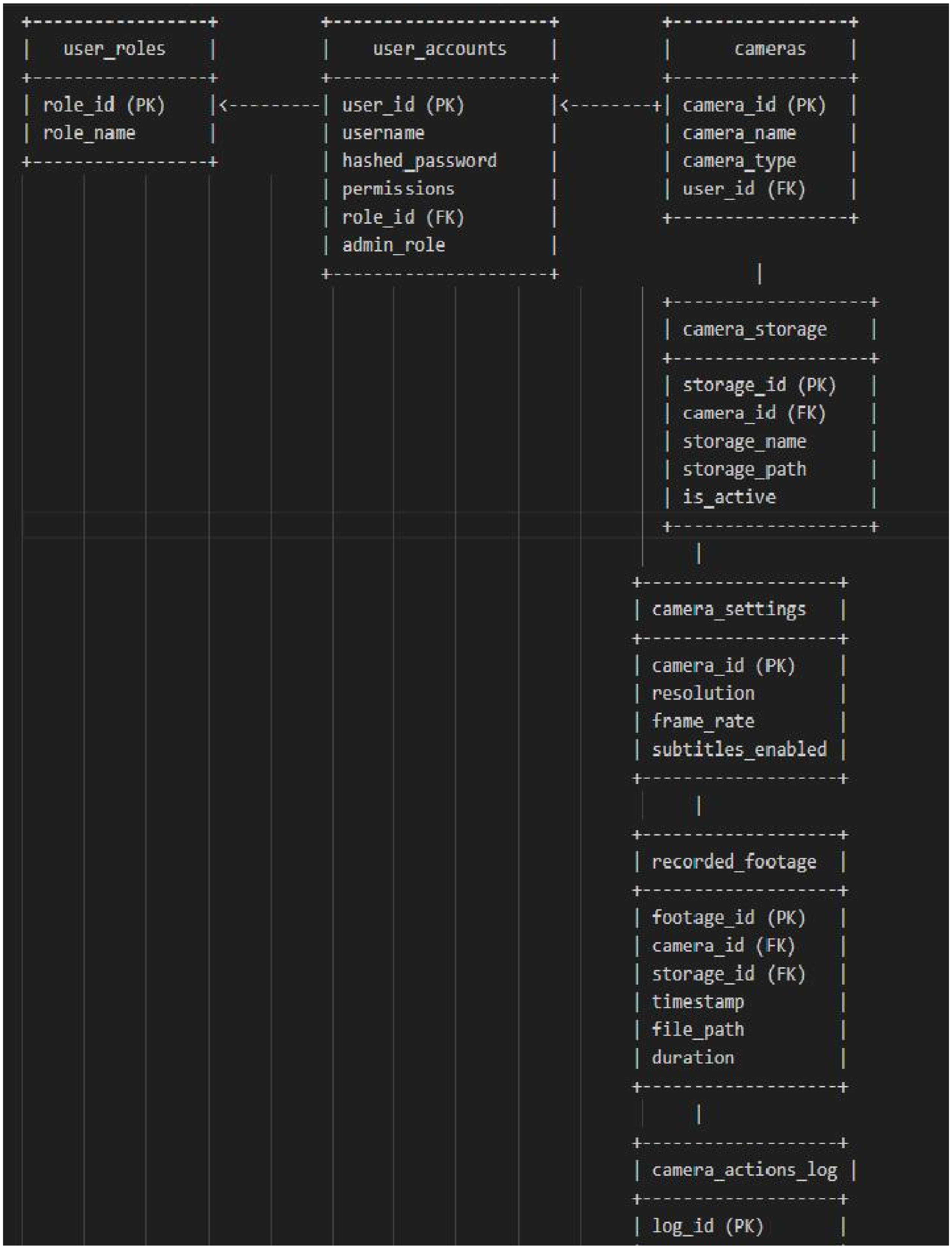
Summary of Recommendations:

1. **Enhance Password Security** – Use bcrypt or Argon2 for hashing passwords and move the password complexity check to the application level.
2. **Role Management** – Move from a boolean `admin_role` to a more flexible `roles` table to handle multiple roles.
3. **Flexible Data Retention** – Introduce a dynamic retention period for footage rather than a fixed 1-year rule.
4. **Log More Data** – Capture more information in logs (e.g., IP addresses, user agents) for traceability and compliance.
5. **Improve Query Performance** – Add composite indexes and optimize queries for high-performance systems.
6. **Backup Strategy** – Set up a reliable backup and disaster recovery plan for critical data like footage and user logs.

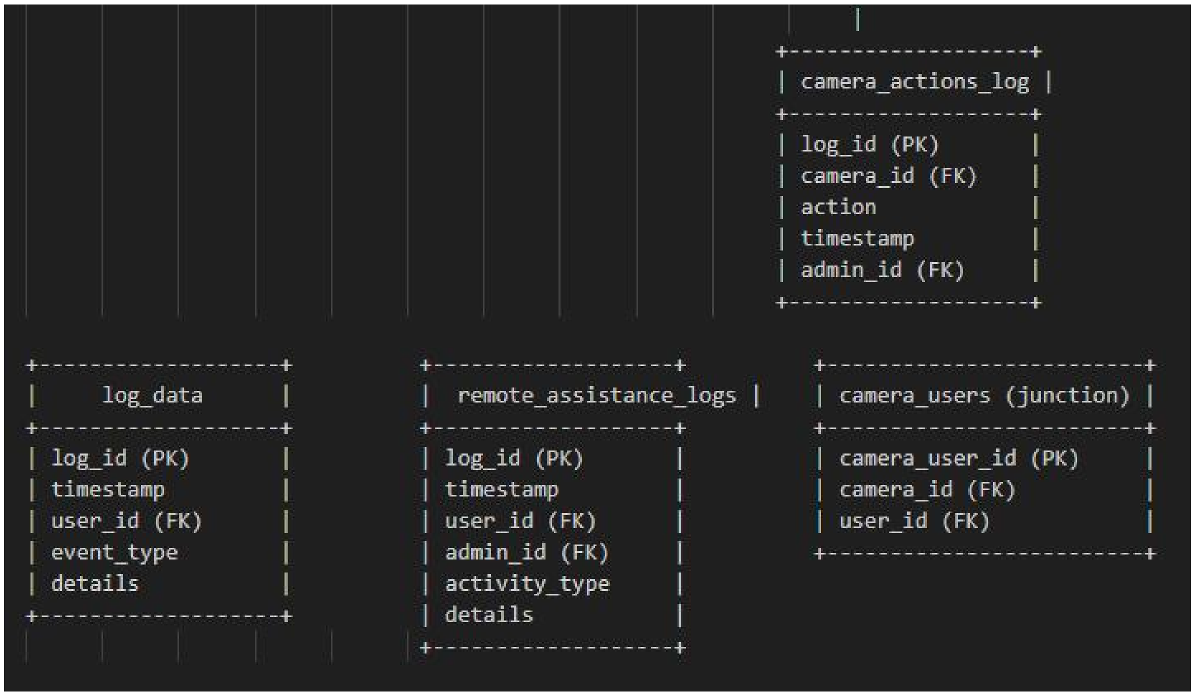
EXISTING Database



ExtendingDatabase



```
+-----------------+        +---------------------+        +-----------------+
|   user_roles    |        |    user_accounts    |        |     cameras     |
+-----------------+        +---------------------+        +-----------------+
| role_id (PK)    |<---------| user_id (PK)      |<---------+| camera_id (PK)  |
| role_name       |        | username            |        | camera_name     |
+-----------------+        | hashed_password     |        | camera_type     |
                           | permissions         |        | user_id (FK)    |
                           | role_id (FK)        |        +-----------------+
                           | admin_role          |
                           +---------------------+               |
                                                          +------------------+
                                                          | camera_storage   |
                                                          +------------------+
                                                          | storage_id (PK)  |
                                                          | camera_id (FK)   |
                                                          | storage_name     |
                                                          | storage_path     |
                                                          | is_active        |
                                                          +------------------+
                                                               |
                                                     +------------------+
                                                     | camera_settings  |
                                                     +------------------+
                                                     | camera_id (PK)   |
                                                     | resolution       |
                                                     | frame_rate       |
                                                     | subtitles_enabled |
                                                     +------------------+
                                                          |
                                                     +------------------+
                                                     | recorded_footage |
                                                     +------------------+
                                                     | footage_id (PK)  |
                                                     | camera_id (FK)   |
                                                     | storage_id (FK)  |
                                                     | timestamp        |
                                                     | file_path        |
                                                     | duration         |
                                                     +------------------+
                                                          |
                                                     +------------------+
                                                     | camera_actions_log |
                                                     +------------------+
                                                     | log_id (PK)      |
```



```
                                                          |
                                                 +--------------------+
                                                 | camera_actions_log |
                                                 +--------------------+
                                                 | log_id (PK)        |
                                                 | camera_id (FK)     |
                                                 | action             |
                                                 | timestamp          |
                                                 | admin_id (FK)      |
                                                 +--------------------+

+--------------------+        +--------------------+        +-------------------------+
|     log_data       |        |  remote_assistance_logs |   | camera_users (junction) |
+--------------------+        +--------------------+        +-------------------------+
| log_id (PK)        |        | log_id (PK)        |        | camera_user_id (PK)     |
| timestamp          |        | timestamp          |        | camera_id (FK)          |
| user_id (FK)       |        | user_id (FK)       |        | user_id (FK)            |
| event_type         |        | admin_id (FK)      |        +-------------------------+
| details            |        | activity_type      |
+--------------------+        | details            |
                              +--------------------+
```