Last Updated : [OCT 2024]

Project Overview

Purpose of the Project:

The goal of this project is to implement a **Kanban Board system** designed to centralize project management for volunteers working under two organizations: **Bear Brown & Company** and **Abecedarian LLC**. The system facilitates the efficient organization, assignment, and tracking of projects from start to completion, ensuring that tasks are appropriately allocated and progress is transparently monitored. This tool will serve as the primary platform for managing all volunteer-related tasks and will help ensure that the right resources are assigned to the right projects at the right time.

Key Technologies Used:

The project is built using web technologies, with a focus on scalability, maintainability, and efficient resource management. The key technologies include:

- **Node.js**: JavaScript runtime environment used for building the server-side logic.
- **Express.js**: Web framework for Node.js, enabling the creation of robust APIs for handling requests and managing responses.
- **MongoDB**: A NoSQL database for storing project, task, and user information, allowing flexibility in data structure.
- **AWS S3**: Amazon Simple Storage Service (S3) is used for managing file uploads and storage, particularly for any documents or attachments that may need to be associated with projects.
- **JWT (JSON Web Tokens)**: Used for secure user authentication and authorization.
- **Mongoose**: An ODM (Object Data Modeling) library for MongoDB and Node.js, used for interacting with the MongoDB database.

Goals:

The Kanban Board system addresses the following:

- **Centralized Project Management**: Provides a unified platform for managing projects, ensuring that all tasks and their progress are visible and organized.
- **Increased Accountability**: Assigns specific tasks to volunteers, tracking individual responsibility and task progress.
- **Improved Communication**: Supports in-app comments for seamless interaction between team members regarding tasks and project updates.
- **Efficient Resource Allocation**: Ensures that each project is appropriately staffed, preventing delays caused by misallocation or understaffing.
- **Real-time Progress Monitoring**: Provides real-time updates on task status, allowing project managers to promptly address delays or issues.

1.`app.js`:

Purpose:
The main setup file for the Express application, configuring middleware, database connection, routes, and API documentation.

Components:

1. **Environment Config**: Loads variables from `.env` for secure configurations.
2. **Middleware**:
   - **CORS**: Enables cross-origin requests.
   - **Body Parser**: Parses incoming JSON data.
3. **Database**: Connects to MongoDB via `mongoose` using `MONGODB_URI`.
4. **Swagger Documentation**: Provides API docs at `/api-docs` using Swagger UI.
5. **Routes**:
   - **Public Route**: `/api/auth` for user authentication (e.g., login, signup).
   - **Protected Routes** (JWT-secured):
     - `/api/users`: User management.
     - `/api/projects`: Project data.
     - `/api/team-groups`: Team groups.
     - `/api/tasks`: Task management.
     - `/api/comments`: Comments on tasks.
     - `/api/notifications`: User notifications.
     - `/api/permissions`: Role assignments.
6. **Export**: Exports `app` for use in `server.js`.

Dependencies:
`express`, `mongoose`, `cors`, `body-parser`, `dotenv`, `swagger-ui-express`, `authMiddleware`.

2.`server.js`:

Purpose:
The main entry point to start the server, loading the Express app and setting it to listen on the specified port.

Components:

1. **Port Configuration**: Sets the server port using `process.env.PORT` or defaults to `3000`.
2. **Server Initialization**: Starts the server with `app.listen()` and logs the listening port to confirm the server is running.

Dependencies

`app`: Imports the configured Express app from `app.js`.

`.env`:

The `.env` file stores sensitive configuration values and environment-specific settings, allowing easy configuration of the application without exposing credentials in code.

Environment Variables

1. **`PORT`**: Defines the port on which the server listens. Default is set to `3000`.
2. **`MONGODB_URI`**: The connection string for the MongoDB database. This URI is used to connect to MongoDB, storing all application data (users, tasks, projects, etc.).
3. **`JWT_SECRET`**: Secret key used to sign JSON Web Tokens (JWTs). Essential for securely handling user authentication. This should be a strong, unique string.
4. **`EMAIL_USER`**: Email address used to send outgoing emails (e.g., for password resets). This should be the account used by `nodemailer`.
5. **`EMAIL_PASS`**: Application Key Password for the email account specified in `EMAIL_USER`. Used to authenticate the email sender.
6. **`FRONTEND_URL`**: Base URL for the frontend application. This is used in links, such as password reset links, to redirect users to the frontend.
7. **`AWS_ACCESS_KEY_ID`**: AWS access key ID with permissions to interact with S3. Required for uploading files to AWS.
8. **`AWS_SECRET_ACCESS_KEY`**: Secret access key for AWS. Used alongside `AWS_ACCESS_KEY_ID` to authenticate AWS API requests.
9. **`AWS_REGION`**: Specifies the AWS region for S3 (e.g., `us-east-2`), defining the geographic location for AWS resources.
10. **`AWS_BUCKET_NAME`**: Name of the S3 bucket where files are stored, such as profile images.

Config:

1. `s3UploadConfig.js`

Purpose:
This module is responsible for configuring the file upload mechanism to Amazon S3. It uses `@aws-sdk/client-s3` for direct interaction with S3 and `multer-s3` to handle file uploads via `multer`. Files are stored in the S3 bucket specified by the environment variables.

Key Components:

- **S3 Client Initialization**: Creates an S3 client using credentials and the AWS region defined in the environment variables.
- **Multer Storage Configuration**: Uses `multer-s3` to define how files are stored:
    - **Bucket**: The S3 bucket name is specified via an environment variable (`AWS_BUCKET_NAME`).
    - **File Key**: Each uploaded file is assigned a unique name consisting of the timestamp and original filename.
    - **File Size Limit**: Limits the file size to 5 MB to prevent overly large uploads.

Exports:

- `upload`: Configured multer instance, set up with S3 as the storage backend.

Dependencies:

- `dotenv`: Loads environment variables from the `.env` file, ensuring sensitive AWS credentials are not hardcoded.
- `@aws-sdk/client-s3`: AWS SDK for JavaScript to interface with S3.
- `multer` and `multer-s3`: Middleware to manage file uploads to S3.

2. `swaggerConfig.js`

Purpose:
This file sets up the configuration for Swagger, an API documentation tool. It uses `swagger-jsdoc` to generate documentation from code and integrates model schemas using `mongoose-to-swagger` for seamless schema documentation.

Key Components:

- **Swagger Options**:
  - **API Metadata**: Defines general information about the API, including the title, description, and version.
  - **Tags**: Organizes endpoints into categories (e.g., Authentication, Authorization, Users, Projects) to simplify documentation.
  - **Servers**: Specifies the local development server URL.
  - **Schemas**: Converts Mongoose models (`User`, `Project`, `TeamGroup`, `Task`, `Comment`, `Notification`) into Swagger-compatible schemas using `mongoose-to-swagger`.
- **API Source Files**: Specifies that Swagger should look at files in `./routes/*.js` for API route definitions.

Exports:

- `swaggerDocs`: Configured Swagger documentation instance for generating and serving the API docs.

Dependencies:

- `swagger-jsdoc`: Generates Swagger documentation from JSDoc comments.
- `mongoose-to-swagger`: Converts Mongoose models into Swagger schemas.
- `Mongoose Models`: Imports models from the `models` folder, which are converted to Swagger schemas and added to the documentation.

Models:

1. `Notification.js`

Purpose:
The `Notification` model defines the structure of notifications that users receive. Notifications are used to inform users of actions or updates within the application.

Schema Fields:

- **`user_id`**: References the `User` who receives the notification (required).
- **`message`**: The notification content, describing the update or action (required, trimmed).
- **`is_read`**: Boolean indicating whether the notification has been read; defaults to `false`.
- **`created_at`**: Timestamp for when the notification was created; defaults to the current date.
- **`updated_at`**: Timestamp for when the notification was last updated; defaults to the current date.

Relationships:

- `user_id` establishes a reference to the `User` model, linking notifications to specific users.

2. `Project.js`

Purpose:
The `Project` model represents a project within the Kanban board system, containing details such as project description, dates, priority, and associated tasks, users, and team groups.

Schema Fields:

- **`name`**: The name of the project (required, trimmed).
- **`description`**: A brief description of the project (trimmed).
- **`start_date`** **&** **`end_date`**: Dates marking the project's planned start and end.
- **`priority`**: Project priority level, with possible values `Low`, `Medium`, or `High`; defaults to `Medium`.
- **`created_by`**: Reference to the `User` who created the project (required).
- **`assigned_users`**: Array of `User` references representing users assigned to the project.
- **`tasks`**: Array of `Task` references associated with the project.
- **`team_groups`**: Array of `TeamGroup` references associated with the project.
- **`created_at`**: Timestamp for when the project was created; defaults to the current date.
- **`updated_at`**: Timestamp for when the project was last updated; defaults to the current date.

Middleware:

- **Pre-save Hook**: A `pre-save` middleware automatically creates a default team group called "Main" when a new project is saved. This team group is assigned the project's users.

Relationships:

- `created_by`, `assigned_users`, `tasks`, and `team_groups` establish references to `User`, `Task`, and `TeamGroup` models, associating projects with users, tasks, and team groups.

3. `Comment.js`

Purpose:
The `Comment` model represents comments left by users on tasks, facilitating collaboration and feedback on specific tasks within a project.

Schema Fields:

- **`task_id`**: Reference to the `Task` associated with the comment (required).
- **`user_id`**: Reference to the `User` who created the comment (required).
- **`content`**: The text content of the comment (required, trimmed).
- **`created_at`**: Timestamp for when the comment was created; defaults to the current date.
- **`updated_at`**: Timestamp for when the comment was last updated; defaults to the current date.

Relationships:

- `task_id` and `user_id` establish references to `Task` and `User` models, connecting comments to specific tasks and users.

4. `Task.js`

Purpose:
The `Task` model represents individual tasks within projects, including details like task status, priority, and assignment information. Tasks are assigned to users and are optionally part of team groups within a project.

Schema Fields:

- **`name`**: The task's name (required, trimmed).
- **`description`**: Brief task description (trimmed).
- **`status`**: Task's current state, with options `Not Started`, `In Progress`, or `Completed`; defaults to `Not Started`.
- **`due_date`**: Deadline for task completion.
- **`priority`**: Task priority level, with options `Low`, `Medium`, or `High`; defaults to `Medium`.
- **`project_id`**: Reference to the associated `Project` (required).
- **`assigned_to`**: Reference to the `User` assigned to the task.
- **`team_group_id`**: Optional reference to the `TeamGroup` the task belongs to.
- **`comments`**: Array of `Comment` references associated with the task.
- **`created_at`**: Timestamp for when the task was created; defaults to the current date.
- **`updated_at`**: Timestamp for when the task was last updated; defaults to the current date.

Relationships:

- `project_id`, `assigned_to`, `team_group_id`, and `comments` establish references to `Project`, `User`, `TeamGroup`, and `Comment` models, connecting tasks to projects, assigned users, teams, and comments.

5. `TeamGroup.js`

Purpose:
The `TeamGroup` model represents groups of users within a project, facilitating collaborative task management and organization of team members within specific groups.

Schema Fields:

- **`name`**: The team group’s name (required).
- **`main_group`**: Boolean flag indicating if the group is the primary one for a project; defaults to `false`.
- **`project`**: Reference to the associated `Project` (required).
- **`tasks`**: Array of `Task` references assigned to the group.
- **`assigned_users`**: Array of `User` references representing users assigned to the team group.
- **`created_at`**: Timestamp for when the team group was created; defaults to the current date.
- **`updated_at`**: Timestamp for when the team group was last updated; defaults to the current date.

Relationships:

- `project`, `tasks`, and `assigned_users` establish references to `Project`, `Task`, and `User` models, associating team groups with projects, tasks, and team members.

6. User.js

Purpose:
The `User` model defines the structure for user accounts, including authentication, roles, and user-specific attributes like assigned projects, tasks, and notifications.

Schema Fields:

- **`first_name`** and **`last_name`**: User's full name (required, trimmed).
- **`email`**: Unique user email for login and identification (required, trimmed).
- **`password`**: Hashed password for authentication (required).
- **`role`**: User role, with options `Admin`, `Project Manager`, or `Volunteer` (required).
- **`reports_to`**: Array of `User` references indicating users to whom the current user reports.
- **`projects`**: Array of `Project` references the user is involved in.
- **`tasks`**: Array of `Task` references the user is assigned to.
- **`comments`**: Array of `Comment` references made by the user.
- **`notifications`**: Array of `Notification` references associated with the user.
- **`profile_image`**: URL of the user's profile image.
- **`resetPasswordToken`** and **`resetPasswordExpire`**: Fields for managing password reset functionality.
- **`created_at`** and **`updated_at`**: Timestamps for creation and last update.

Relationships:

- `reports_to`, `projects`, `tasks`, `comments`, and `notifications` establish connections to other `User`, `Project`, `Task`, `Comment`, and `Notification` models, linking users to associated records in the system.

Services:

1. `commentService.js`

Purpose:
Provides functions for managing comments, including creating, updating, retrieving, and deleting comments. It also maintains associations between comments and tasks.

Key Functions:

- **`getAllComments`**: Retrieves all comments from the database.
- **`getCommentById`**: Fetches a specific comment by its ID.
- **`createComment`**: Creates a new comment and associates it with a task by adding the comment ID to the task's `comments` array.
- **`updateComment`**: Updates an existing comment's content by ID.
- **`deleteComment`**: Deletes a comment and removes its reference from the associated task.

Dependencies:

- **`Comment`** model: For database operations related to comments.
- **`Task`** model: Used to update task comments when a comment is created or deleted(commentService).

2. `notificationService.js`

Purpose:
Handles operations for creating, retrieving, updating, and deleting notifications.

Key Functions:

- **`getAllNotifications`**: Retrieves all notifications.
- **`getNotificationById`**: Fetches a specific notification by ID.
- **`createNotification`**: Creates a new notification.
- **`updateNotification`**: Updates a notification by ID.
- **`deleteNotification`**: Deletes a notification by ID.

Dependencies:

- **`Notification`** model: For performing CRUD operations on notifications(notificationService).

3. `authService.js`

Purpose:
Provides authentication-related functionalities, such as user registration, login, and password reset.

Key Functions:

- `registerUser`: Registers a new user after verifying the email is unique and hashing the password.
- `loginUser`: Authenticates a user by verifying the email and password, then generates a JWT token upon successful login.
- `requestPasswordReset`: Generates a password reset token, saves it to the user's profile, and emails a reset link to the user.
- `resetPassword`: Resets the user's password using a valid token, hashing the new password and clearing the reset token.

Dependencies:

- `bcryptjs`: For password hashing and verification.
- `jsonwebtoken`: Used to generate JWT tokens for secure sessions.
- `crypto`: Generates and hashes tokens for password reset.
- `nodemailer`: Sends password reset emails to users(authService).

4. `authzService.js`

Purpose:
Handles authorization checks based on user roles, allowing role-based access control throughout the application.

Key Functions:

- `checkRole`: Verifies if the user's role is included in the list of allowed roles, throwing an error if permissions are insufficient.

Dependencies:

- `allowedRoles` parameter: Takes an array of roles to be checked against the user's role(authzService).

5. `teamGroupService.js`

Purpose:
Manages operations for team groups, such as creating, updating, and deleting groups, as well as adding or removing users.

Key Functions:

- `createTeamGroup`: Creates a new team group and associates it with a project.
- `getAllTeamGroups`: Retrieves all team groups, populated with associated projects, tasks, and users.
- `getTeamGroupById`: Fetches a team group by ID, including related project, tasks, and assigned users.
- `updateTeamGroup`: Updates team group details.
- `deleteTeamGroup`: Deletes a team group if it's not the main group and removes it from the project.
- `addUsersToTeamGroup`: Adds users to a non-main team group after verifying that they're already assigned to the project.
- `removeUsersFromTeamGroup`: Removes users from a non-main team group.

Dependencies:

- `TeamGroup` and `Project` models: For team group operations and associations with projects(teamGroupService).

6. `userService.js`

Purpose:
Handles user management operations, including creation, retrieval, updating, and deletion of users.

Key Functions:

- `getAllUsers`: Retrieves all users.
- `getUserById`: Fetches a specific user by ID and populates reporting relationships.
- `createUser`: Creates a new user.
- `updateUser`: Updates a user's information.
- `deleteUser`: Deletes a user by ID.
- `findUserByEmail`: Looks up a user by email for authentication purposes.

Dependencies:

- `User` model: For performing CRUD operations on users(userService).

7. `projectService.js`

Purpose:
Provides services for managing projects, including creation, updates, deletions, and user assignments.

Key Functions:

- `createProject`: Creates a new project.
- `getAllProjects`: Retrieves all projects, including associated users, tasks, and team groups.
- `getProjectById`: Fetches a project by ID, including related users, tasks, and team groups.
- `updateProject`: Updates project details.
- `deleteProject`: Deletes a project and its associated tasks, team groups, and comments.
- `addUserToProject`: Adds users to a project and ensures they're part of the main team group.
- `removeUserFromProject`: Removes users from a project and all associated team groups.

Dependencies:

- `Project`, `Task`, `TeamGroup`, `User` models: Manages project associations with users, tasks, and team groups(projectService).

8. `taskService.js`

Purpose:
Handles operations related to task management, including creation, updates, retrieval, and deletion of tasks.

Key Functions:

- `getAllTasks`: Retrieves tasks based on the user's role (Admin, Project Manager, or assigned user).
- `getTaskById`: Fetches a specific task by ID.
- `createTask`: Creates a new task and updates the corresponding team group and user assignment.
- `updateTask`: Updates task details, adjusting user task lists if reassigned.
- `deleteTask`: Deletes a task, removing references from associated users and team groups.

Dependencies:

- `Task`, `Project`, `TeamGroup`, `User` models: Manages task-related associations and updates(taskService).

Middleware:

1. `authMiddleware.js`

Purpose:
This middleware file handles **user authentication** by verifying JSON Web Tokens (JWTs) to ensure that incoming requests are from authenticated users.

Key Functionality:

- **`authenticate`**: Middleware function that checks the `Authorization` header for a JWT.
  - **Token Extraction**: Retrieves the token from the `Authorization` header, expecting the format `Bearer <token>`.
  - **Token Verification**: Verifies the token using the secret stored in the environment variable (`JWT_SECRET`). If valid, it decodes the token payload.
  - **Attaches User Info**: Adds the decoded user information to `req.user` for use in subsequent middleware or routes.
  - **Error Handling**: Responds with `401 Unauthorized` if no token is provided, or `400 Bad Request` if the token is invalid.

Dependencies:

- **`jsonwebtoken`**: Used for JWT verification and decoding.

2. `authzMiddleware.js`

Purpose:
This middleware enforces **role-based authorization** by ensuring that users have the necessary role permissions to access specific routes.

Key Functionality:

- **`requireRole`**: Middleware function generator that takes a `requiredRole` parameter.
  - **Role Checking**: Uses `authzService.checkRole()` to verify that `req.user` (set by `authMiddleware`) has the appropriate role to access the endpoint.
  - **Error Handling**: If the user's role doesn't match `requiredRole`, the middleware responds with a `403 Forbidden` status and an error message.

Dependencies:

- **`authzService`**: A service that handles role-checking logic to determine if a user has the required role. This abstracts the role-checking functionality, making it reusable across the application.

Controllers:

1. `authController.js`

Purpose:
This controller handles **authentication** functionalities such as user registration, login, and password reset operations. It relies on `authService` to manage the underlying logic.

Key Functions:

- **`registerUser`**: Registers a new user.
  - Calls `authService.registerUser()` with `req.body` data.
  - **Response**: Returns `201 Created` status with a success message and the created user object.
- **`loginUser`**: Logs in an existing user.
  - Calls `authService.loginUser()` with the user's email and password from `req.body`.
  - Set the token in the `Authorization` header.
  - **Response**: Returns `200 OK` with the generated token and user details.
- **`requestPasswordReset`**: Initiates a password reset process.
  - Calls `authService.requestPasswordReset()` with the user's email.
  - **Response**: Returns `200 OK` with a message indicating that the reset link has been sent (if successful).
- **`resetPassword`**: Completes the password reset process.
  - Calls `authService.resetPassword()` with the reset token and new password.
  - **Response**: Returns `200 OK` with a message confirming the password reset.

Dependencies:

- **`authService`**: Provides the underlying logic for authentication operations.

2. `authzController.js`

Purpose:
This controller is responsible for **role assignment and authorization management**. It uses `userService` to fetch user data and `authzService` for role-based access control.

Key Functions:

- **`assignRole`**: Assigns a specific role to a user.
  - Retrieves the user by ID using `userService.getUserById(req.params.id)`.
  - Checks if the logged-in user has the “Admin” role using `authzService.checkRole()`.
  - If authorized, updates the user's role based on `req.body.role`.
  - **Response**: Returns `200 OK` with a success message and the updated user data.

Dependencies:

- **`userService`**: Fetches user details from the database.
- **`authzService`**: Handles role-based access checks to verify if the logged-in user has the necessary permissions.

3. `commentController.js`

Purpose:
This controller manages **comment operations** associated with tasks, allowing users to create, read, update, and delete comments.

Key Functions:

- **`getAllComments`**: Retrieves all comments.
  - Calls `commentService.getAllComments()` to fetch all comment records.
  - **Response**: Returns a list of comments or a `500 Internal Server Error` message if an issue occurs.
- **`getCommentById`**: Retrieves a specific comment by its ID.
  - Calls `commentService.getCommentById()` with the comment ID from `req.params`.
  - **Response**: Returns the found comment or `404 Not Found` if no matching comment exists.
- **`createComment`**: Creates a new comment.
  - Prepares `commentData` from `req.body` (task ID, user ID, and content).
  - Calls `commentService.createComment()` with `commentData`.
  - **Response**: Returns `201 Created` with the newly created comment, or `400 Bad Request` if the request fails.
- **`updateComment`**: Updates a comment's content, checking ownership.
  - Verifies if the current user is the comment owner before allowing updates.
  - Calls `commentService.updateComment()` with the comment ID and new content from `req.body`.
  - **Response**: Returns the updated comment or `403 Forbidden` if the user is unauthorized.
- **`deleteComment`**: Deletes a comment, checking ownership or admin rights.
  - Ensures that the user is either the comment owner or an Admin before deletion.
  - Calls `commentService.deleteComment()` with the comment ID.
  - **Response**: Returns a success message or `403 Forbidden` if unauthorized.

Dependencies:

- **`commentService`**: Provides core logic for CRUD operations.
- **`Comment` Model**: Used to fetch specific comment data for ownership validation.

4. `notificationController.js`

Purpose:
This controller manages **notification operations**, allowing the application to create, read, update, and delete notifications for users.

Key Functions:

- **`getAllNotifications`**: Retrieves all notifications.
    - Calls `notificationService.getAllNotifications()` to get a list of all notifications.
    - **Response**: Returns a list of notifications or a `500 Internal Server Error` if an error occurs.
- **`getNotificationById`**: Retrieves a specific notification by its ID.
    - Calls `notificationService.getNotificationById()` using the notification ID from `req.params`.
    - **Response**: Returns the notification or `404 Not Found` if it does not exist.
- **`createNotification`**: Creates a new notification.
    - Collects `notificationData` (user ID, message, read status) from `req.body`.
    - Calls `notificationService.createNotification()` with the provided data.
    - **Response**: Returns `201 Created` with the new notification, or `400 Bad Request` if the request fails.
- **`updateNotification`**: Updates the `is_read` status of a notification.
    - Calls `notificationService.updateNotification()` with the notification ID and new read status from `req.body`.
    - **Response**: Returns the updated notification, or `400 Bad Request` if an issue arises.
- **`deleteNotification`**: Deletes a notification by its ID.
    - Calls `notificationService.deleteNotification()` with the notification ID.
    - **Response**: Returns a success message or `500 Internal Server Error` if an error occurs.

Dependencies:

- **`notificationService`**: Contains the logic for CRUD operations on notifications.

5. `projectController.js`

Purpose:
This controller manages **project operations** such as creating, reading, updating, and deleting projects. Role-based access is enforced for certain actions, requiring users to have “Admin” or “Project Manager” roles.

Key Functions:

- **`createProject`**: Creates a new project.
    - **Authorization**: Checks if the user is an Admin or Project Manager.
    - Collects project details from `req.body`, including assigned users and tasks.
    - Calls `projectService.createProject()` with the collected data.
    - **Response**: Returns `201 Created` with the new project or `400 Bad Request` if unsuccessful.
- **`getAllProjects`**: Retrieves all projects.
    - Calls `projectService.getAllProjects()`.
    - **Response**: Returns a list of all projects or `500 Internal Server Error` if an error occurs.
- **`getProjectById`**: Retrieves a specific project by its ID.
    - Calls `projectService.getProjectById()` with the project ID.
    - **Response**: Returns the project or `404 Not Found` if it doesn’t exist.
- **`updateProject`**: Updates project details.
    - **Authorization**: Ensures the user is an Admin or Project Manager.
    - Updates relevant fields with data from `req.body`.
    - Calls `projectService.updateProject()` with the project and update data.
    - **Response**: Returns the updated project or `400 Bad Request` on failure.
- **`deleteProject`**: Deletes a specific project by ID.
    - **Authorization**: Ensures the user is an Admin or Project Manager.
    - Calls `projectService.deleteProject()` with the project ID.
    - **Response**: Returns a success message or `500 Internal Server Error` if an issue arises.
- **`addUserToProject`**: Adds users to a project.
    - **Authorization**: Ensures the user is an Admin or Project Manager.
    - Calls `projectService.addUserToProject()` with project ID and user IDs from `req.body`.
    - **Response**: Returns a success message and updated project.
- **`removeUserFromProject`**: Removes users from a project.
    - **Authorization**: Ensures the user is an Admin or Project Manager.
    - Calls `projectService.removeUserFromProject()` with project ID and user IDs from `req.body`.
    - **Response**: Returns a success message and updated project.

Dependencies:

- **`projectService`** and **`authzService`**.

6. `taskController.js`

Purpose:
This controller manages **task operations** including creation, retrieval, updates, and deletion of tasks. Role-based permissions control access to various task operations.

Key Functions:

- **`getAllTasks`**: Retrieves all tasks.
  - **Authorization**: Restricted to Admin and Project Managers.
  - Calls `taskService.getAllTasks()` with the user.
  - **Response**: Returns a list of tasks or `500 Internal Server Error`.
- **`getTaskById`**: Retrieves a specific task by ID.
  - **Authorization**: Allows Admin, Project Managers, and Volunteers.
  - Calls `taskService.getTaskById()` with the task ID and user.
  - **Response**: Returns the task or `404 Not Found`.
- **`createTask`**: Creates a new task.
  - **Authorization**: Restricted to Admin and Project Managers.
  - Collects task details from `req.body`.
  - Calls `taskService.createTask()` with the collected data.
  - **Response**: Returns `201 Created` with the new task or `400 Bad Request`.
- **`updateTask`**: Updates task details.
  - **Authorization**: Allows Admin, Project Managers, and Volunteers.
  - Calls `taskService.updateTask()` with the task ID, update data, and user.
  - **Response**: Returns the updated task or `400 Bad Request` if the request fails.
- **`deleteTask`**: Deletes a specific task by ID.
  - **Authorization**: Restricted to Admin and Project Managers.
  - Calls `taskService.deleteTask()` with the task ID.
  - **Response**: Returns a success message or `500 Internal Server Error`.

Dependencies:

- **`taskService`**: Contains core logic for task CRUD operations.
- **`authzService`**: Ensures the user has the appropriate role for each action.

7. `teamGroupController.js`

Purpose:
This controller manages **team group operations** such as creating, retrieving, updating, and deleting team groups within a project. Role-based access is enforced to ensure only authorized users can perform specific actions.

Key Functions:

- **`createTeamGroup`**: Creates a new team group.
  - **Authorization**: Checks if the user is an Admin or Project Manager.
  - Collects `teamGroupData` from `req.body` with optional fields for assigned users and tasks.
  - Calls `teamGroupService.createTeamGroup()` with `teamGroupData`.
  - **Response**: Returns `201 Created` with the new team group or `400 Bad Request` if creation fails.
- **`getAllTeamGroups`**: Retrieves all team groups.
  - Calls `teamGroupService.getAllTeamGroups()`.
  - **Response**: Returns a list of team groups or `500 Internal Server Error`.
- **`getTeamGroupById`**: Retrieves a specific team group by ID.
  - Calls `teamGroupService.getTeamGroupById()` with the team group ID.
  - **Response**: Returns the team group or proceeds to the next middleware.
- **`updateTeamGroup`**: Updates team group details.
  - **Authorization**: Ensures the user is an Admin or Project Manager.
  - Calls `teamGroupService.updateTeamGroup()` with the team group and update data from `req.body`.
  - **Response**: Returns the updated team group or `400 Bad Request` on failure.
- **`deleteTeamGroup`**: Deletes a specific team group.
  - **Authorization**: Ensures the user is an Admin or Project Manager.
  - Calls `teamGroupService.deleteTeamGroup()` with the team group ID.
  - **Response**: Returns a success message or `500 Internal Server Error`.
- **`addUsersToTeamGroup`**: Adds users to a team group.
  - **Authorization**: Ensures the user is an Admin or Project Manager.
  - Calls `teamGroupService.addUsersToTeamGroup()` with the team group ID and user IDs.
  - **Response**: Returns a success message and updated team group.
- **`removeUsersFromTeamGroup`**: Removes users from a team group.
  - **Authorization**: Ensures the user is an Admin or Project Manager.
  - Calls `teamGroupService.removeUsersFromTeamGroup()` with the team group ID and user IDs.
  - **Response**: Returns a success message and updated team group.

Dependencies:

- **`teamGroupService`**: Handles core logic for managing team groups.
- **`authzService`**: Ensures only authorized users can manage team groups.

8. `userController.js`

Purpose:
This controller handles **user management operations**, including CRUD operations for user profiles and profile image uploads. It leverages S3 for profile image storage and integrates role-based access control where necessary.

Key Functions:

- **`getAllUsers`**: Retrieves all users.
    - Calls `userService.getAllUsers()`.
    - **Response**: Returns a list of users or `500 Internal Server Error`.
- **`getUserById`**: Retrieves a specific user by ID.
    - Calls `userService.getUserById()` with the user ID from `req.params`.
    - **Response**: Returns the user or `404 Not Found` if not found.
- **`createUser`**: Creates a new user.
    - Collects `userData` from `req.body`, with fields for first and last name, email, password, role, and profile image.
    - Calls `userService.createUser()` with the collected data.
    - **Response**: Returns `201 Created` with the new user, or `400 Bad Request` if the email is already in use.
- **`updateUser`**: Updates a user's profile information and uploads a new profile image if provided.
    - **Image Handling**: Uses S3 to store new profile images, deleting the old image if it exists.
    - Calls `userService.updateUser()` with `updateData` from `req.body` and the updated image location.
    - **Response**: Returns the updated user or `400 Bad Request` if an error occurs.
- **`deleteUser`**: Deletes a user by ID.
    - Calls `userService.deleteUser()` with the user ID.
    - **Response**: Returns a success message or `500 Internal Server Error` if deletion fails.

Dependencies:

- **`userService`**: Handles core logic for user CRUD operations.
- **`s3UploadConfig`**: Provides the S3 upload configuration for profile images.
- **`@aws-sdk/client-s3`**: Used to delete old profile images from S3 if they are updated.

Routes:

1. `commentRoute.js`

Purpose:
Defines routes for managing comments, including creating, updating, retrieving, and deleting comments. Integrates with `commentController` for handling requests and responses.

Key Routes:

- **`GET /api/comments`**: Retrieves all comments.
- **`GET /api/comments/:id`**: Retrieves a specific comment by its ID.
- **`POST /api/comments`**: Creates a new comment.
- **`PATCH /api/comments/:id`**: Updates a comment by its ID.
- **`DELETE /api/comments/:id`**: Deletes a comment by its ID.

2. `notificationRoute.js`

Purpose:
Handles routes related to notifications, allowing the creation, retrieval, updating, and deletion of notification records.

Key Routes:

- **`GET /api/notifications`**: Retrieves all notifications.
- **`GET /api/notifications/:id`**: Retrieves a notification by its ID.
- **`POST /api/notifications`**: Creates a new notification.
- **`PATCH /api/notifications/:id`**: Updates a notification by its ID.
- **`DELETE /api/notifications/:id`**: Deletes a notification by its ID.

3. `authRoute.js`

Purpose:
Defines routes for authentication, including user registration, login, and password reset functionalities. All routes delegate to `authController`.

Key Routes:

- **`POST /auth/register`**: Registers a new user.
- **`POST /auth/login`**: Logs in a user.
- **`POST /auth/forgot-password`**: Requests a password reset.
- **`POST /auth/reset-password`**: Resets a user’s password.

4. `authzRoute.js`

Purpose:
Manages routes for authorization operations, specifically assigning roles to users. Protected by JWT authentication and role-based access using `authMiddleware`.

Key Routes:

- **`POST /assign-role/:id`**: Assigns a specified role to a user by ID, restricted to Admins.

5. `teamGroupRoutes.js`

Purpose:
Defines routes for managing team groups, allowing authorized users to create, update, retrieve, and delete team groups, as well as add or remove users from groups.

Key Routes:

- `POST /team-groups`: Creates a new team group.
- `GET /team-groups`: Retrieves all team groups.
- `GET /team-groups/:id`: Retrieves a team group by its ID.
- `PATCH /team-groups/:id`: Updates a team group.
- `DELETE /team-groups/:id`: Deletes a team group.
- `POST /team-groups/:id/add-users`: Adds users to a team group.
- `POST /team-groups/:id/remove-users`: Removes users from a team group.

Swagger Documentation:
All routes include Swagger documentation, describing expected input, output, and error responses for easier API testing and documentation(teamGroupRoutes).

6. `userRoute.js`

Purpose:
Defines routes for managing user accounts, including creation, updating, retrieval, and deletion of user records.

Key Routes:

- `POST /users`: Creates a new user.
- `GET /users`: Retrieves a list of all users.
- `GET /users/:id`: Retrieves a specific user by ID.
- `PATCH /users/:id`: Updates user information.
- `DELETE /users/:id`: Deletes a user by ID.

7. `projectRoute.js`

Purpose:
Manages routes for project operations, including creating, updating, and deleting projects, as well as adding or removing users from projects.

Key Routes:

- `POST /projects`: Creates a new project.
- `GET /projects`: Retrieves all projects.
- `GET /projects/:id`: Retrieves a project by ID.
- `PATCH /projects/:id`: Updates a project's details.
- `DELETE /projects/:id`: Deletes a project by ID.
- `POST /projects/:id/add-users`: Adds users to a project.
- `POST /projects/:id/remove-users`: Removes users from a project.

8. `taskRoute.js`

Purpose:
Handles routes for task management, allowing authorized users to create, update, retrieve, and delete tasks within a project.

Key Routes:

- `GET /api/tasks`: Retrieves all tasks.
- `GET /api/tasks/:id`: Retrieves a specific task by ID.
- `POST /api/tasks`: Creates a new task.
- `PATCH /api/tasks/:id`: Updates a task by ID.
- `DELETE /api/tasks/:id`: Deletes a task by ID.