Project Documentation: Historical Changes System

Introduction

This document serves as a comprehensive guide to the implementation and functionality of the Historical Changes System. The system is designed to track changes made to fields in a collection, storing historical records efficiently and securely in a database. The system utilizes NestJS for the backend development and connects to a database of choice MongoDB.

System Overview

The Historical Changes System is built as a NestJS project, utilizing TypeScript for enhanced type safety and code clarity. The system includes the following key components:

1. **NestJS Backend**: The backend is developed using NestJS, a progressive Node.js framework. NestJS provides a robust foundation for building scalable and maintainable server-side applications.
2. **Database Integration**: The system connects to a chosen database MongoDB to store historical change records efficiently and securely.
3. **Interceptors**: NestJS interceptors are implemented to capture changes made to fields in the collection. These interceptors intercept incoming requests and process them accordingly to generate historical change records.
4. **Controller**: The main controller of the system utilizes the implemented interceptors to create historical change records for each modified field. It ensures that the previous and new values along with the user who made the change are stored accurately.
5. **Design Patterns**: The system incorporates various design patterns to ensure code maintainability, scalability, and flexibility. Examples of design patterns used include Dependency Injection, Module Pattern, Facade Pattern, Decorator Pattern, and Middleware Pattern.
6. **SOLID Principles**: The system adheres to the SOLID principles, ensuring that the codebase is structured in a way that promotes maintainability, extensibility, and reusability.
7. **Postman Collection**: A Postman collection is provided as a JSON file containing a set of requests to test the functionality of the application endpoints. The collection covers both positive and negative test cases to validate the robustness of the application.

Implementation Details

Backend Development

The backend is developed using NestJS, leveraging its powerful features such as decorators, modules, providers, middleware, and interceptors. TypeScript is extensively used to enhance code readability and maintainability.

Database Integration

The system connects to the chosen database using appropriate NestJS modules Mongoose for MongoDB. The database is utilized to store historical change records efficiently and securely.

Interceptors and Controllers

Interceptors are implemented to capture changes made to fields in the collection. These interceptors are utilized in the Update Logs Controller to create historical change records for each modified field.

Design Patterns and SOLID Principles

Various design patterns such as Dependency Injection, Module Pattern, Facade Pattern, Decorator Pattern, and Middleware Pattern are incorporated into the codebase to ensure code maintainability and extensibility. The SOLID principles are followed to ensure that the codebase is structured in a way that promotes scalability and flexibility.

Project Structure

The project follows a structured layout to maintain clarity and organization of code components. Here's an overview of the directory structure:

Directories:

- **constants**: Contains authentication-related constants, enums, and interfaces for enhanced code readability and maintainability.
- **controllers**: Houses controllers responsible for handling incoming HTTP requests, including authentication, product management, and update logs.
- **guards**: Defines authentication guards to control access to specific routes based on user authentication status.
- **interceptors**: Contains the logging interceptor responsible for capturing changes made to fields in the collection.
- **modules**: Houses modules defining the controllers and providers responsible for the particular task.
- **schemas**: Defines schema for the documents and collections
- **services**: Provides services for business logic implementation, including authentication, product management, and update logs.
- **dto**: Holds data transfer objects (DTOs) used for data validation and manipulation in controllers and services.

Authentication Flow

The authentication flow is managed by the **AuthController** and associated services. Key components include:

- **AuthController**: Handles user authentication requests, including sign-in and retrieving user details.
- **AuthService**: Implements authentication logic, including user authentication and token generation.

- **AuthenticationGuard**: A guard to protect routes that require authentication, ensuring only authenticated users can access them.

Product Management

Product-related functionality is managed by the **ProductController** and associated services. Key components include:

- **ProductController**: Handles product creation and retrieval requests, ensuring proper validation and error handling.
- **ProductService**: Implements business logic for product management, including creating and fetching products from the database.

Update Logs

The **UpdateLogsController** facilitates centralized logging of field updates. Key components include:

- **UpdateLogsController**: Provides endpoints for updating products and retrieving update logs.
- **LoggingInterceptor**: Intercepts requests to capture changes made to fields in the collection, logging historical records in the database.
- **UpdateLogsService**: Implements logic for updating products and retrieving update logs.

Data Transfer Objects (DTOs)

Authentication DTO

The AuthControllerDto and UserSignUp DTOs are used for handling authentication-related data. These DTOs ensure proper validation of user input and help maintain data integrity. Key features include:

- **AuthControllerDto**: Handles email input for user authentication, ensuring it is a non-empty string.
- **UserSignUp**: Manages user sign-up data, including email and password, with appropriate validation rules.

Product DTO

The ProductDto DTO facilitates data transfer for product-related operations. It ensures that product information is properly validated before being processed. Key features include:

- **ProductDto**: Contains fields for product name, price, description, and optional offer, with validation rules to ensure data integrity.

Guards

The AuthenticationGuard is responsible for protecting routes that require user authentication. It ensures that only authenticated users can access protected endpoints. Key features include:

- **AuthenticationGuard**: Verifies the presence of a valid authentication token in the request headers and throws an error if authentication fails.

Interceptor

The LoggingInterceptor intercepts incoming requests to capture changes made to fields in the collection. It facilitates centralized logging of historical change records in the database. Key features include:

- **LoggingInterceptor**: Captures changes made to fields in the collection, logging historical records with details such as old and new values and user ID.

Modules

The project utilizes NestJS modules to organize and manage dependencies. Here's an overview of the modules used in the project:

AppModule

The AppModule serves as the root module of the application. It imports and configures other modules required for the application to function properly. Key features include:

- **ConfigModule**: Handles application configuration, including environment variables and configuration files.
- **DatabaseModule**: Configures database connections for MongoDB.
- **AuthModule**: Manages authentication-related functionality, including user authentication and token generation.
- **ProductModule**: Handles product management functionality, including product creation and retrieval.
- **UpdateLogsModule**: Facilitates centralized logging of field updates, including updating products and retrieving update logs.

AuthModule

The AuthModule is responsible for managing authentication-related functionality, including user authentication and token generation. Key features include:

- **AuthService**: Implements authentication logic, including user authentication and token generation.
- **AuthController**: Handles authentication requests, including user sign-in and retrieving user details.
- **AuthenticationGuard**: Protects routes that require user authentication, ensuring only authenticated users can access them.

ProductModule

The ProductModule handles product management functionality, including product creation and retrieval. Key features include:

- **ProductService**: Implements business logic for product management, including creating and fetching products from the database.
- **ProductController**: Handles product creation and retrieval requests, ensuring proper validation and error handling.

UpdateLogsModule

The UpdateLogsModule facilitates centralized logging of field updates. Key features include:

- **UpdateLogsService**: Implements logic for updating products and retrieving update logs.
- **UpdateLogsController**: Provides endpoints for updating products and retrieving update logs.

Services and Schemas Documentation Services

1. **AppService**
    - Description: Provides health check functionality to ensure the server is up and running.
    - Methods:
        - getHealth(): Returns an object containing status, health, and message indicating the server's health status.
1. **AuthService**
    - Description: Manages user authentication and authorization.
    - Dependencies:
        - AuthModel: Model for interacting with the Auth collection.
        - LoggerService: Service for logging errors and information.
        - JwtService: Service for handling JSON Web Tokens (JWT).
    - Methods:
        - SignIn(user: AuthControllerDto): Authenticates a user and generates an access token upon successful authentication.
        - SignUp(user: any): Registers a new user with hashed password and handles duplicate user registration.
        - isAdmin(user: string): Checks if the user is an admin.
        - isAnUser(user: any): Checks if the user is registered.
        - getUser(id: string): Retrieves user information by ID.
1. **Logger**
    - Description: Manages logging functionality for error and information messages.
    - Dependencies: None
    - Methods:
        - error(message: string, stack?: string): Logs an error message with an optional stack trace.
        - info(message: string): Logs an informational message.
1. **MongoDBLoggerService**
    - Description: Handles logging of historical changes to the MongoDB database.
    - Dependencies: MongoDB client
    - Methods:

- logChange(oldValue: any, newValue: any, userId: string): Logs a change made to a document.
- getLogs(): Retrieves all logs from the MongoDB database.

5. **ProductService**

- Description: Manages product-related operations such as creation and retrieval.
- Dependencies:
  - ProductModel: Model for interacting with the Product collection.
  - LoggerService: Service for logging errors and information.
- Methods:
  - createProduct(product: ProductDto): Creates a new product and logs the operation.
  - getProducts(): Retrieves all products from the database.

1. **UpdateLogsService**
   - Description: Handles update operations and retrieval of change logs.
   - Dependencies:
     - MongoDBLoggerService: Service for logging historical changes.
     - ProductModel: Model for interacting with the Product collection.
   - Methods:
     - updateProduct(product: any): Updates a product and logs the change.
     - getAllLogs(): Retrieves all change logs from the MongoDB database.

Schemas

1. **Auth**
   - Description: Defines the schema for user authentication data.
   - Fields:
     - _id: Unique identifier (automatically generated).
     - email: User email address (required, unique).
     - password: User password (required).
     - token: Authentication token (optional).

1. **Product**
   - Description: Defines the schema for product data.
   - Fields:
     - _id: Unique identifier (automatically generated).
     - productName: Name of the product (required, unique).
     - price: Price of the product (required).
     - description: Description of the product (required).
     - offer: Special offer associated with the product (optional).

These services and schemas form the backbone of the application, providing essential functionality for user authentication, product management, logging, and historical change tracking.

Environment Variables PORT

- Description: Specifies the port number on which the server will listen for incoming connections.
- Value: 3999

NODE

- Description: Specifies the environment in which the application is running.
- Value: Development

DATABASE_HOST

- Description: Specifies the connection URI for the MongoDB database where application data is stored.
- Value:

```
DATABASE_HOST=mongodb+srv://unnit:test1234@cluster0.xliv9yt.mongodb.net/hcsm
```

JWT_SECRET

- Description: Secret key used for signing JSON Web Tokens (JWT) for user authentication.
- Value: QZx611X5C_qrcGlOH5PxhQDn1zlbBvbK

These environment variables are crucial for configuring the application's behavior and connecting it to external services such as databases. Ensure these values are correctly set according to your deployment environment.

Testing

A Postman collection is provided to test the functionality of the application endpoints. The collection includes requests covering both positive and negative test cases to validate the robustness of the application. Detailed descriptions and expected outcomes are provided for each request.

Setup Instructions

To set up and run the Historical Changes System, follow the instructions provided in the project README file. Ensure that all necessary dependencies are installed, environment variables are configured, and database connections are established before running the application.

Note: In the GitHub repository, there are two files: 'client' and 'server'. To run both, please follow the instructions provided in the README file.

Conclusion

The Historical Changes System is designed and implemented to effectively track changes made to fields in a collection, storing historical records efficiently and securely in a database. By adhering to design patterns, SOLID principles, and leveraging TypeScript features, the system ensures code maintainability, scalability, and flexibility. The provided Postman collection facilitates thorough testing of the application endpoints, ensuring its robustness and reliability.

This documentation covers the implementation details, functionality, and testing procedures of the Historical Changes System. For any further inquiries or assistance, please refer to the project README file or contact the developer.