Documentation

Objectives

Welcome to the documentation for the Node.js application, a versatile and modular system designed for managing users, tasks, guilds, and related functionalities. This application provides a robust backend framework for creating and managing guilds, tracking user tasks and progress, and ensuring secure user authentication and authorization. With a focus on modularity and maintainability, this codebase is suitable for various scenarios, from task management systems to community-focused platforms.

This documentation aims to provide a comprehensive understanding of the Node.js application, covering the following key objectives:

1. **Architecture Overview:**
    - Explore the modular structure of the codebase, understanding how different components are organized for clarity and maintainability.
2. **Authentication and Authorization:**
    - Learn about the implemented authentication and authorization mechanisms, ensuring that user access is secure and roles are appropriately managed.
3. **User Management:**
    - Understand how the application handles user-related operations, including user creation, retrieval, modification, and deletion.
4. **Task Management:**
    - Explore the functionality related to task creation, retrieval, update, and deletion, providing insights into how the application manages user tasks.
5. **Guild Operations:**
    - Delve into the features related to guilds, including guild creation, modification, and deletion. Learn about the membership system and the roles assigned to guild members.
6. **Error Handling:**
    - Gain insights into the implemented error-handling mechanisms, ensuring that the application gracefully handles unexpected scenarios and provides meaningful error messages.
7. **Middleware Functions:**
    - Understand the middleware functions used for various purposes, such as user role checks, guild membership verification, and channel access control.
8. **File Uploads:**
    - Explore the handling of file uploads, particularly for guild banners and logos, and understand how the application manages and serves these files.
9. **Security Considerations:**
    - Learn about security practices implemented in the codebase, including input validation, protection against common vulnerabilities, and the overall security posture of the application.
10. **Best Practices and Recommendations:**
    - Receive guidance on best practices for maintaining and extending the codebase. Understand recommendations for code readability, performance optimization, and staying up-to-date with dependencies.

Endpoints

In this Project there are a reasonable amount of Endpoints that we need to discuss apart from the endpoints for Section A ; The Structure of the Endpoints is organized as such :

1. Authentication
2. User
3. Task
4. Task Progress
5. Guild
6. Channel
7. Quest

Authentication Endpoints

In the Authentication Endpoints, I have implemented 2 endpoints and they are :

1. POST /login

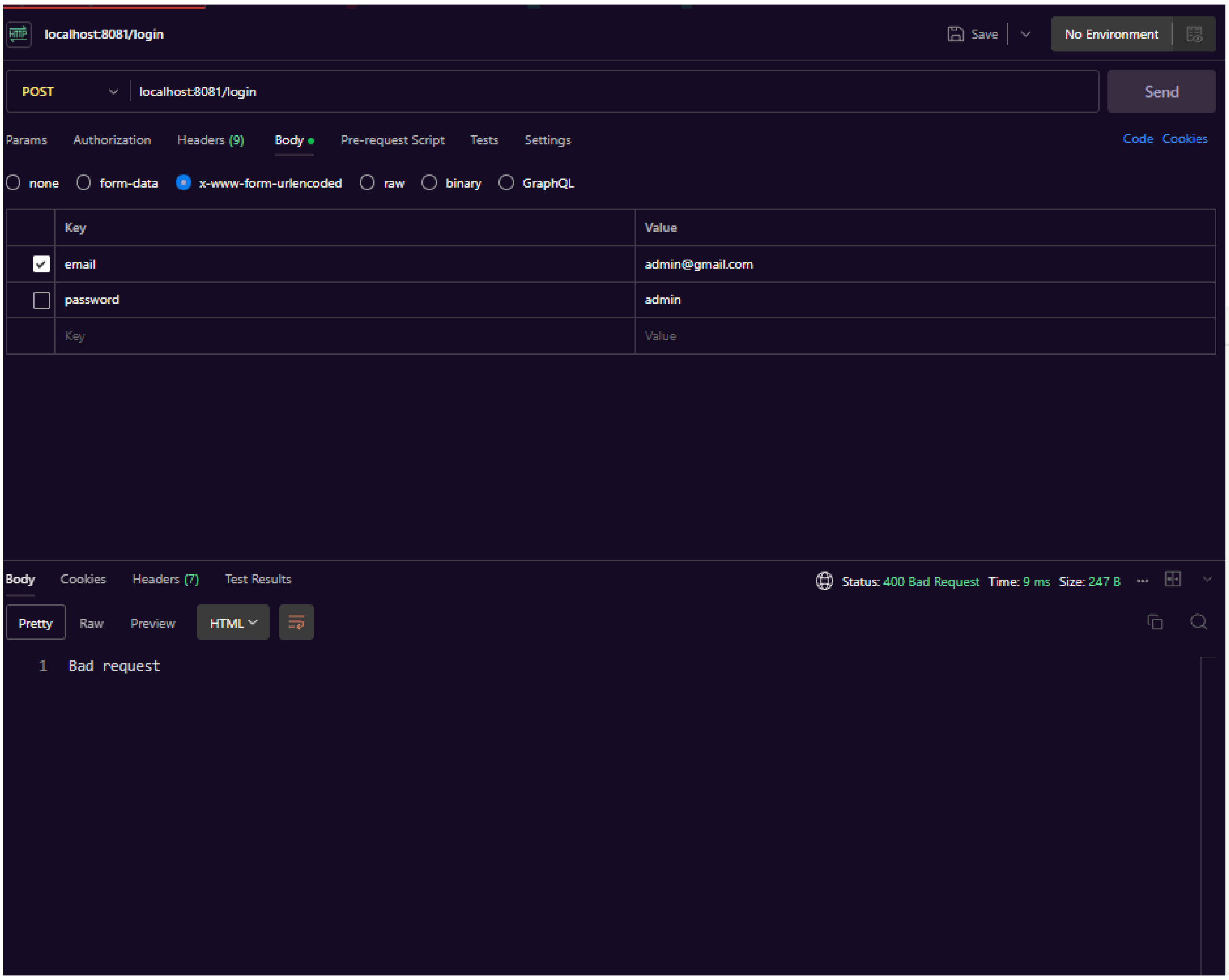
2. POST /refresh

The *POST /login* endpoint is supposed to be the endpoint that is responsible for authenticating a user so that he/she can make use of the features offered in the Node JS Application. The *POST /refresh* endpoint on the other hand is supposed to be present to allow the user to be reauthenticated without logging in by making use of a refresh-token. This mechanism has been implemented so that users have an seamless interaction with the application and to ensure that is the user can be persistently logged into the service.

```
- **Endpoint:**

    - `POST /login`
- **Description:**

    - This endpoint is responsible for authenticating users and generating access and refresh tokens upon successful login.
- **Request Payload:**

    - The endpoint expects a JSON payload with the following fields:
        - `email` (string): The email address of the user.
        - `password` (string): The user's password.
- **Response:**

    - If authentication is successful, the server responds with a status code of 201 (Created) and returns a JSON object containing:
        - `id` (string): User ID.
        - `type` (string): User role.
        - `refreshTkt` (string): Refresh token for extending the session.
        - `accessTkt` (string): Access token for authenticating future requests.
- **Error Responses:**
    - If either email or password are not provided, the server responds with a status code of 400 and the message 'Bad request'
    - If the user is not found, the server responds with a status code of 404 and the message "User Unknown."
```

Here are some sample screenshots on the usage of this endpoint :

1. Attempt to login without providing the email or password :



localhost:8081/login

POST localhost:8081/login

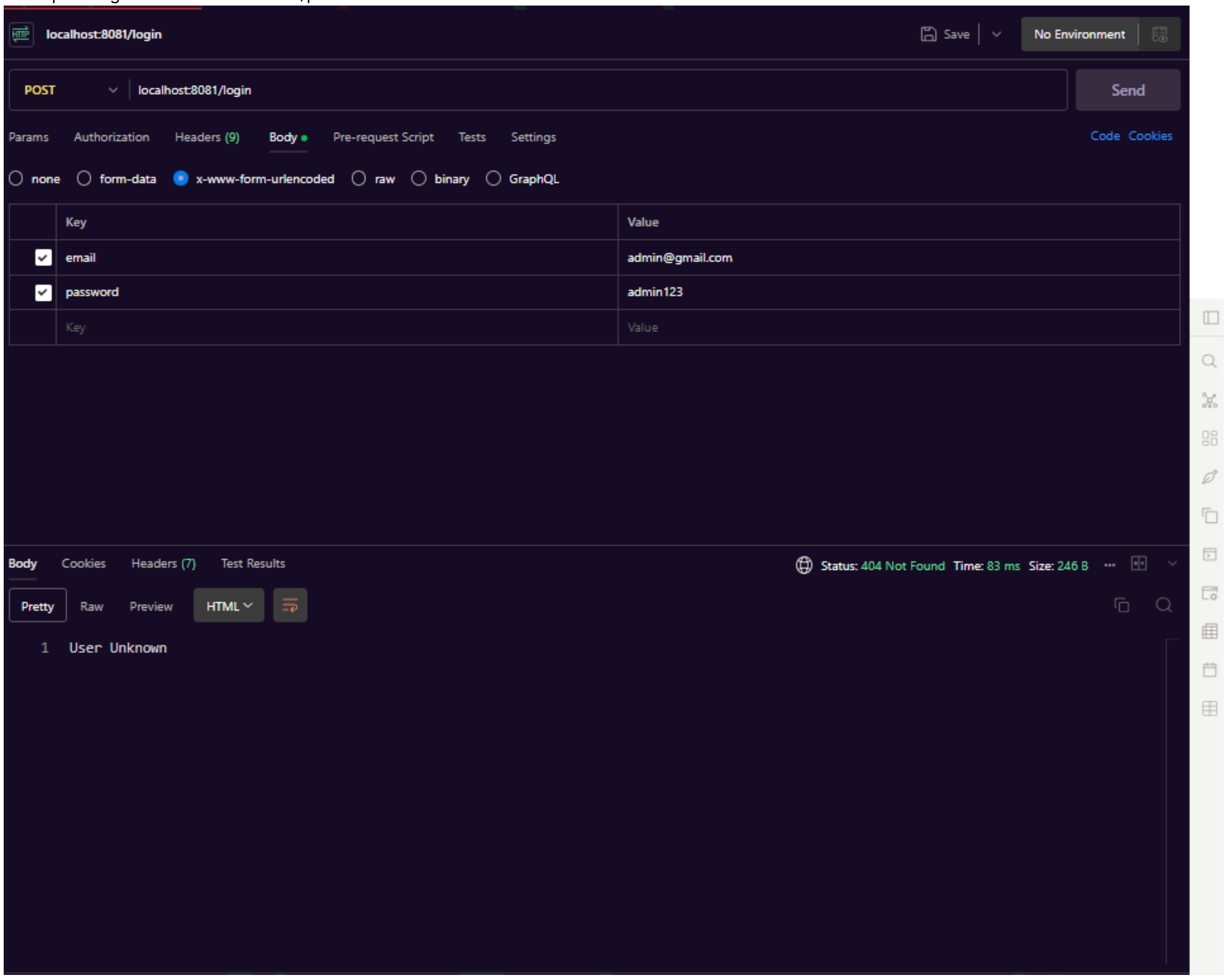
Body: x-www-form-urlencoded

	Key	Value
✓	email	admin@gmail.com
☐	password	admin
	Key	Value

Status: 400 Bad Request Time: 9 ms Size: 247 B

```
Bad request
```

2. Attempt to login with incorrect email/password :



localhost:8081/login

Save | No Environment

POST localhost:8081/login Send

Params Authorization Headers (9) Body Pre-request Script Tests Settings Code Cookies

none form-data [✓ x-www-form-urlencoded] raw binary GraphQL

	Key	Value
✓	email	admin@gmail.com
✓	password	admin123
	Key	Value

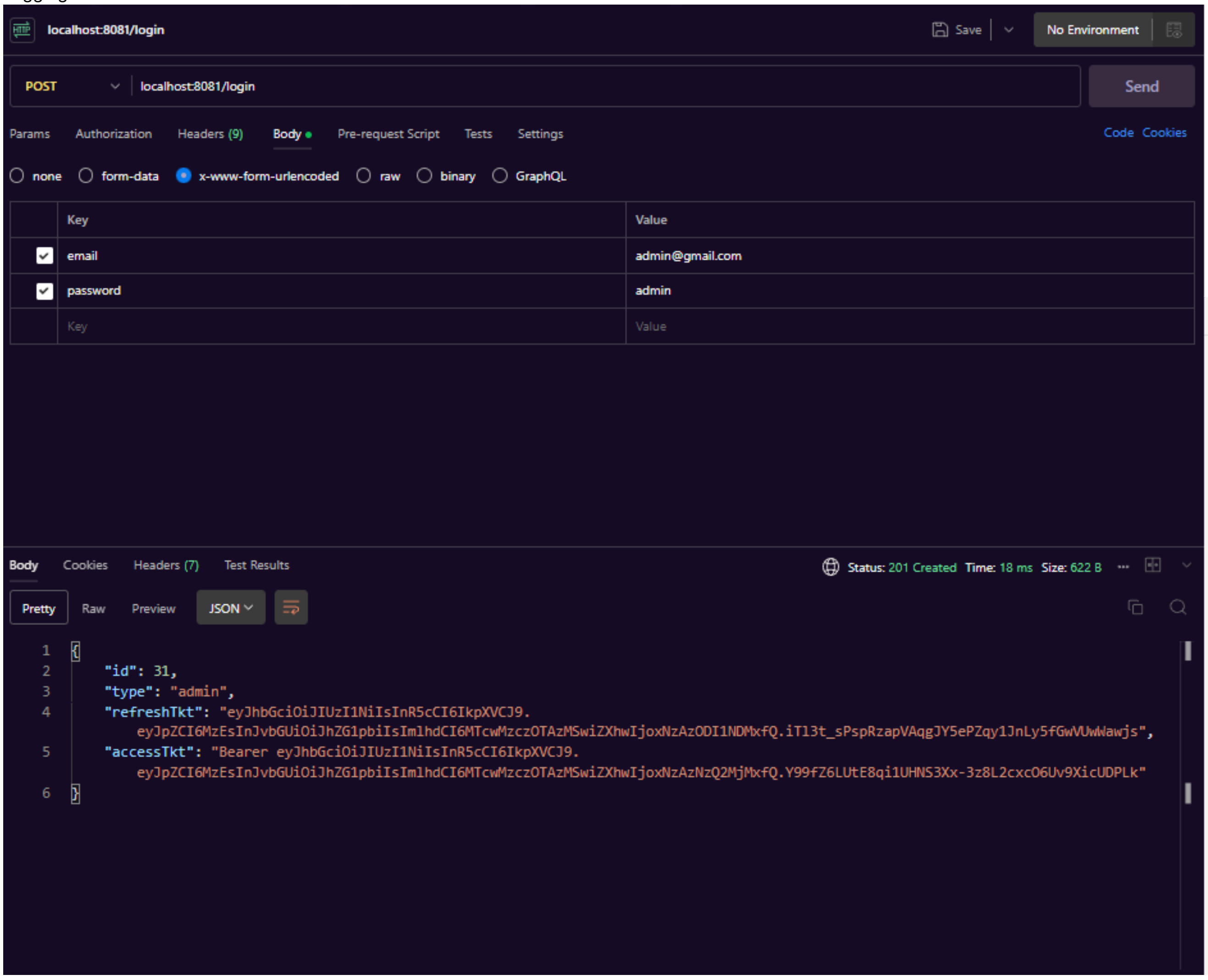
Body Cookies Headers (7) Test Results

Status: 404 Not Found Time: 83 ms Size: 246 B

Pretty Raw Preview HTML

```
User Unknown
```

3. Logging in with the Correct credentials :



POST localhost:8081/login

Body: x-www-form-urlencoded

	Key	Value
✓	email	admin@gmail.com
✓	password	admin
	Key	Value

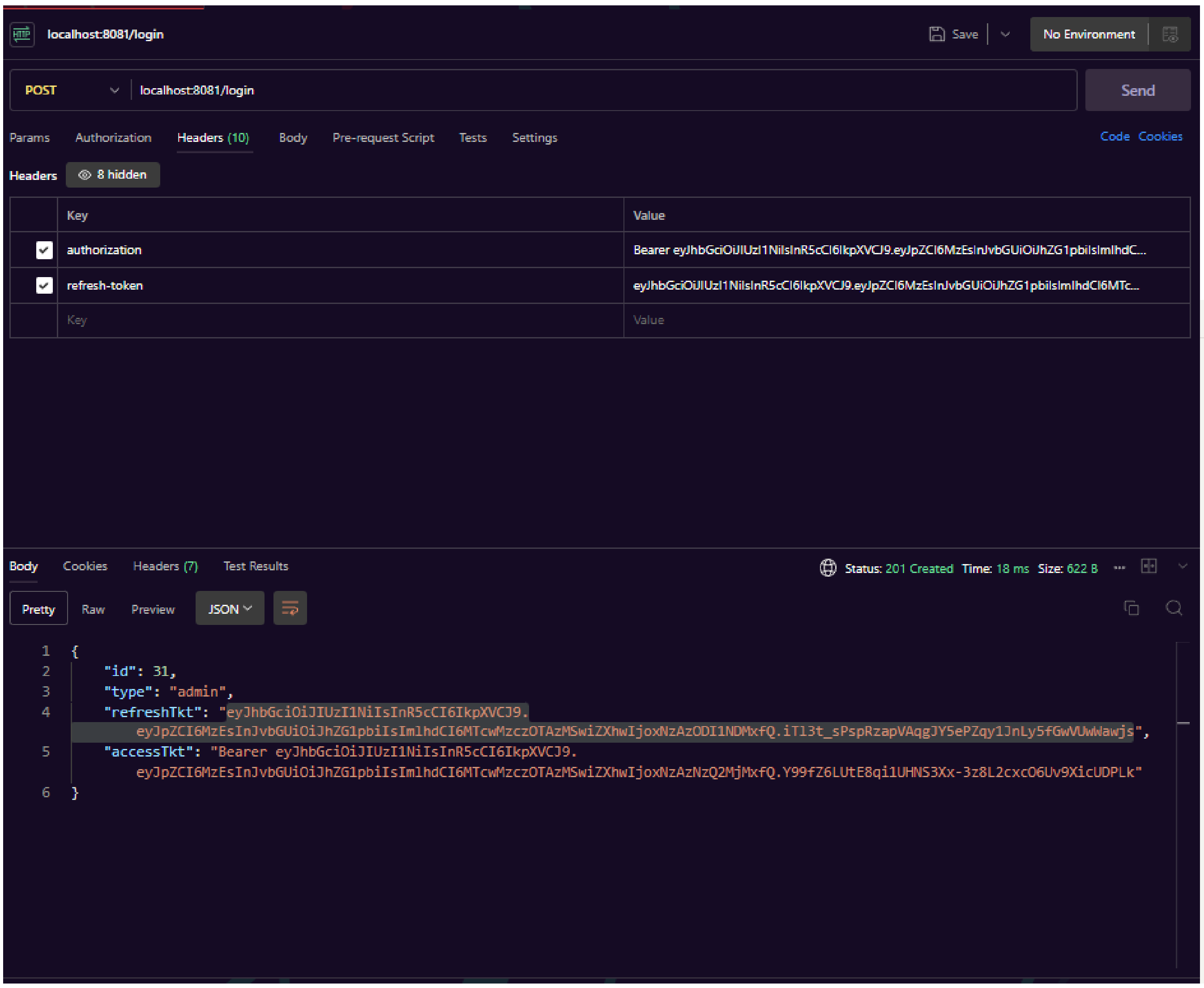
Body — Status: 201 Created Time: 18 ms Size: 622 B

```
{
    "id": 31,
    "type": "admin",
    "refreshTkt": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
        eyJpZCI6MzEsInJvbGUiOiJhZG1pbiIsImlhdCI6MTcwMzczOTAzMSwiZXhwIjoxNzAzODI1NDMxfQ.iTl3t_sPspRzapVAqgJY5ePZqy1JnLy5fGwVUwWawjs",
    "accessTkt": "Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
        eyJpZCI6MzEsInJvbGUiOiJhZG1pbiIsImlhdCI6MTcwMzczOTAzMSwiZXhwIjoxNzAzNzQ2MjMxfQ.Y99fZ6LUtE8qi1UHNS3Xx-3z8L2cxcO6Uv9XicUDPLk"
}
```

In the last Figure, the response returns the user signed tokens, his ID and type. Usually, using mechanisms like axiosinterceptors. However for now we can manually add in the header names :

1. authorization : *access Token*
2. refresh-token : *refreshtkn*

Like so :


POST localhost:8081/login

Headers

	Key	Value
✓	authorization	Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6MzEsInJvbGUiOiJhZG1pbiIsImlhdC...
✓	refresh-token	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6MzEsInJvbGUiOiJhZG1pbiIsImlhdCI6MTc...
	Key	Value

Status: 201 Created Time: 18 ms Size: 622 B

```json
{
    "id": 31,
    "type": "admin",
    "refreshTkt": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
        eyJpZCI6MzEsInJvbGUiOiJhZG1pbiIsImlhdCI6MTcwMzczOTAzMSwiZXhwIjoxNzAzODI1NDMxfQ.iTl3t_sPspRzapVAqgJY5ePZqy1JnLy5fGwVUwWawjs",
    "accessTkt": "Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
        eyJpZCI6MzEsInJvbGUiOiJhZG1pbiIsImlhdCI6MTcwMzczOTAzMSwiZXhwIjoxNzAzNzQ2MjMxfQ.Y99fZ6LUtE8qi1UHNS3Xx-3z8L2cxcO6Uv9XicUDPLk"
}
```

Once this is done, we are authenticated. Then we are able to access the protected routes.

User endpoints

1. Get All Users Endpoint

- **Endpoint:**
  - `GET /users`
- **Description:**
  - Retrieves information about all users. Accessible only to administrators.
- **Middleware:**
  - `verifyToken`: Verifies the authenticity of the access token.
  - `checkAdminRole`: Checks if the user has administrator privileges.
- **Response:**
  - If successful, the server responds with a status code of `200` (OK) and a JSON object containing user data.
- **Error Responses:**
  - If an error occurs during the process, the server responds with a status code of `500` (Internal Server Error).

2. Get User by ID Endpoint

- **Endpoint:**
  - `GET /users/:userid`
- **Description:**
  - Retrieves information about a specific user identified by `userid`.
- **Middleware:**
  - `verifyToken`: Verifies the authenticity of the access token.
  - `memberRestriction`: Restricts access to members only.
- **Response:**
  - If the user is found, the server responds with a status code of `200` (OK) and a JSON object containing user data.
  - If the user is not found, the server responds with a status code of `404` (Not Found).
- **Error Responses:**

- If an error occurs during the process, the server responds with a status code of `500` (Internal Server Error).

3. Delete User Endpoint

- **Endpoint:**
  - `DELETE /users/:userid`
- **Description:**
  - Deletes the user identified by `userid`. Admins can access any user, while members can only delete their own profile.
- **Middleware:**
  - `verifyToken`: Verifies the authenticity of the access token.
  - `memberRestriction`: Restricts access to members only.
- **Response:**
  - If the user is deleted, the server responds with a status code of `204` (No Content).
  - If the user is not found, the server responds with a status code of `404` (Not Found).
- **Error Responses:**
  - If an error occurs during the process, the server responds with an appropriate status code and message.

4. Add New User Endpoint

- **Endpoint:**
  - `POST /users`
- **Description:**
  - Adds a new user to the system.
- **Request Payload:**
  - The endpoint expects a JSON payload with the following fields:
    - `username` (string): The username of the new user.
    - `email` (string): The email address of the new user.
    - `password` (string): The password of the new user.
- **Response:**
  - If the user is added successfully, the server responds with a status code of `201` (Created) and a JSON object containing the new user's ID.
- **Error Responses:**
  - If an error occurs during the process or if required fields are missing, the server responds with an appropriate status code and message.

5. Update User Endpoint

- **Endpoint:**
  - `PUT /users/:userid`
- **Description:**
  - Updates the information of the user identified by `userid`. Admins can update any user, while members can only update their own profile.
- **Middleware:**
  - `verifyToken`: Verifies the authenticity of the access token.
  - `checkAdminRole`: Checks if the user has administrator privileges.
- **Request Payload:**
  - The endpoint expects a JSON payload with optional fields:
    - `username` (string): The updated username of the user.
    - `email` (string): The updated email address of the user.
    - `password` (string): The updated password of the user.
- **Response:**
  - If the user is updated successfully, the server responds with a status code of `200` (OK) and a JSON object containing the updated user information.
- **Error Responses:**
  - If an error occurs during the process or if the user is not found, the server responds with an appropriate status code and message.

Authentication Middleware:

- **`verifyToken` Middleware:**
  - Verifies the authenticity of the access token.
  - Responds with a status code of `401` (Unauthorized) if the token is invalid or missing.
- **`checkAdminRole` Middleware:**
  - Checks if the user has administrator privileges.
  - Responds with a status code of `403` (Forbidden) if the user is not an administrator.
- **`memberRestriction` Middleware:**
  - Restricts access to members only.
  - Responds with a status code of `403` (Forbidden) if the user is not a member.

These endpoints and middleware provide functionality for user management, including retrieval, creation, update, and deletion of user profiles. The authentication middleware ensures secure access to these operations based on user roles.

Task endpoints

1. Get All Tasks Endpoint

- **Endpoint:**
  - `GET /tasks`
- **Description:**
  - Retrieves information about all tasks.
- **Response:**
  - If successful, the server responds with a status code of `200` (OK) and a JSON object containing task data.
  - If an error occurs during the process, the server responds with an appropriate status code and message.

2. Get Task by ID Endpoint

- **Endpoint:**
  - `GET /tasks/:taskID`
- **Description:**
  - Retrieves information about a specific task identified by `taskID`.
- **Response:**
  - If the task is found, the server responds with a status code of `200` (OK) and a JSON object containing task data.
  - If the task is not found, the server responds with a status code of `404` (Not Found).
  - If an error occurs during the process, the server responds with an appropriate status code and message.

3. Create New Task Endpoint

- **Endpoint:**
  - `POST /tasks`
- **Description:**
  - Adds a new task to the system.
- **Request Payload:**
  - The endpoint expects a JSON payload with the following fields:
    - `title` (string): The title of the new task.
    - `description` (string): The description of the new task.
    - `points` (integer): The points associated with the new task.
- **Response:**
  - If the task is added successfully, the server responds with a status code of 201 (Created) and a JSON object containing the new task's ID.
  - If a task with the same title already exists, the server responds with a status code of 422 (Unprocessable Entity).
  - If an error occurs during the process, the server responds with a status code of `500` (Internal Server Error).

4. Delete Task Endpoint

- **Endpoint:**
  - `DELETE /tasks/:taskID`
- **Description:**
  - Deletes the task identified by `taskID`.
- **Response:**
  - If the task is deleted, the server responds with a status code of `200` (OK) and a JSON object indicating success.
  - If the task is not found, the server responds with a status code of `404` (Not Found).
  - If an error occurs during the process, the server responds with an appropriate status code and message.

5. Update Task Endpoint

- **Endpoint:**
  - `PUT /tasks/:taskID`
- **Description:**
  - Updates the information of the task identified by `taskID`.
- **Request Payload:**
  - The endpoint expects a JSON payload with optional fields:
    - `title` (string): The updated title of the task.
    - `description` (string): The updated description of the task.
    - `points` (integer): The updated points associated with the task.
- **Response:**
  - If the task is updated successfully, the server responds with a status code of `200` (OK) and a JSON object containing the updated task information.
  - If the task progress is not found, the server responds with a status code of `404` (Not Found).
  - If an error occurs during the process, the server responds with an appropriate status code and message.

Task Progress endpoints

1. Get Task Progress by ID Endpoint

- **Endpoint:**

  - `GET /progress/:progressID`
- **Description:**
  - Retrieves information about task progress identified by `progressID`.
- **Response:**
  - If the task progress is found, the server responds with a status code of 200 (OK) and a JSON object containing progress data.
  - If the task progress is not found, the server responds with a status code of 404 (Not Found).
  - If an error occurs during the process, the server responds with an appropriate status code and message.

2. Delete Task Progress Endpoint

- **Endpoint:**
  - `DELETE /progress/:progressID`
- **Description:**
  - Deletes the task progress identified by `progressID`.
- **Response:**
  - If the task progress is deleted, the server responds with a status code of 200 (OK) and a JSON object indicating success.
  - If the task progress is not found, the server responds with a status code of 404 (Not Found).
  - If an error occurs during the process, the server responds with an appropriate status code and message.

3. Create New Task Progress Endpoint

- **Endpoint:**
  - `POST /progress`
- **Description:**
  - Records the progress of a user on a specific task.
- **Request Payload:**
  - The endpoint expects a JSON payload with the following fields:
    - `user_id` (integer): The ID of the user making progress.
    - `task_id` (integer): The ID of the task for which progress is being recorded.
    - `completion_Date` (string): The date on which the task was completed.
    - `notes` (string, optional): Additional notes about the progress (default is 'Nill').
- **Response:**
  - If the progress is recorded successfully, the server responds with a status code of 201 (Created) and a JSON object containing the new progress ID.
  - If any required fields are missing, the server responds with a status code of 400 (Bad Request).
  - If an error occurs during the process, the server responds with a status code of 500 (Internal Server Error).

4. Update Task Progress Endpoint

- **Endpoint:**
  - `PUT /progress/:progressID`
- **Description:**
  - Updates the notes field of the task progress identified by `progressID`.
- **Request Payload:**
  - The endpoint expects a JSON payload with the following optional field:
    - `notes` (string): The updated notes about the progress.
- **Response:**
  - If the progress is updated successfully, the server responds with a status code of 200 (OK) and a JSON object containing the updated progress information.
  - If the task progress is not found, the server responds with a status code of 404 (Not Found).
  - If an error occurs during the process, the server responds with an appropriate status code and message.

Guild Endpoints

For the Guild Endpoints, it can be complicated, we are going to be going through the following :

1. Guild Roles, Permissions Middleware
2. Guild Basics
3. Guild Membership
4. Guild Roles and Permissions

Guild Roles and Permission Middleware

1. checkOwnerAccess Middleware:

Description:

- Checks if the user has owner access to the guild.
- Assumes user ID is stored in `req.userId` after token verification.

Usage:

- Apply this middleware to routes requiring owner access.

Error Handling:

- Responds with a 403 (Forbidden) status if the user lacks owner access.
- Responds with an appropriate status and message for other errors.

2. checkMemberAccess Middleware:

Description:

- Checks if the user is a member of the guild.
- Assumes user ID is stored in `req.userId` after token verification.

Usage:

- Apply this middleware to routes requiring membership access.

Error Handling:

- Responds with a 403 (Forbidden) status if the user is not a guild member.
- Responds with an appropriate status and message for other errors.

3. memberPerm Function:

Description:

- Checks if a user has a specific permission within a guild.
- Used to validate user permissions for various actions.

Parameters:

- `permID`: The ID of the permission being checked.
- `userId`: The ID of the user being checked.
- `guildId`: The ID of the guild in which the user's permission is checked.

Response:

- Returns `true` if the user has the specified permission; otherwise, returns `false`.
- Responds with `false` in case of errors.

4. checkGuildExistence Middleware:

Description:

- Checks if the guild exists based on the provided `guildID` parameter.

Usage:

- Apply this middleware to routes requiring guild existence.

Error Handling:

- Responds with a 404 (Not Found) status if the guild does not exist.
- Responds with an appropriate status and message for other errors.

5. **checkChannelMembership Middleware:**

Description:

- Checks if the user is a member of a specific channel within the guild.

Usage:

- Apply this middleware to routes requiring channel membership access.

Error Handling:

- Responds with a 403 (Forbidden) status if the user is not a member of the channel.
- Responds with an appropriate status and message for other errors.

Guild basics

1. Get All Guilds Endpoint:

```
###### Endpoint:
```

- `GET /guild`

Description:

- Retrieves information about all guilds.

Authorization:

- Requires a valid token (verified by `verifyToken` middleware).

Response:

- Responds with a JSON object containing information about all guilds.
- Responds with an appropriate status and message in case of errors.

2. Get Specific Guild By ID Endpoint:

Endpoint:

- `GET /guild/:guildID`

Description:

- Retrieves information about a specific guild based on the provided `guildID` parameter.

Authorization:

- Requires a valid token (verified by `verifyToken` middleware).
- Requires the guild to exist (verified by `checkGuildExistence` middleware).

Response:

- Responds with a JSON object containing information about the guild.
- Responds with a 404 (Not Found) status if the guild does not exist.
- Responds with an appropriate status and message in case of errors.

3. Get Banner Image of a Specific Guild Endpoint:

Endpoint:

- `GET /guild/:guildID/banner`

Description:

- Retrieves the banner image of a specific guild based on the provided `guildID` parameter.

Authorization:

- Requires a valid token (verified by `verifyToken` middleware).
- Requires the guild to exist (verified by `checkGuildExistence` middleware).

Response:

- Responds with the banner image file.
- Responds with a 404 (Not Found) status if the guild does not exist or has no banner.
- Responds with an appropriate status and message in case of errors.

4. Create New Guild Endpoint:

Endpoint:

- `POST /guild`

Description:

- Creates a new guild based on the provided information in the request body.

Authorization:

- Requires a valid token (verified by `verifyToken` middleware).

Request Payload:

- Expects a JSON payload with `guild_name`, `description`, `banner`, and `logo` fields.

Response:

- Responds with a JSON object containing the ID of the newly created guild.

- Responds with a 400 (Bad Request) status if required fields are missing.
- Responds with an appropriate status and message in case of errors.

5. Delete Guild Endpoint:

Endpoint:

- `DELETE /guild/:guildID`

Description:

- Deletes a guild based on the provided `guildID` parameter.

Authorization:

- Requires a valid token (verified by `verifyToken` middleware).
- Requires the guild to exist (verified by `checkGuildExistence` middleware).
- Requires owner access (verified by `checkOwnerAccess` middleware).

Response:

- Responds with a JSON object indicating the success of the deletion.
- Responds with a 404 (Not Found) status if the guild does not exist.
- Responds with a 403 (Forbidden) status if the user lacks owner access.
- Responds with an appropriate status and message in case of errors.

6. Edit Guild Information Endpoint:

Endpoint:

- `PUT /guild/:guildID`

Description:

- Edits information about a guild based on the provided `guildID` parameter.

Authorization:

- Requires a valid token (verified by `verifyToken` middleware).
- Requires the guild to exist (verified by `checkGuildExistence` middleware).
- Requires owner access (verified by `checkOwnerAccess` middleware).

Request Payload:

- Expects a JSON payload with optional fields: `guild_name`, `description`, `banner`, and `logo`.

Response:

- Responds with a JSON object containing information about the updated guild.
- Responds with a 404 (Not Found) status if the guild does not exist.
- Responds with a 403 (Forbidden) status if the user lacks owner access.
- Responds with an appropriate status and message in case of errors.

Guild Membership

1. Add Guild Member Endpoint:

Endpoint:

- `POST /guild/:guildID/member`

Description:

- Adds a new member to the guild based on the provided information in the request body.

Authorization:

- Requires a valid token (verified by `verifyToken` middleware).
- Requires the guild to exist (verified by `checkGuildExistence` middleware).
- Requires membership access (verified by `checkMemberAccess` middleware).

Request Payload:

- Expects a JSON payload with `member_id` and `nickname` fields.

Response:

- Responds with a JSON object indicating the success of adding the member.
- Responds with a 404 (Not Found) status if the guild does not exist.
- Responds with a 403 (Forbidden) status if the user lacks membership access.
- Responds with an appropriate status and message in case of errors.

2. Remove Guild Member Endpoint:

Endpoint:

- `DELETE /guild/:guildID/member`

Description:

- Removes a member from the guild based on the provided information in the request body.

Authorization:

- Requires a valid token (verified by `verifyToken` middleware).
- Requires the guild to exist (verified by `checkGuildExistence` middleware).
- Requires membership access (verified by `checkMemberAccess` middleware).

Request Payload:

- Expects a JSON payload with `member_id` field.

Response:

- Responds with a 204 (No Content) status indicating the success of removing the member.
- Responds with a 404 (Not Found) status if the guild does not exist.
- Responds with a 403 (Forbidden) status if the user lacks membership access.
- Responds with an appropriate status and message in case of errors.

3. Get Guild Members Endpoint:

Endpoint:

- `GET /guild/:guildID/member`

Description:

- Retrieves information about all members of a specific guild based on the provided `guildID` parameter.

Authorization:

- Requires a valid token (verified by `verifyToken` middleware).
- Requires the guild to exist (verified by `checkGuildExistence` middleware).
- Requires membership access (verified by `checkMemberAccess` middleware).

Response:

- Responds with a JSON object containing information about all members of the guild.
- Responds with a 404 (Not Found) status if the guild does not exist.
- Responds with a 403 (Forbidden) status if the user lacks membership access.
- Responds with an appropriate status and message in case of errors.

4. Get Guilds of a User Endpoint:

Endpoint:

- `GET /myGuild`

Description:

- Retrieves information about all guilds in which the user is a member.

Authorization:

- Requires a valid token (verified by `verifyToken` middleware).

Response:

- Responds with a JSON object containing information about all guilds of the user.
- Responds with an appropriate status and message in case of errors.

5. Get Guild Member Profile Endpoint:

Endpoint:

- `GET /guild/:guildID/member/:memberID`

Description:

- Retrieves information about a specific member of a guild based on the provided `guildID` and `memberID` parameters.

Authorization:

- Requires a valid token (verified by `verifyToken` middleware).
- Requires the guild to exist (verified by `checkGuildExistence` middleware).
- Requires membership access (verified by `checkMemberAccess` middleware).

Response:

- Responds with a JSON object containing information about the guild member.
- Responds with a 404 (Not Found) status if the guild does not exist.
- Responds with a 403 (Forbidden) status if the user lacks membership access.
- Responds with an appropriate status and message in case of errors.

6. Edit Guild Member Info Endpoint:

Endpoint:

- `PUT /guild/:guildID/myInfo`

Description:

- Edits the nickname property of the guild member.

Authorization:

- Requires a valid token (verified by `verifyToken` middleware).
- Requires the guild to exist (verified by `checkGuildExistence` middleware).
- Requires membership access (verified by `checkMemberAccess` middleware).

Request Payload:

- Expects a JSON payload with an optional `nickname` field.

Response:

- Responds with a JSON object indicating the success of updating the guild member's information.
- Responds with a 404 (Not Found) status if the guild does not exist.
- Responds with a 403 (Forbidden) status if the user lacks membership access.
- Responds with an appropriate status and message in case of errors.

Guild Role and Permission

1. Create New Role with Permissions

- **Endpoint:** `POST /guild/:guildID/role`
- **Description:** Create a new custom role for a guild along with specified permissions.
- **Request Body:**
  - `role_name` (string): Name of the new role.
  - `permission_string` (string): Comma-separated string of permission flags (e.g., "1,0,1,1").
- **Permissions:**
  - Requires the user to have the permission to manage roles (`Permission ID 10`).
- **Response:**
  - 201 Created: New role created successfully.
  - 403 Forbidden: Insufficient permissions.

2. Assign Users with Roles

- **Endpoint:** `PATCH /guild/:guildID/member/role`
- **Description:** Assign a specific role to a member in the guild.
- **Request Body:**
  - `member_id` (string): ID of the member to assign the role.
  - `new_role_id` (string): ID of the role to assign to the member.
- **Permissions:**
  - Requires the user to have the permission to manage roles (`Permission ID 11`).
- **Response:**
  - 200 OK: Role assigned successfully.
  - 403 Forbidden: Unauthorized permissions or member not found.

3. Delete Non-Default Role of A Guild

- **Endpoint:** `DELETE /guild/:guildID/role/:roleID`
- **Description:** Delete a non-default custom role of a guild.
- **Permissions:**
  - Requires the user to have the permission to manage roles (`Permission ID 10`).
- **Response:**
  - 204 No Content: Role deleted successfully.
  - 404 Not Found: Custom guild role not found.

4. Assign New Permission For Role

- **Endpoint:** `POST /guild/:guildID/role/permission`
- **Description:** Assign a new permission to a custom role in a guild.
- **Request Body:**
  - `role_id` (string): ID of the role to assign the permission.
  - `permission_id` (string): ID of the permission to assign.
- **Permissions:**
  - Requires the user to have the permission to manage roles (`Permission ID 10`).
- **Response:**
  - 200 OK: Permission assigned successfully.
  - 403 Forbidden: Unauthorized permissions or custom guild role not found.

5. Remove Permission For Role

- **Endpoint:** `DELETE /guild/:guildID/permission/role`
- **Description:** Remove a permission from a custom role in a guild.
- **Request Body:**
  - `role_id` (string): ID of the role to remove the permission.
  - `permission_id` (string): ID of the permission to remove.
- **Permissions:**
  - Requires the user to have the permission to manage roles (`Permission ID 10`).
- **Response:**
  - 204 No Content: Permission removed successfully.
  - 404 Not Found: Custom guild role not found.

Channel Endpoints

1. Create a New Channel for a Guild

- **Endpoint:** `POST /guild/:guildID/channel`
- **Description:** Create a new channel for a guild.
- **Request Body:**
  - `channel_type` (string, optional): Type of the channel (default: "chat").
  - `channel_name` (string): Name of the new channel.
- **Permissions:**
  - Requires the user to have the permission to create channels (`Permission ID 5`).
- **Response:**
  - 201 Created: New channel created successfully.
  - 403 Forbidden: Insufficient permissions.

2. Get All Channels of a Guild

- **Endpoint:** `GET /guild/:guildID/channel`
- **Description:** Get all channels belonging to a guild.
- **Permissions:**
  - Requires the user to have the permission to view channels (`Permission ID 4`).
- **Response:**
  - 200 OK: List of channels.
  - 404 Not Found: No channels created or unauthorized permissions.

3. Delete a Channel

- **Endpoint:** `DELETE /guild/:guildID/channel/:ChannelID`
- **Description:** Delete a channel belonging to a guild.
- **Permissions:**
  - Requires the user to have the permission to delete channels (`Permission ID 7`).
- **Response:**
  - 201 Created: Channel deleted successfully.
  - 403 Forbidden: Insufficient permissions.

4. Edit Channel Information

- **Endpoint:** `PUT /guild/:guildID/channel/:ChannelID`
- **Description:** Edit information (type or name) of a channel.
- **Request Body:**
  - `channel_type` (string, optional): New type of the channel.
  - `channel_name` (string, optional): New name of the channel.
- **Permissions:**
  - Requires the user to have the permission to edit channels (`Permission ID 6`).
- **Response:**
  - 201 Created: Channel information edited successfully.
  - 403 Forbidden: Insufficient permissions.

5. Add Guild Member to a Channel

- **Endpoint:** `POST /guild/:guildID/channel/:ChannelID`
- **Description:** Add a guild member to a specific channel.
- **Query Parameters:**
  - `userID` (string): ID of the user to add to the channel.
- **Permissions:**
  - Requires the user to have the permission to manage channel membership (`Permission ID 8`).
- **Response:**
  - 201 Created: User added to the channel successfully.
  - 403 Forbidden: Insufficient permissions.

6. Remove Guild Member from a Channel

- **Endpoint:** `DELETE /guild/:guildID/channel/:ChannelID`
- **Description:** Remove a guild member from a specific channel.
- **Query Parameters:**
  - `userID` (string): ID of the user to remove from the channel.
- **Permissions:**
  - Requires the user to have the permission to manage channel membership (`Permission ID 9`).
- **Response:**
  - 204 No Content: User removed from the channel successfully.
  - 403 Forbidden: Insufficient permissions.

7. Get Members of a Specific Channel

- **Endpoint:** `GET /guild/:guildID/channel/:ChannelID`
- **Description:** Get all members of a specific voice or text channel.
- **Query Parameters:**
  - `userID` (string, optional): ID of the user for additional checks.
- **Permissions:**
  - Requires the user to have the permission to view channel membership (`Permission ID 4`).
- **Response:**
  - 200 OK: List of channel members.
  - 404 Not Found: No members found for the channel or unauthorized permissions.

Quest Endpoints

For the Quest Endpoints we have :

1. Quest Basics
2. Quest Content
3. Quest Participation

Quest Basics

1. Get All Quests

- **Endpoint:** `GET /quests`
- **Description:** Retrieve information about all quests.
- **Permissions:**
  - None required.
- **Response:**
  - 200 OK: List of quests.
  - 404 Not Found: No quests available.

2. Get a Specific Quest by ID

- **Endpoint:** `GET /quests/:questID`
- **Description:** Retrieve information about a specific quest.
- **Parameters:**
  - `questID` (string): ID of the quest to retrieve.
- **Permissions:**
  - None required.
- **Response:**
  - 200 OK: Quest information.
  - 404 Not Found: Quest not found.

3. Create a New Quest

- **Endpoint:** `POST /quests`
- **Description:** Create a new quest.
- **Request Body:**
  - `title` (string): Title of the quest.
  - `description` (string): Description of the quest.
  - `reward_points` (number): Points rewarded for completing the quest.
  - `start_date` (string): Start date of the quest.
  - `end_date` (string): End date of the quest.
  - `guildID` (string): ID of the guild associated with the quest.
- **Query Parameters:**
  - `userID` (string): ID of the user creating the quest.
- **Permissions:**
  - Requires the user to be an active member of the guild associated with the quest.
- **Response:**
  - 201 Created: Quest created successfully.
  - 400 Bad Request: Missing required fields.
  - 403 Forbidden: User not a member of the server.

4. Delete a Quest

- **Endpoint:** `DELETE /quests/:questID`
- **Description:** Delete a quest.
- **Parameters:**
  - `questID` (string): ID of the quest to delete.
- **Permissions:**
  - None required.
- **Response:**
  - 200 OK: Quest deleted successfully.
  - 404 Not Found: Quest not found.

5. Edit Quest Information

- **Endpoint:** `PUT /quests/:questID`
- **Description:** Edit information (title, description, reward points, start date, end date) of a quest.
- **Parameters:**
  - `questID` (string): ID of the quest to edit.
- **Request Body:**
  - `title` (string, optional): New title of the quest.
  - `description` (string, optional): New description of the quest.
  - `reward_points` (number, optional): New points rewarded for completing the quest.
  - `start_date` (string, optional): New start date of the quest.
  - `end_date` (string, optional): New end date of the quest.
- **Permissions:**
  - None required.
- **Response:**
  - 201 Created: Quest information updated successfully.
  - 404 Not Found: Quest not found.

Quests Content

1. Add Quest Content

- **Endpoint:** `POST /quest-content`
- **Description:** Add content to a quest.
- **Request Body:**
  - `quest_id` (string): ID of the quest.
  - `level` (string): Level of the content.
  - `part` (string): Part of the content.

    - `content_type` (string): Type of content.
    - `content_description` (string, optional): Description of the content.
    - `pathway` (string, optional): Pathway associated with the content.
- **Permissions:**
    - None required.
- **Response:**
    - 201 Created: Content added successfully.

2. Get All Content for a Specific Quest

- **Endpoint:** `GET /quest-content/:questID`
- **Description:** Retrieve all content for a specific quest.
- **Parameters:**
    - `questID` (string): ID of the quest.
- **Permissions:**
    - None required.
- **Response:**
    - 200 OK: List of content for the quest.
    - 404 Not Found: Quest not found or no content available.

3. Get Content by Level and Part for a Specific Quest

- **Endpoint:** `GET /quest-content/:questID/:level`
- **Description:** Retrieve content by level and part for a specific quest.
- **Parameters:**
    - `questID` (string): ID of the quest.
    - `level` (string): Level of the content.
- **Permissions:**
    - None required.
- **Response:**
    - 200 OK: List of content for the specified level and part.
    - 404 Not Found: Quest not found or no content available for the specified level.

4. Remove Content by Content ID

- **Endpoint:** `DELETE /quest-content/:contentID`
- **Description:** Remove content by content ID.
- **Parameters:**
    - `contentID` (string): ID of the content to remove.
- **Permissions:**
    - None required.
- **Response:**
    - 200 OK: Content removed successfully.
    - 404 Not Found: Content not found.

5. Edit Content by Content ID

- **Endpoint:** `PUT /quest-content/:contentID`
- **Description:** Edit content details by content ID.
- **Parameters:**
    - `contentID` (string): ID of the content to edit.
- **Request Body:**
    - `level` (string, optional): New level of the content.
    - `part` (string, optional): New part of the content.
    - `content_type` (string, optional): New type of content.
    - `content_description` (string, optional): New description of the content.
    - `pathway` (string, optional): New pathway associated with the content.
- **Permissions:**
    - None required.
- **Response:**
    - 201 Created: Content edited successfully.
    - 404 Not Found: Content not found.

Quest Participation

1. Add User Quest Participation

- **Endpoint:** `POST /user-quest-participation`
- **Description:** Add a user's participation in a quest.
- **Request Body:**

- `user_id` (string): ID of the user.
  - `quest_id` (string): ID of the quest.
- **Query Parameter:**
  - `guildID` (string): ID of the guild to which the user belongs.
- **Permissions:**
  - None required.
- **Response:**
  - 201 Created: User participation added successfully.

2. Remove User Participation in a Quest

- **Endpoint:** `DELETE /user-quest-participation/:userID/:questID`
- **Description:** Remove a user's participation in a quest.
- **Parameters:**
  - `userID` (string): ID of the user.
  - `questID` (string): ID of the quest.
- **Permissions:**
  - None required.
- **Response:**
  - 200 OK: User participation removed successfully.

3. Set Completion Status of a User for a Quest

- **Endpoint:** `PUT /user-quest-completion/:userID/:questID`
- **Description:** Set the completion status of a user for a quest.
- **Parameters:**
  - `userID` (string): ID of the user.
  - `questID` (string): ID of the quest.
- **Request Body:**
  - `completion_date` (string): Completion date of the quest.
- **Permissions:**
  - None required.
- **Response:**
  - 200 OK: User completion status set successfully.

4. Get Quests of a User

- **Endpoint:** `GET /user-quests/:userID`
- **Description:** Get the quests in which a user is participating.
- **Parameters:**
  - `userID` (string): ID of the user.
- **Permissions:**
  - None required.
- **Response:**
  - 200 OK: List of quests in which the user is participating.
  - 404 Not Found: User not found or not participating in any quests.

SQL database

1. **User Table:**
   - `user_id` (Primary Key)
   - `username`
   - `email` (Unique)
   - `password`
   - `role` (ENUM: 'admin', 'member' - Default: 'member')
   - `creation_date`
2. **Task Table:**
   - `task_id` (Primary Key)
   - `title`
   - `description`
   - `points`
3. **TaskProgress Table:**
   - `progress_id` (Primary Key)
   - `user_id` (Foreign Key referencing User)
   - `task_id` (Foreign Key referencing Task)
   - `completion_date`
   - `notes`
4. **Guild_Roles Table:**
   - `role_id` (Primary Key)

    - `role_name` (Unique)
    - `default_role`
    - `guild_id` (Foreign Key referencing Guild)
5. **Channels Table:**
    - `channel_id` (Primary Key)
    - `channel_type`
    - `channel_name`
    - `guild_id` (Foreign Key referencing Guild)
6. **Guild_membership Table:**
    - `member_id` (Foreign Key referencing User)
    - `join_date`
    - `leave_date`
    - `role_id` (Default: 5)
    - `nickname`
    - `is_admin`
    - `guild_id` (Foreign Key referencing Guild)
7. **ChannelMembership Table:**
    - `member_id` (Foreign Key referencing Guild_membership)
    - `guild_id` (Foreign Key referencing Guild_membership)
    - `channel_id` (Foreign Key referencing Channels)
    - `join_date`
    - `leave_date`
8. **Permissions_Guild Table:**
    - `permission_id` (Primary Key)
    - `permission_name` (Unique)
9. **GuildRolesPermissions Table:**
    - `role_id` (Foreign Key referencing Guild_Roles)
    - `permission_id` (Foreign Key referencing Permissions_Guild)
10. **Quest Table:**
    - `quest_id` (Primary Key)
    - `title`
    - `description`
    - `reward_points`
    - `status` (Default: 'active')
    - `start_date`
    - `end_date`
    - `quest_master_id` (Foreign Key referencing User)
    - `guild_id` (Foreign Key referencing Guild)
11. **UserQuestParticipation Table:**

- `user_id` (Foreign Key referencing User)
- `quest_id` (Foreign Key referencing Quest)
- `start_date`
- `completion_date`

12. **QuestContentDetails Table:**

- `content_id` (Primary Key)
- `quest_id` (Foreign Key referencing Quest)
- `level`
- `part`
- `content_type`
- `content_description`
- `pathway`

13. **UserQuestProgress Table:**

- `user_id` (Foreign Key referencing User)
- `quest_id` (Foreign Key referencing QuestContentDetails)
- `level`
- `progress_percentage`

SQL Connection and Code Structure

For this section I aim to talk about how the code is used to connect and execute the SQL Commands ;

Async Template (`model`)

This template is designed to execute SQL queries asynchronously. It provides two main functions:

1. **template:**
    - Parameters:
        - `query`: The SQL query string.
        - `values`: An array of values to be used in the query (for parameterized queries).
        - `callback`: A callback function to handle the results or errors.
    - Usage:
        - Connects to the database.
        - Executes the query.
        - Calls the callback function with the results or an error.
        - Closes the database connection.
2. **async_template:**
    - Parameters:
        - `query`: The SQL query string.
        - `values`: An array of values to be used in the query (for parameterized queries).
    - Returns:
        - A Promise that resolves with the query results or rejects with an error.
    - Usage:
        - Connects to the database asynchronously.
        - Executes the query.
        - Resolves the Promise with the results or rejects with an error.
        - Closes the database connection.

SQL Connection

This module exports a function (`getConnection`) that creates and returns a MySQL database connection. It uses the `mysql` package.

- **getConnection:**
    - Returns:
        - A MySQL database connection object configured with the provided connection details.

```
const dbconnect = require('./configurations/databaseconfig');
const model = require('./path-to-model-file');

// Example usage with async/await
async function fetchData() {
  const query = 'SELECT * FROM users';
  try {
    const results = await model.async_template(query, []);
    console.log(results);
  } catch (error) {
    console.error(error);
  }
}

fetchData();
```

Why This method

1. **Asynchronous Nature:**
    - Node.js is designed to be non-blocking and asynchronous. Using asynchronous operations, like connecting to a database, allows your application to continue executing other tasks while waiting for the database operation to complete. This is crucial for maintaining high concurrency and responsiveness.
2. **Promises for Readability:**
    - The `async_template` function returns a Promise, which is a modern and readable way to handle asynchronous operations. It simplifies the code and makes it more understandable, especially when dealing with multiple asynchronous calls in sequence.
3. **Connection Pooling:**
    - The template doesn't create a new database connection for every query. Instead, it utilizes a single connection, which is more efficient. In a production environment, you might want to consider connection pooling to manage multiple connections effectively and improve performance.
4. **Error Handling:**
    - The template includes error handling at each step of the process. If an error occurs during the connection or query execution, it is properly handled, and the connection is closed. This helps prevent resource leaks and ensures that your application can gracefully recover from errors.
5. **Separation of Concerns:**
    - The code structure separates the database connection logic (`dbconnect`) from the query execution logic (`model`). This separation of concerns makes the code modular, easier to maintain, and promotes reusability.
6. **Parameterized Queries:**
    - The template supports parameterized queries (`values` array). This is a good practice for preventing SQL injection attacks and enhancing security.
7. **Connection Closing:**

- Properly closing the database connection after each query execution is important for managing system resources efficiently. The template ensures that the connection is closed, even in the case of an error.

8. **Promotes Best Practices:**
    - The code structure adheres to best practices, such as closing database connections, handling errors, and using asynchronous patterns. Following such practices contributes to the overall reliability and maintainability of the application.