**Movie Backend API: Design, Caching Strategy, and Security Considerations**

**1. Design Decisions**

The design of this API is centered around providing a secure, scalable, and efficient movie database backend, allowing users to authenticate, search for movies, and manage their favorite movies while still sdhering to the technical assessment requirement. Key design decisions include:

* **Modular Route Structure**:
  + The routes are separated into logical files (authRoutes.js, movieRoutes.js, favoritesRoutes.js), ensuring clear separation of concerns and making the project more maintainable.
  + Middleware is used for handling authentication, ensuring that protected routes require valid JWT tokens.
* **AWS Cognito for Authentication**:
  + I chose AWS Cognito for user authentication due to its secure and scalable nature. It abstracts much of the complexity around authentication, token management, and user pools while integrating seamlessly with other AWS services for future use. I try looking at Okta as well but to signup for free trial requires **business email** which I don’t have yet.
* **MySQL for Data Persistence**:
  + MySQL was selected as the database to store user data and favorite movies. It's a robust relational database that supports ACID transactions, ensuring data consistency, especially for user management and movie relationships. I’ve tried MongoDB long ago, but I don’t really like the concept of no

**2. Caching Strategy**

To improve the efficiency and reduce the number of external API calls, I implemented a caching layer using **Redis**. The caching strategy focuses on movie search queries to minimize calls to the external movie database (TMDb).

* **Movie Search Caching**:
  + When a user searches for a movie, the result is first checked in the Redis cache.
  + If the result is cached, the API returns the cached response directly, reducing the need for repeated API calls to TMDb.
  + If the result is not cached, the API fetches the movie data from TMDb, stores it in Redis with an expiration time (e.g., 1 hour), and then returns the result.
  + This approach significantly improves performance, especially for popular search queries that may be repeated across users.
* **Expiration Policy**:
  + Each cached search result is set to expire after 1 hour. This ensures that the cache is up to date while still offering performance benefits by reducing external API calls.

**3. Database Design**

The **MySQL database** is designed to store user data, favorite movies, and movie details efficiently. The design follows a relational database model to maintain relationships between users and their favorite movies.It uses three main tables: Users, Movies, and Favourites to store the necessary data.

**MySQL-Specific Data Types**

* CHAR(36): Used to store UUIDs (for id in Users and Favourites). It’s a fixed-length string, which is appropriate for UUIDs.
* VARCHAR(255): Used for variable-length fields like username, email, and title to save space when storing shorter strings.
* BIGINT: Used for the id of movies, assuming that the external API (TMDb) uses large integers for movie IDs.

**1. Users Table**

This table will store information about the users, including their authentication details from the identity provider (e.g., AWS Cognito or Okta).

|  |  |  |  |
| --- | --- | --- | --- |
| Column Name | Data Type | Constraints | Description |
| id | UUID | PRIMARY KEY | Unique identifier for the user |
| provider\_id | VARCHAR | NOT NULL, UNIQUE | ID provided by the external identity provider |
| username | VARCHAR | NOT NULL | Username of the user |
| email | VARCHAR | NOT NULL, UNIQUE | User's email address |
| created\_at | TIMESTAMP | DEFAULT CURRENT\_TIMESTAMP | When the user was created |
| updated\_at | TIMESTAMP | ON UPDATE CURRENT\_TIMESTAMP | When the user was last updated |

Explanation:

* The id column uses CHAR(36) to store UUIDs, which are widely used for globally unique identifiers in MySQL.
* provider\_id and email are marked as UNIQUE to prevent duplicate entries. The provider\_id stores the user ID from the authentication service (AWS Cognito, Okta).
* created\_at and updated\_at use TIMESTAMP with automatic updates on record modification.

**2. Movies Table**

This table will store information about movies that users interact with, including their favorite movies. The data can be fetched from the external movie database (e.g., TMDb) when needed.

|  |  |  |  |
| --- | --- | --- | --- |
| Column Name | Data Type | Constraints | Description |
| id | BIGINT | PRIMARY KEY | Unique identifier for the movie (from TMDb) |
| title | VARCHAR | NOT NULL | Title of the movie |
| release\_date | DATE | NULLABLE | Release date of the movie |
| overview | TEXT | NULLABLE | Short description or overview of the movie |
| poster\_url | VARCHAR | NULLABLE | URL for the movie poster |
| created\_at | TIMESTAMP | DEFAULT CURRENT\_TIMESTAMP | When the movie was added to the database |
| updated\_at | TIMESTAMP | ON UPDATE CURRENT\_TIMESTAMP | When the movie record was last updated |

Explanation:

* Movie information (e.g., title, release\_date, etc.) is fetched from TMDb and cached in the database for performance.
* The id is the movie's unique identifier from the external movie database (TMDb ID).

**3. Favourites Table**

This is a join table that links Users with their favorite Movies, allowing for a many-to-many relationship. Each user can have multiple favorite movies, and the same movie can be favorited by multiple users.

|  |  |  |  |
| --- | --- | --- | --- |
| Column Name | Data Type | Constraints | Description |
| id | UUID | PRIMARY KEY | Unique identifier for the favorite entry |
| user\_id | UUID | FOREIGN KEY REFERENCES Users(id) ON DELETE CASCADE | ID of the user who favorited the movie |
| movie\_id | BIGINT | FOREIGN KEY REFERENCES Movies(id) | ID of the movie that was favorited |
| created\_at | TIMESTAMP | DEFAULT CURRENT\_TIMESTAMP | When the movie was added to favorites |

Explanation:

* user\_id and movie\_id are foreign keys that establish relationships with the Users and Movies tables, respectively.
* The UNIQUE KEY (user\_id, movie\_id) constraint prevents duplicate favorite entries for the same user and movie.
* The ON DELETE CASCADE ensures that if a user is deleted, their favorite movies are automatically removed.

**Schema Relationships**

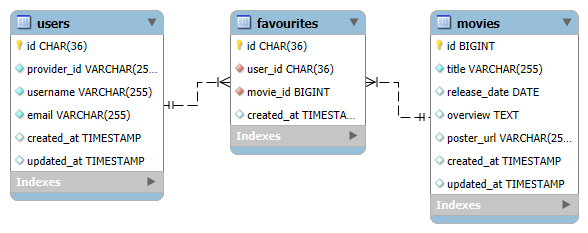
1. **One-to-Many Relationship between Users and Favourites:**

* Each user can have multiple favorite movies.
* The Favourites table links a user to their favorite movies.

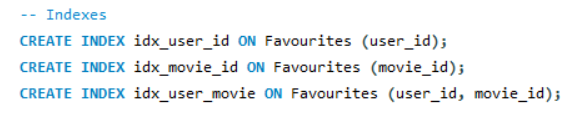
1. **Many-to-Many Relationship between Users and Movies (via Favourites table):**

* A user can favorite multiple movies.
* A movie can be favorited by multiple users.

**ER Diagram**

****

**Indexes and Performance Optimizations**

****

1. **Indexes for Foreign Keys:**
   * Indexes on foreign keys (user\_id, movie\_id) in the Favourites table will improve performance for queries that involve joining users and movies.
2. **Composite Index for Search Performance:**
   * For faster lookups of a user's favorite movies, you can create a composite index on user\_id and movie\_id.

**4. Handling Security Concerns**

Security is a top priority in the design of this API, with various measures implemented to protect user data and the system from common attacks.

* **Authentication and Authorization**:
  + I used **AWS Cognito** for user authentication and JWT (JSON Web Tokens) for authorizing access to protected routes. JWT tokens are validated using middleware, ensuring only authenticated users can access sensitive data like their favorites or search results.
* **Input Validation and Sanitization**:
  + **validator.js** was used to sanitize and validate user inputs (e.g., email, username, search queries). This prevents attacks like SQL injection and cross-site scripting (XSS) by ensuring that only properly formatted inputs are processed by the system.
  + All inputs are validated before being passed into database queries or external API calls to ensure data integrity.
* **Rate Limiting**:
  + To prevent brute-force attacks on sensitive endpoints like login and sign-up, **express-rate-limit** was used to restrict the number of requests from a single IP within a certain time window.
  + This helps mitigate security risks by limiting repeated failed attempts, which could otherwise lead to password guessing or denial-of-service attacks.
* **Use of HTTPS (in Production)**:
  + In production environments, the API should be run over **HTTPS** to ensure that data is encrypted in transit, preventing man-in-the-middle attacks.
  + Security headers are enforced using **helmet.js**, which adds standard security-related HTTP headers to protect against attacks like clickjacking and cross-site scripting.
* **Token Expiration and Refresh**:
  + Tokens issued by AWS Cognito have a limited lifespan, and users are required to authenticate again or refresh tokens once they expire, ensuring session security.
* **Error Handling**:
  + The application avoids leaking sensitive information in error messages. For example, database errors or invalid tokens are caught and returned as generic error messages (e.g., "Unauthorized" or "Internal Server Error") while logging detailed errors on the server.

**Conclusion**

In summary, the API was designed with security, scalability, and performance in mind. AWS Cognito handles secure user authentication and authorization, while Redis caching optimizes performance by reducing external API calls. The database design ensures efficient data storage and relationships between users and movies. Security best practices, such as input sanitization, rate limiting, and secure token management, were implemented to protect the application from common threats. This might miss some security concerns which assumes will be covered by the frontend as well which is beyond the scope of this assessment.