Overview of Geet Backend Documentation

December 1, 2024

1 Introduction

This is the backend documentation for Geet Project.

2 Overview of the technologies used

- **Node.js** - Serves as the primary runtime environment for executing backend logic.
- **Express.js** - Provides a robust web framework to handle server-side routing and middleware functions.
- **Nest.js** - Framework used for testingfor testing
- **Third-Party Software**:
  - **Amazon S3** - Facilitates efficient and scalable file uploads.
  - **PostgreSQL Database (PSQL)** - Manages and stores all relevant application data.

3 Modules

Modules - Our code is organized into the following modules:

- **Config** - Contains all necessary database schema and configuration files.
- **Controllers** - Handles the logic for processing requests and interacting with the database.
- **Tests** - Includes automated test cases to ensure code reliability and correctness.
- **Helpers** - Provides utility functions and reusable code to support various modules.
- **Middlewares** - Contains middleware functions for tasks such as authentication and request validation.
- **Routes** - Manages API endpoint definitions and their associated controllers.

3.1 Config

- **db.ts** - This file manages database connections for the application. It includes functions to connect to both MongoDB and PostgreSQL, as well as a dedicated function for connecting to a test PostgreSQL database.
  - **MongoDB Connection**: The `connectMongoDB` function establishes a connection to MongoDB using the Mongoose library and ensures the database is ready for operations.
  - **PostgreSQL Connection**: The `connectPostgresDB` function handles the PostgreSQL connection by configuring a connection pool with environment variables for credentials, host, database, and SSL settings. It tests the connection upon initialization.

  - **Test PostgreSQL Connection**: The `connectTestPostgresDB` function allows connections to a dedicated test database using similar configurations, ensuring the application can operate in a test environment without affecting production data.
  - **Global Pool Access**: The `getPool` function provides a way to access the initialized PostgreSQL pool globally within the application.

- **s3.ts** - This file handles the connection and configuration for Supabase's S3-like storage service
  - **S3 Client Initialization**: The `connectS3` function initializes the S3 client using the AWS SDK, configured with environment variables for region, endpoint URL, and credentials. The `forcePathStyle` option is enabled to ensure compatibility with Supabase's S3 implementation.
  - **Global S3 Client Access**: The `getS3Client` function provides a globally accessible way to retrieve the initialized S3 client, ensuring that other parts of the application can interact with Supabase storage seamlessly.
  - **Error Handling**: The `connectS3` function includes robust error handling, logging connection errors, and terminating the process if the connection fails.

- **certification.ts** - Defines the schema for storing and managing teacher certifications in the database.
- **comments.ts** - Creates the schema for user comments on courses or posts.
- **courseDocuments.ts** - Defines the schema for storing course-related documents.
- **courseEnrollments.ts** - Creates the schema for managing student enrollments in courses.
- **courses.ts** - Defines the schema for course details and their associated properties.
- **notifications.ts** - Creates the schema for notifications related to system events.
- **notificationTokens.ts** - Defines the schema for storing and managing notification tokens.
- **pairingSetup.ts** - Creates the schema for setting up pairings between students and mentors.
- **postDocuments.ts** - Defines the schema for documents associated with user posts.
- **posts.ts** - Creates the schema for storing user posts and their metadata.
- **refreshToken.ts** - Defines the schema for refresh tokens used in authentication.
- **user.ts** - Creates the schema for the "Users" table in PostgreSQL.
- **userNotes.ts** - Defines the schema for user-generated notes related to courses or posts.

3.2 Controllers

Controllers handle the bulk of the application's logic and decision-making.

- **authController.ts** - Handles user authentication, including login, registration, and token management.
- **commentController.ts** - Manages logic related to user comments, such as adding, editing, and deleting comments.
- **courseController.ts** - Processes operations related to course management, including creating, updating, and deleting course details.
- **mentorController.ts** - Handles mentor-specific functionality, such as managing mentor profiles and assignments.
- **notificationController.ts** - Processes logic for managing and sending notifications to users.

- **pairingController.ts** - Handles the pairing and unpairing of mentors and students, ensuring correct relationships are established and updated.
- **postController.ts** - Manages user posts, including creating, editing, and deleting posts, as well as related logic.
- **userController.ts** - Oversees user-related operations, such as profile updates, role management, and data retrieval.

3.3 Routes

Routes define the API endpoints for the application and they are REST-compliant interactions between the client and server. These routes are equipped with type validation using libraries like `express-validator`, which validate incoming requests at the field level for parameters and payloads.

- **adminRoutes.ts** - Defines routes for managing administrative tasks, including system configurations and privileged operations.
- **authRoutes.ts** - Handles authentication-related routes such as login, registration, and token refresh.
- **commentRoutes.ts** - Provides endpoints for adding, editing, deleting, and retrieving user comments on posts or courses.
- **courseRoutes.ts** - Manages routes for creating, updating, deleting, and retrieving course details and associated information.
- **mentorRoutes.ts** - Defines routes for managing mentor-specific actions, including profile updates and mentor assignments.
- **notificationRoutes.ts** - Provides endpoints for sending and retrieving user notifications.
- **pairingRoutes.ts** - Handles routes for pairing and unpairing mentors and students, facilitating relationship management.
- **postRoutes.ts** - Manages routes for creating, editing, deleting, and retrieving user posts and related data.
- **userRoutes.ts** - Defines routes for user management, including profile updates, role assignments, and data retrieval.

3.4 Test

The test suite ensures the correctness and reliability of the application's controllers by providing comprehensive test coverage for each feature. Each `spec.ts` file corresponds to a specific controller, validating its functionality through a series of automated tests. For instance, `authController.spec.ts` tests the authentication logic, while `courseController.spec.ts` ensures the accuracy of course-related operations. This one-to-one mapping maintains modular and organized testing, with each controller's logic isolated and verified independently.

The tests simulate the application's behavior by creating connecting to a test database. The test database acts as a temporary replica of the production database which is destroyed completely after the tests are done. This ensures that tests do not interfere with actual data or configurations. Before each test, the required tables and data are initialized in the database, providing a controlled and reproducible environment. After each test, the database is destroyed, ensuring no residual data carries over between tests. This approach maintains data integrity, prevents test contamination, and ensures accurate results for all test scenarios.

While this approach provides a robust foundation for verifying the application's functionality, it is important to note that our test coverage was not exhaustive due to the limited time available. Certain edge cases and complex scenarios may not have been fully addressed. Despite these limitations, the implemented test suite focuses on critical paths in the application, ensuring that our core features are solid for future enhancements.

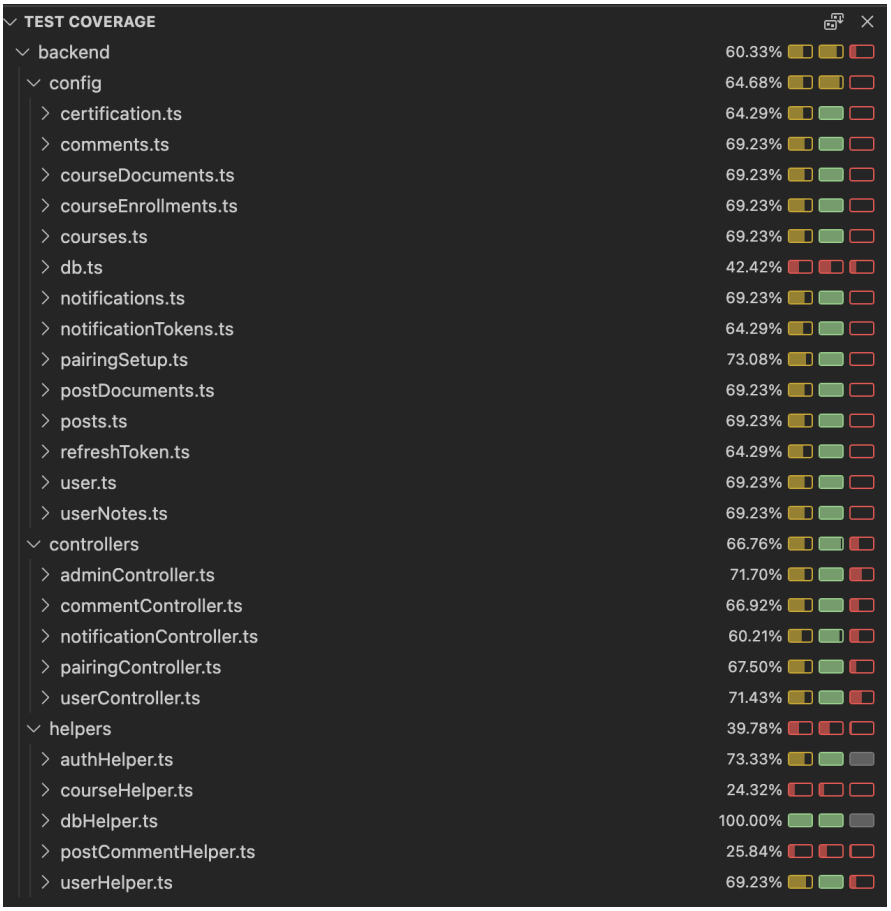

Figure 1: Test Coverage of Our current tests

By using a separate database for testing, the suite effectively verifies database interactions while maintaining the speed and simplicity required for automated testing. This setup ensures that the application functions as intended, providing confidence in the system's overall reliability.

4 Docker

Deployment of the backend is automated using a GitHub Actions pipeline, which builds and pushes a Docker image to Docker Hub. The deployment process ensures consistency and reliability across different environments.

4.1 Docker and Docker Compose

The application is containerized using Docker, with services orchestrated via Docker Compose. This setup simplifies the deployment and scaling process by managing dependencies and configurations for multiple services.

- **Docker Image Creation**: Each push to the main branch triggers a GitHub Actions workflow that builds the backend application into a Docker image. The image is tagged with the latest commit hash and pushed to a dedicated Docker Hub repository.
- **Docker Compose Configuration**: A `docker-compose.yml` file is used to define and manage services required by the application. This can be found in the `examples` folder.
- **Environment Variables**: Critical settings like database credentials, storage configurations, and API keys are managed via a `.env` file, ensuring secure and consistent deployment across environments.

4.2 Deployment Process

To deploy the backend:

1. Ensure the latest changes are merged into the main branch and pushed to the GitHub repository.
2. GitHub Actions will automatically:
    - Build the Docker image for the backend service.
    - Push the image to Docker Hub.
3. On the Automatically redeploys when there's a new image (using the watchtower container):
    - Pulls the latest Docker image from Docker Hub.
    - Runs `docker-compose up -d --build` to deploy the updated services.

4.3 Monitoring and Maintenance

Logs and container statuses can be monitored using Docker Compose commands such as:

- `docker-compose logs` - View logs for all services.
- `docker-compose ps` - Check the status of running containers.

5 Deployment and Automation

5.1 Automated Testing

To ensure the reliability of the codebase, all tests are automatically executed whenever a branch is merged into the `main` branch. This process is facilitated by GitHub Actions, which runs the test suite and reports any failures. The workflow file that defines this process can be found at `.github/workflows/test.yml`. By automating the testing process, we ensure that only tested and validated code is integrated into the production environment, reducing the likelihood of introducing bugs or regressions.

5.2 Automated Deployment

Deployment is fully automated using Docker builds and GitHub Actions. After merging changes into the `main` branch, a GitHub Actions workflow triggers the creation of a new Docker image for the backend. This image is packaged and pushed to the container registry. The deployment workflow, defined in `.github/workflows/cloud-deploy.yml`, subsequently redeploys the updated Docker image to an AWS-like cloud service. This automated pipeline ensures that the latest changes are consistently and reliably deployed to the production environment without manual intervention.

5.3 Local Testing

For developers who wish to test the backend locally, the process is straightforward:

1. Clone the repository to your local machine.
2. Open the repository in the terminal of your choice.
3. Navigate to the backend folder: `cd backend`.
4. Install the required dependencies: `npm install`.
5. Run the development server: `npm run dev`.

This setup provides a local instance of the backend, allowing developers to test features and debug any issues without deploying to the cloud.

6 Further Implementations

6.1 Khan Academy Resources

To enhance the learning experience, we propose integrating Khan Academy resources into each course. By utilizing the Khan Academy API, we can dynamically link relevant videos, exercises, and materials to individual courses. This integration will provide students with access to supplementary content that aligns with the curriculum, ensuring a richer and more comprehensive learning experience. The API documentation is available at Khan Academy API Documentation, which outlines the endpoints and data structure required for implementation. Each course in our platform can query the API to retrieve curated resources based on topics or keywords, seamlessly embedding them within the course interface.

6.2 Chatrooms

To facilitate better communication and collaboration among students and instructors, we propose implementing real-time chatrooms using `socket.io`. This technology enables low-latency, bi-directional communication, making it ideal for real-time messaging. Additionally, we recommend adding a new database schema to store chat history, ensuring that messages can be retrieved and reviewed later. This schema would include fields for sender ID, receiver ID, message content, timestamps, and any relevant metadata. By combining
textttsocket.io for real-time interactions with persistent chat storage, we can create a robust messaging system that supports group discussions, private chats, and instructor-led Q&A sessions, enhancing the overall learning experience.