scriptorium backend

Author: Kevin Le 1007952805

To first introduce the context of the app, it is a blog post platform that allows users to share code and run code all in one app. This is the documentation for the backend server of the app.

Getting Started

IMPORTANT NOTE: - please read the directories of each endpoint section as it will help explain some prerequisites, order of execution for requests etc.

Pre-reqs:

Please ensure you have ran the `./startup.sh` to set up a fresh new environment with fresh data and an admin account. After so, run `./run.sh` to start up the backend server to start calling the endpoints.

If you run into permission issues running the scripts please run the following command: `chmod +x ./startup.sh ./run.sh`

Admin Account:

Below is the admin account that should have been generated / created upon running the `./startup.sh` script. Just for reference and whenever needed - but there is already a `login admin` postman request under the `auth directory` that has prefilled the admin data for you to just login as an admin.

- email: admin@gmail.com
- password: 12345

First Steps:

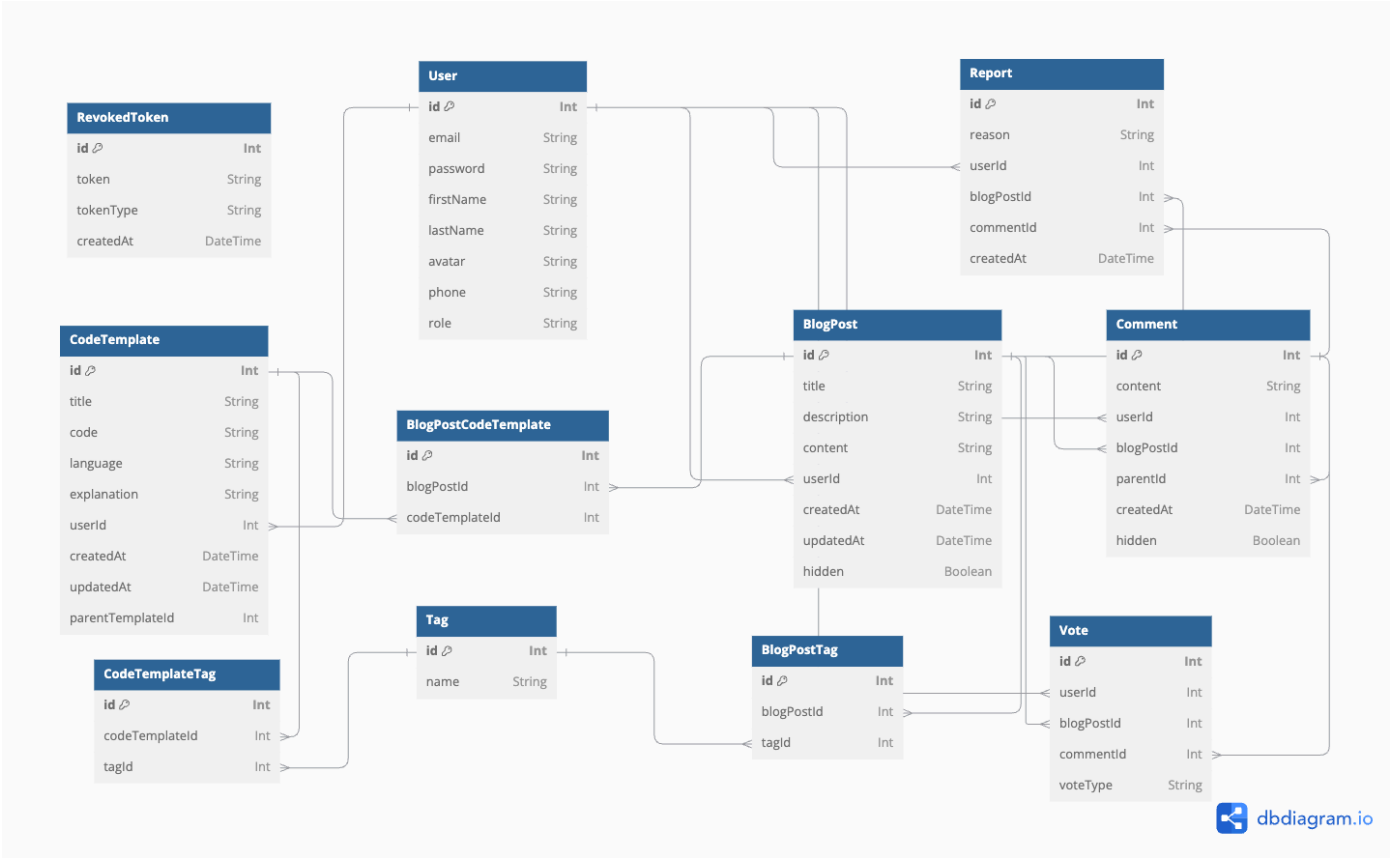
- First, the postman collection assumes that the baseUrl is `http://localhost:3000` . If it is different, feel free to change the `baseUrl` environment variable of the postman collection to the your baseUrl. Make sure it is the `environment` variables.
- Next, please head to the live folder to ensure that the app is running. This is just a health check endpoint to ensure that everything is up and running.
- Next, head over to the auth folder where you can start logging in/ registering new users to initialize postman environment variables that will be propped to the other endpoint folders.
    - Register the user register user , there is already data there for you to register immediately.
    - The creds of the user email and passwordwill automatically be passed into the login user postman request
    - There is also a login admin request that prefills the admin creds to log in with the user.
    - Important Note: the tokens are set to expire in a 24h window for both access and login
- This is a good chance to try out the logout request and refresh request features

With these, now you can navigate to specific directories with more information of the different endpoints. Again, please refer to the respective directory documentation for more information about order of execution and such.

However, the order of execution should be top to bottom respective in context of the requests with in the level of the directory.

Data Models

Here is a database diagram showcasing the database tables and its relation, below will be a brief outline and explanation of each one. Here is the link: https://dbdiagram.io/d/scriptorium-backend-db-diagram-6727b01fb1b39dd858520245



Here is a brief skim of the database models that helps orchestrate all the endpoints together.

1. User

This is the user table that stores all information relating to the user.

- Fields:
  - id (Int): Primary key, automatically increments.
  - email (String): Unique email identifier for each user. (this is encrpyed in db)
  - password (String): Hashed password for authentication.
  - firstName (String): User's first name.
  - lastName (String): User's last name.
  - avatar (String?): Optional URL to the user's profile image.
  - phone (String): Contact phone number.
  - role (String): Role of the user, default is "user".
- Relationships:

    - codeTemplates (CodeTemplate[]): Code templates created by the user.
    - blogs (BlogPost[]): Blog posts created by the user.
    - comments (Comment[]): Comments created by the user.
    - reports (Report[]): Reports created by the user.
    - votes (Vote[]): Votes cast by the user.

2. CodeTemplate

This is a saved code template that a user would have saved or forked.

- Fields:
    - id (Int): Primary key, automatically increments.
    - title (String): Title of the code template.
    - code (String): The code content.
    - language (String): Programming language of the code.
    - explanation (String?): Optional explanation or documentation for the code.
    - userId (Int): Foreign key linking to the user who created the template.
    - createdAt (DateTime): Timestamp when the template was created.
    - updatedAt (DateTime): Timestamp when the template was last updated.
    - parentTemplateId (Int?): Optional ID of the parent template if the code is a fork.
- Relationships:
    - user (User): The user who created the template.
    - tags (CodeTemplateTag[]): Tags associated with the template.
    - blogPosts (BlogPostCodeTemplate[]): Blog posts associated with the template.
    - forkedTemplates (CodeTemplate[]): Templates forked from this template.

3. BlogPost

This is a blog post by a certain user that can be hidden.

- Fields:
    - id (Int): Primary key, automatically increments.
    - title (String): Title of the blog post.
    - description (String): Short description or excerpt.
    - content (String): Full content of the blog post.
    - userId (Int): Foreign key linking to the author.
    - createdAt (DateTime): Timestamp when the post was created.
    - updatedAt (DateTime): Timestamp when the post was last updated.
    - hidden (Boolean): Flag indicating if the post is hidden due to reports or moderation.
- Relationships:
    - user (User): The author of the blog post.
    - tags (BlogPostTag[]): Tags associated with the blog post.
    - codeTemplates (BlogPostCodeTemplate[]): Code templates linked to this post.
    - comments (Comment[]): Comments on this post.

    - votes (Vote[]): Votes on this post.
    - report (Report[]): Reports filed against the post.

4. BlogPostCodeTemplate (Relationship Table)

To associate a relationship between blog post and code template, this table is created for blogposts to be able to easily fetch the code templates relating to the blog post (attached within the post)

- Fields:
    - id (Int): Primary key, automatically increments.
    - blogPostId (Int): Foreign key linking to the blog post.
    - codeTemplateId (Int): Foreign key linking to the code template.
- Relationships:
    - blogPost (BlogPost): The blog post associated with the template.
    - codeTemplate (CodeTemplate): The code template associated with the blog post.

5. Comment

This table is for commenting. It can either be a comment to a blog post or a comment to another comment (i.e a reply). This can also be hidden as well.

- Fields:
    - id (Int): Primary key, automatically increments.
    - content (String): Text content of the comment.
    - userId (Int): Foreign key linking to the author of the comment.
    - blogPostId (Int?): Foreign key linking to the blog post, if the comment is on a post.
    - parentId (Int?): Foreign key linking to the parent comment, if it is a reply.
    - createdAt (DateTime): Timestamp when the comment was created.
    - hidden (Boolean): Flag indicating if the comment is hidden due to reports or moderation.
- Relationships:
    - user (User): The author of the comment.
    - blogPost (BlogPost?): The blog post associated with the comment, if applicable.
    - parent (Comment?): The parent comment if it is a reply.
    - replies (Comment[]): Replies to the comment.
    - votes (Vote[]): Votes on this comment.
    - reports (Report[]): Reports filed against the comment.

6. Report

This is a report table that stores all reports for blog posts or comments.

- Fields:
    - id (Int): Primary key, automatically increments.
    - reason (String): Reason for reporting the content.
    - userId (Int): Foreign key linking to the user who reported the content.
    - blogPostId (Int?): Foreign key linking to the blog post if it was reported.

    - commentId (Int?): Foreign key linking to the comment if it was reported.
    - createdAt (DateTime): Timestamp when the report was created.
- Relationships:
    - user (User): The user who reported the content.
    - blogPost (BlogPost?): The blog post being reported, if applicable.
    - comment (Comment?): The comment being reported, if applicable.

7. Tag

This is a generic tag table that stores a unique tag by its name.

- Fields:
    - id (Int): Primary key, automatically increments.
    - name (String): Unique name of the tag.
- Relationships:
    - codeTags (CodeTemplateTag[]): Tags associated with code templates.
    - blogTags (BlogPostTag[]): Tags associated with blog posts.

8. CodeTemplateTag (Relationship Table)

Using the Tag table, this table helps declare a relationship / tag with a code template.

- Fields:
    - id (Int): Primary key, automatically increments.
    - codeTemplateId (Int): Foreign key linking to the code template.
    - tagId (Int): Foreign key linking to the tag.
- Relationships:
    - codeTemplate (CodeTemplate): The code template associated with the tag.
    - tag (Tag): The tag associated with the code template.

9. BlogPostTag (Relationship Table)

Using the Tag table, this table helps declare a relationship / tag with a blog post.

- Fields:
    - id (Int): Primary key, automatically increments.
    - blogPostId (Int): Foreign key linking to the blog post.
    - tagId (Int): Foreign key linking to the tag.
- Relationships:
    - blogPost (BlogPost): The blog post associated with the tag.
    - tag (Tag): The tag associated with the blog post.

10. Vote

This table stores a certain vote that a user has invoked on a blog post or comment. UP or DOWN.

- Fields:
  - id (Int): Primary key, automatically increments.
  - userId (Int): Foreign key linking to the user who cast the vote.
  - blogPostId (Int?): Foreign key linking to the blog post, if applicable.
  - commentId (Int?): Foreign key linking to the comment, if applicable.
  - voteType (String): Type of vote (UP or DOWN).
- Relationships:
  - user (User): The user who cast the vote.
  - blogPost (BlogPost?): The blog post associated with the vote.
  - comment (Comment?): The comment associated with the vote.

11. RevokedToken

This table is used to store all revoked tokens when the user logs out so that they can't use the same token to log back in again or else the backend server will know and decline access the respective endpoint.

- Fields:
  - id (Int): Primary key, automatically increments.
  - token (String): Unique identifier for the revoked token.
  - tokenType (String): Type of token (e.g., "access" or "refresh").

createdAt (DateTime): Timestamp when the token was revoked.

---

live

First let's make sure that the server is running properly. This "live" endpoint does a basic ping to ensure that the server is running.

GET health check

http://localhost:3000/api/live

This endpoint is a health check to indicate if the server is running and up. It makes an HTTP GET request to http://localhost:3000/api/live and returns a status code of 200 with a JSON response containing a message field.

---

auth

This directory holds all endpoints relating to authentication and authorization. The `login user` and `login admin` is recommended to be ran before going to other directories to automatically associate environment `accessToken` and `refreshToken` variables that will use in other requests in other directories

POST register user

http://localhost:3000/api/users/register

Registers a new user, supporting both user and admin roles

{
"email": "user@example.com",
"phone": "+123456789",
"firstName": "John",
"lastName": "Doe",
"password": "your_password",
"role": "user",
"avatar": "https://example.com/avatar.png"
}

Required Fields: `email`, `phone`, `firstName`, `lastName`, `password`, `role`

Optional Field: `avatar`

{
"message": "User registered successfully",
"user": {
"id": 1,
"email": "user@example.com",
"phone": "+123456789",
"firstName": "John",
"lastName": "Doe",
"avatar": "https://example.com/avatar.png"
}
}

Body raw (json)

json

```
{
    "email": "user@gmail.com",
    "phone": "+16479903830",
    "firstName": "John",
    "lastName": "Doe",
    "password": "12345",
    "role": "user",
    "avatar": "https://example.com/avatar.png"
}
```

POST login user

http://localhost:3000/api/users/login

Authenticates a user and returns access and refresh tokens if credentials are valid.

Required Fields: `email`, `password`

returns:

{
"accessToken": "your_access_token",
"refreshToken": "your_refresh_token",
"user": {
"id": 1,
"email": "user@example.com",
"phone": "+123456789",
"firstName": "John",
"lastName": "Doe",
"avatar": "https://example.com/avatar.png",
"role": "user"
}
}

Body raw (json)

json

```
{
    "email": "",
    "password": ""
}
```

POST login admin

http://localhost:3000/api/users/login

same as `login user` but here to login to admin user to test admin operations later on (down below in `admin` directory)

Body raw (json)

json

```
{
    "email": "admin@gmail.com",
    "password": "12345"
}
```

POST refresh

http://localhost:3000/api/users/refresh

Refreshes the access token using a valid, non-revoked refresh token. If the refresh token is not revoked and valid, new tokens are issued

Required Field: `refreshToken`

returns

{
"accessToken": "new_access_token",
"refreshToken": "new_refresh_token"
}

AUTHORIZATION Bearer Token

Token	{{accessToken}}

Body raw (json)

json

```
{
    "refreshToken": "{{refreshToken}}"
}
```

POST logout

http://localhost:3000/api/users/logout

Logs out the user by revoking both the access token provided in the request headers and the refresh token in the request body.

Headers

- `Authorization` : `Bearer`

Body:

Required Field: `refreshToken`

returns

{
"message": "Logged out successfully"
}

AUTHORIZATION Bearer Token

Token	{{accessToken}}

HEADERS

Authorization	Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VySWQiOjEsImVtYWlsIjoia2V2aW5AZ21haWwuY29tIiwicm9sZSI6InVzZXIiLCJpYXQiOjE3MzAzNDIwNDQsImV4cCI6MTczMDM0NTY0NH0.WsI8Zvgioc4-2vOyjhaAXnXoslVQyRVEY5oaE0AWeAA

Body raw (json)

```json
{
    "refreshToken": "{{refreshToken}}"
}
```

users

Before trying requests in this user folder, make sure you have ran any of the login requests in the auth folder of this postman request to make sure postman environment variables as respective `userId` , `tokens` etc.

GET get user profile

http://localhost:3000/api/users/{{userId}}

Fetches the user profile by ID. This endpoint performs an identity check by matching the `idToken` in the request headers with the user ID in the path. If the IDs do not match, access is denied.

Request

- **Headers**
  - `Authorization` : `Bearer`
- **Path Parameters**
  - `id` : The ID of the user to fetch.

returns

{
"message": "User fetched successfully",
"user": {
"id": 1,
"email": "user@example.com",
"phone": "+123456789",
"firstName": "John",
"lastName": "Doe",
"avatar": "https://example.com/avatar.png"
}
}

AUTHORIZATION Bearer Token

Token	{{accessToken}}

HEADERS

Authorization	Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VySWQiOjEsImVtYWlsIjoia2V2aW5AZ21haWwuY29tIiwicm9sZSI6InVzZXIiLCJpYXQiOjE3MzAzNDI2ODYsImV4cCI6MTczMDM0NjI4Nn0.HDcN1URgh3LI7swbCe3VP4etfgQAWLyw8d436dfopK0

PUT update user

http://localhost:3000/api/users/{{userId}}

Updates the user's profile details based on provided fields. This endpoint also performs an identity check by matching the `idToken` in the request headers with the user ID in the path, ensuring only the respective user can modify their data.

Request

- **Headers**
  - `Authorization` : `Bearer`
- **Path Parameters**
  - `id` : The ID of the user to update.
- **Body Parameters** (at least one field is required)
  - `email` *(optional)*: Updated email address.
  - `phone` *(optional)*: Updated phone number.
  - `firstName` *(optional)*: Updated first name.
  - `lastName` *(optional)*: Updated last name.
  - `avatar` *(optional)*: Updated avatar URL.

returns

{
"message": "User updated successfully",
"user": {
"id": 1,
"email": "updated@example.com",
"phone": "+123456789",
"firstName": "John",
"lastName": "Doe",
"avatar": "https://example.com/new-avatar.png"
}
}

AUTHORIZATION Bearer Token

Token	{{accessToken}}

HEADERS

Body raw (json)

```json
{
    "firstName": "Newname2"
}
```

code templates

Before trying requests in this folder, make sure you have ran any of the login requests in the auth folder of this postman request to ensure that the environment variables for `userId` , `accessToken` etc are. set.

execute

no auth required to run these.

POST python

http://localhost:3000/api/code/execute

This endpoint allows users to execute a code snippet in a specified programming language with optional input. If the execution is successful, it returns the output of the code; otherwise, it returns an error message.

Body Parameters (all required):

- `language` : The programming language to execute the code in. Supported values might include `python` , `javascript` , `c` , `cpp` , or `java` .
- `code` : The code snippet to be executed.
- `stdin` *(optional)*: Standard input to provide to the code during execution (useful for interactive programs).

returns {
"message": "Code executed successfully",
"result": "Hello, World!"
}

returns a 400 if there is a code execution error and the logs will be sent in the response body

Body raw (json)

json

```
{
    "language": "python",
    "code": "print(input().strip())",
    "stdin": "Hello, World!"
}
```

POST c++

http://localhost:3000/api/code/execute

This endpoint allows users to execute a code snippet in a specified programming language with optional input. If the execution is successful, it returns the output of the code; otherwise, it returns an error message.

Body Parameters (all required):

- `language`: The programming language to execute the code in. Supported values might include `python`, `javascript`, `c`, `cpp`, or `java`.
- `code`: The code snippet to be executed.
- `stdin` *(optional)*: Standard input to provide to the code during execution (useful for interactive programs).

returns {
"message": "Code executed successfully",
"result": "Hello, World!"
}

returns a 400 if there is a code execution error and the logs will be sent in the response body

Body raw (json)

json

```
{
    "language": "c++",
    "code": "#include <iostream>\n#include <string>\nint main() {\n    std::string input;\n
    "stdin": "Hello, World!"
}
```

POST c

http://localhost:3000/api/code/execute

This endpoint allows users to execute a code snippet in a specified programming language with optional input. If the execution is successful, it returns the output of the code; otherwise, it returns an error message.

Body Parameters (all required):

- `language`: The programming language to execute the code in. Supported values might include `python`, `javascript`, `c`, `cpp`, or `java`.

- `code` : The code snippet to be executed.
- `stdin` *(optional)*: Standard input to provide to the code during execution (useful for interactive programs).

returns {
"message": "Code executed successfully",
"result": "Hello, World!"
}

returns a 400 if there is a code execution error and the logs will be sent in the response body

Body raw (json)

```json
{
    "language": "c",
    "code": "#include <stdio.h>\n#define BUFFER_SIZE 100\nint main() {\n    char input[BUFFER_S
    "stdin": "Hello, World!"
}
```

POST javascript

http://localhost:3000/api/code/execute

This endpoint allows users to execute a code snippet in a specified programming language with optional input. If the execution is successful, it returns the output of the code; otherwise, it returns an error message.

Body Parameters (all required):

- `language` : The programming language to execute the code in. Supported values might include `python` , `javascript` , `c` , `cpp` , or `java` .
- `code` : The code snippet to be executed.
- `stdin` *(optional)*: Standard input to provide to the code during execution (useful for interactive programs).

returns {
"message": "Code executed successfully",
"result": "Hello, World!"
}

returns a 400 if there is a code execution error and the logs will be sent in the response body

Body raw (json)

json

```
{
    "language": "javascript",
    "code": "process.stdin.on('data', data => {\n    console.log(data.toString().trim());\n
    "stdin": "Hello, World!"
}
```

POST java

http://localhost:3000/api/code/execute

This endpoint allows users to execute a code snippet in a specified programming language with optional input. If the execution is successful, it returns the output of the code; otherwise, it returns an error message.

Body Parameters (all required):

- `language`: The programming language to execute the code in. Supported values might include `python`, `javascript`, `c`, `cpp`, or `java`.
- `code`: The code snippet to be executed.
- `stdin` *(optional)*: Standard input to provide to the code during execution (useful for interactive programs).

returns {
"message": "Code executed successfully",
"result": "Hello, World!"
}

returns a 400 if there is a code execution error and the logs will be sent in the response body

Body raw (json)

json

```
{
    "language": "java",
    "code": "import java.util.Scanner;\n\npublic class Main {\n    public static void main(Stri
    "stdin": "Hello, Java!"
}
```

POST save / create code template

http://localhost:3000/api/code/

- **Description**: Creates a new code template for a user, requiring `userId`, `title`, `language`, and `code` fields. The user's authorization is validated based on `userId` from the request.
- **Authorization**: Requires identity verification to ensure `userId` in the request matches the authenticated user.

Request Body:

- `userId` *(required)*: ID of the user creating the template.
- `title` *(required)*: Title of the code template.
- `language` *(required)*: Programming language of the template.
- `code` *(required)*: The code snippet content.
- `tags` *(optional)*: Tags associated with the code template.
- `parentTemplateId` *(optional)*: ID of the parent template if this is a forked template.

returns

{
"message": "Code Template created successfully",
"codeTemplate": {
"id": 2,
"userId": 1,
"title": "New JavaScript Template",
"language": "javascript",
"code": "console.log('Hello World');",
"tags": ["javascript", "console"],
"createdAt": "2023-01-01T00:00:00Z",
"updatedAt": "2023-01-01T00:00:00Z"
}
}

AUTHORIZATION Bearer Token

Token	{{accessToken}}

Body raw (json)

```json
{
    "title": "Hello World 2 Example",
    "userId": {{userId}},
    "code": "print('Hello, World!')",
    "parentTemplateId": null,
    "language": "Python",
    "explanation": "This code prints 'Hello, World!' to the console.",
    "tags": ["basic", "hello world", "python"]
}
```

POST fork code template

http://localhost:3000/api/code/

- **Description**: Creates a new code template for a user, requiring `userId`, `title`, `language`, and `code` fields. The user's authorization is validated based on `userId` from the request.
- **Authorization**: Requires identity verification to ensure `userId` in the request matches the authenticated user.

Request Body:

- `userId` *(required)*: ID of the user creating the template.
- `title` *(required)*: Title of the code template.
- `language` *(required)*: Programming language of the template.
- `code` *(required)*: The code snippet content.
- `tags` *(optional)*: Tags associated with the code template.
- `parentTemplateId` *(optional)*: ID of the parent template if this is a forked template.

returns

{
"message": "Code Template created successfully",
"codeTemplate": {
"id": 2,
"userId": 1,
"title": "New JavaScript Template",
"language": "javascript",
"code": "console.log('Hello World');",
"tags": ["javascript", "console"],
"createdAt": "2023-01-01T00:00:00Z",
"updatedAt": "2023-01-01T00:00:00Z"
}
}

AUTHORIZATION Bearer Token

Token	{{accessToken}}

Body raw (json)

json

```
{
    "title": "Hello World 2 Example FORKED",
    "userId": {{userId}},
    "code": "print('Hello, World!')"
```

```
    "parentTemplateId": {{codeTemplateId}},
    "language": "Python",
    "explanation": "This code prints 'Hello, World!' to the console.",
    "tags": ["basic", "hello world", "python"]
}
```

PUT update code template

http://localhost:3000/api/code/{{codeTemplateId}}

- **Description**: Updates an existing code template by ID, modifying attributes like title, code, language, and tags. Only the original author (identified by `userId`) is permitted to update.
- **Authorization**: Requires user identity verification to ensure the `userId` in the request matches the template's creator.
- Path Parameters:Request Body:
  - `id` *(required)*: The unique identifier for the code template to be updated.
  - `title` *(optional)*: New title for the code template.
  - `code` *(optional)*: Updated code snippet.
  - `language` *(optional)*: Programming language.
  - `explanation` *(optional)*: Explanation for the code template.
  - `tags` *(optional)*: Updated tags for the code template.

returns

{
"message": "Code template updated successfully",
"codeTemplate": {
"id": 1,
"userId": 1,
"title": "Updated Template Title",
"code": "console.log('Updated Code');",
"language": "javascript",
"explanation": "Logs an updated message.",
"tags": ["javascript", "update"]
}
}

AUTHORIZATION Bearer Token

Token	{{accessToken}}

Body raw (json)

```
.
```

json

```
{
    "title": "new title",
    "tags": ["new", "basic", "python"]
}
```

GET get code template

http://localhost:3000/api/code/{{codeTemplateId}}

- **Description**: Fetches a code template by its ID, returning all relevant details including tags, language, and explanation if provided.
- **Authorization**: No authorization check for this endpoint; anyone can access code templates by ID.
- Path Parameters:
  - `id` *(required)*: The unique identifier for the code template.

returns

{
"message": "Code template fetched successfully",
"codeTemplate": {
"id": 1,
"userId": 1,
"title": "Example Template",
"code": "console.log('Hello World');",
"language": "javascript",
"explanation": "Logs a message to the console.",
"tags": ["javascript", "console"]
}
}

AUTHORIZATION Bearer Token

Token	{{accessToken}}

GET get saved code templates

http://localhost:3000/api/code/?userId={{userId}}

- For this example, we filter by userId for when we want to fetch a user's saved code templates
- **Description**: Fetches code templates based on optional filters like `title`, `content`, `tags`, `userId`, `page`, and `limit`.

- **Authorization**: Requires user identity verification if `userId` is provided to ensure only authorized users can access data.

- Query Parameters:
  - `page` *(optional)*: Page number for pagination (e.g., `1`, `2`).
  - `limit` *(optional)*: Number of templates per page.
  - `userId` *(optional)*: User ID to filter code templates by a specific user.
  - `title` *(optional)*: Filters by title.
  - `content` *(optional)*: Searches templates containing specific content.
  - `tags` *(optional)*: Comma-separated list of tags to filter templates (e.g., `javascript,api`).

AUTHORIZATION Bearer Token

Token	{{accessToken}}

PARAMS

userId	{{userId}}

GET get code templates

http://localhost:3000/api/code/?tag=python

- **Description**: Fetches code templates based on optional filters like `title`, `content`, `tags`, `userId`, `page`, and `limit`.
- **Authorization**: Requires user identity verification if `userId` is provided to ensure only authorized users can access data.
- Query Parameters:
  - `page` *(optional)*: Page number for pagination (e.g., `1`, `2`).
  - `limit` *(optional)*: Number of templates per page.
  - `userId` *(optional)*: User ID to filter code templates by a specific user.
  - `title` *(optional)*: Filters by title.
  - `content` *(optional)*: Searches templates containing specific content.
  - `tags` *(optional)*: Comma-separated list of tags to filter templates (e.g., `javascript,api`).

returns {
"message": "Code templates fetched successfully",
"codeTemplates": [
{
"id": 1,
"userId": 1,
"title": "Sample Template",
"language": "javascript"

"language": "javascript",
"code": "console.log('Hello World');",
"tags": ["javascript", "console"],

"createdAt": "2023-01-01T00:00:00Z",
"updatedAt": "2023-01-01T00:00:00Z"
}
],
"totalCount": 50
}

AUTHORIZATION Bearer Token

Token	{{accessToken}}

PARAMS

tag	python

blog posts

Before trying requests in this folder, make sure you have ran any of the login requests in the auth folder of this postman request.

also for the nested folders relating to

- `fetch sorted blog posts`
- `comments`

I suggest ensuring you have created a blog post already with the respective user before going any further into the nested folders. Go to `create blog post` request to do so.

fetch sorted blog posts

before fetching blogposts make sure to have created some blog posts to see results and such

GET search blog posts

http://localhost:3000/api/blogs?title=timeless&tags=Async

- **Description**: Retrieves a list of blog posts, with options to filter by title, content, tags, and associated code templates. Supports pagination and sorting.

- **Authorization**: Checks the user ID from the request header and filters blog posts accordingly.
- Query Parameters:
    - `page` *(optional)*: Page number for pagination.
    - `limit` *(optional)*: Number of blog posts per page.
    - `title` *(optional)*: Filter blog posts by title.
    - `content` *(optional)*: Filter blog posts by content.
    - `codeTemplateIds` *(optional)*: Comma-separated list of code template IDs to filter by.
    - `tags` *(optional)*: Comma-separated list of tags to filter by.
    - `orderBy` *(optional)*: Sort blog posts by `mostReported`, `mostValued`, or `mostControversial`. (NOTE mostValued is only available to be used in admin endpoint, so it will be canceled / nulled at this endpoint)

returns {
"message": "Blog Posts fetched successfully",
"blogPosts": [
{
"id": 1,
"userId": 2,
"title": "Understanding Async in JavaScript",
"description": "An intro to asynchronous programming in JS.",
"content": "Content goes here...",
"hidden": false,
"codeTemplateIds": [1, 3],
"createdAt": "2024-11-01T12:00:00Z",
"updatedAt": "2024-11-01T12:30:00Z",
"tags": ["javascript", "async"],
"commentIds": [10, 11]
}
],
"totalCount": 1
}

AUTHORIZATION Bearer Token

Token	{{accessToken}}

PARAMS

page	1
limit	1
title	timeless
tags	Async

GET fetch most controversial blog posts

GET fetch most controversial blog posts

http://localhost:3000/api/blogs?orderBy=mostControversial

same as `search blog posts` but this time just an example of the orderBy usage for most controversial blog posts which means most impressions (upvotes and downvotes)

AUTHORIZATION Bearer Token

Token	{{accessToken}}

PARAMS

orderBy	mostControversial

GET fetch most valued blog posts

http://localhost:3000/api/blogs?orderBy=mostValued

same as `search blog posts` but this time just an example of the orderBy usage for most valued which is the highest positive difference between upvotes and downvotes.

AUTHORIZATION Bearer Token

Token	{{accessToken}}

PARAMS

orderBy	mostValued

comments

Before trying requests in this folder, make sure a blog post has been created within `create blog post` in the parent directory.

POST comment on blog post

http://localhost:3000/api/blogs/{{blogPostId}}/comment

- **Description**: Allows an authorized user to add a comment to a specific blog post by `id`. This endpoint ensures the user making the comment matches the `userId` specified in the request.
- **Authorization**: Requires an identity check to confirm that the user making the request matches the `userId` associated with the comment.

Path Parameters:

- `id` *(required)*: The ID of the blog post to comment on.

Request Body:

- `userId` *(required)*: The ID of the user adding the comment.
- `content` *(required)*: The content of the comment.

return {
"message": "Blog Post successfully commented on",
"comment": {
"id": 45,
"userId": 3,
"blogPostId": 12,
"content": "Great post!",
"createdAt": "2024-10-01T12:00:00.000Z"
}
}

AUTHORIZATION Bearer Token

Token	{{accessToken}}

Body raw (json)

json

```
{
    "userId": {{userId}},
    "content": "Wow!"
}
```

GET get direct comments from blog post

http://localhost:3000/api/blogs/{{blogPostId}}/comment

- **Description**: Retrieves a paginated list of direct comments for a specific blog post by `id`. Optionally filters comments to only include those visible to the `userId` making the request.

- **Authorization**: Requires an identity check to confirm that the requesting user has permission to view comments on the blog post if any comments are restricted.

Path Parameters:

- `id` *(required)*: The ID of the blog post for which to retrieve comments.

Query Parameters:

- `page` *(optional)*: The page number to fetch (default: `1`).
- `limit` *(optional)*: The number of comments per page (default: `10`).


```
{
"message": "Blog Post direct comments fetched successfully",
"comments": [
{
"id": 45,
"userId": 3,
"blogPostId": 12,
"content": "Great post!",
"createdAt": "2024-10-01T12:00:00.000Z"
}
],
"totalCount": 25
}
```


AUTHORIZATION Bearer Token

Token	{{accessToken}}

POST reply to comment

http://localhost:3000/api/comments/{{commentId}}/comment

- **Description**: Allows an authorized user to add a reply to a specific comment by `id`. The endpoint ensures that the user making the request matches the `userId` specified in the request.
- **Authorization**: Requires an identity check to confirm that the user making the request matches the `userId` associated with the reply; otherwise, the request is rejected.

Path Parameters:

- `id` *(required)*: The ID of the comment to reply to.

Request Body:

- `userId` *(required)*: The ID of the user adding the reply.
- `content` *(required)*: The content of the reply.

returns {
"message": "Comment successfully replied",
"comment": {
"id": 78,
"userId": 3,
"parentId": 45,
"content": "Thanks for the insight!"
}
}

AUTHORIZATION Bearer Token

Token	{{accessToken}}

Body raw (json)

```json
{
    "userId": {{userId}},
    "content": "you said the same thing before!"
}
```

GET get direct replies from comments

http://localhost:3000/api/comments/{{commentId}}/comment

- **Description**: Retrieves a paginated list of direct replies for a specific comment by `id`. Optionally filters replies to only include those visible to the `userId` making the request.
- **Authorization**: Requires an identity check to confirm that the requesting user has permission to view replies on the comment if any replies are restricted.
- Path Parameters:Query Parameters:
  - `id` *(required)*: The ID of the comment for which to retrieve replies.
  - `page` *(optional)*: The page number to fetch (default: `1`).
  - `limit` *(optional)*: The number of replies per page (default: `10`).

returns {
"message": "Comment direct replies fetched successfully",
"comments": [
{
"id": 78,

"userId": 3,
"parentId": 45,
"content": "Thanks for the insight!"
}
],
"totalCount": 5
}

AUTHORIZATION Bearer Token

Token	{{accessToken}}

GET get comment

http://localhost:3000/api/comments/{{commentId}}

- **Description**: Retrieves details of a specific comment by `id`. If the comment is hidden, only the comment owner can access it.
- **Authorization**: If the comment is hidden, the authorization check ensures only the comment owner can view it.

Path Parameters:

- `id` *(required)*: The ID of the comment to retrieve.

return {
"message": "Comment fetched successfully",
"comment": {
"id": 12,
"content": "This is a comment.",
"userId": 3,
"blogPostId": 5,
"parentId": null,
"createdAt": "2023-10-31T10:00:00Z",
"hidden": false,
"replyIds": [13, 14],
"upVotes": 10,
"downVotes": 2,
"reportIds": [21, 22]
}
}

AUTHORIZATION Bearer Token

Token	{{accessToken}}

PUT update comment

PUT update comment

http://localhost:3000/api/comments/{{commentId}}

- **Description**: Updates the content of a specific comment by `id`. Only the comment owner can update it.
- **Authorization**: Requires an identity check to confirm that the user making the request matches the comment's `userId`.

Path Parameters:

- `id` *(required)*: The ID of the comment to update.

Request Body:

- `content` *(required)*: The updated content of the comment.

return {
"message": "Comment updated successfully",
"comment": {
"id": 12,
"userId": 3,
"content": "Updated comment content."
}
}

AUTHORIZATION Bearer Token

Token	{{accessToken}}

Body raw (json)

json

```
{
    "content": "wow edited!"
}
```

POST report comment

http://localhost:3000/api/comments/{{commentId}}/report

- **Description**: Allows a user to report a specific comment by `id`, specifying a reason for the report. Only accessible if the user matches the `userId` provided in the request.

- **Authorization**: Requires an identity check to confirm that the user making the request matches the `userId` associated with the report.

Path Parameters:

- `id` *(required)*: The ID of the comment to be reported.

Request Body:

- `userId` *(required)*: The ID of the user reporting the comment.
- `reason` *(required)*: The reason for reporting the comment (e.g., "Spam," "Inappropriate content").

return {
"message": "Comment successfully reported",
"report": {
"id": 45,
"userId": 3,
"commentId": 12,
"reason": "Inappropriate content"
}
}

AUTHORIZATION Bearer Token

Token	{{accessToken}}

Body raw (json)

```json
{
    "userId": {{userId}},
    "reason": "hurt my feelings"
}
```

POST vote comment

http://localhost:3000/api/comments/{{commentId}}/rate

- **Description**: Allows an authorized user to toggle a vote on a specific comment by `id`. The endpoint verifies that the user making the request is the same as the `userId` specified.
- **Authorization**: Requires an identity check to confirm that the `userId` in the request matches the authenticated user.

Path Parameters:

- `id` *(required)*: The ID of the comment to vote on.

Request Body:

- `userId` *(required)*: The ID of the user toggling the vote.
- `voteType` *(required)*: The type of vote to toggle, such as `upvote` or `downvote`.

return {
"message": "Comment vote toggle successfully applied",
"vote": {
"id": 123,
"userId": 1,
"commentId": 456,
"voteType": "upvote"
}
}

AUTHORIZATION Bearer Token

Token	{{accessToken}}

Body raw (json)

```
json

{
    "userId": {{userId}},
    "voteType": "UP"
}
```

GET get vote for comment

http://localhost:3000/api/comments/{{commentId}}/rate

- **Description**: Fetches the current vote of an authenticated user for a specific comment by `id`.
- **Authorization**: Requires that the user be authenticated, and the user ID must match the user associated with the vote.

Path Parameters:

- `id` *(required)*: The ID of the comment for which to fetch the vote.

return {
"message": "Comment vote successfully fetched for user",

"vote": {
"id": 123,
"userId": 1,
"commentId": 456,
"voteType": "upvote"
}
}

AUTHORIZATION Bearer Token

Token	{{accessToken}}

DELETE delete comment

http://localhost:3000/api/comments/{{commentId}}

- **Description**: Deletes a specific comment by `id`.
- **Authorization**: Only accessible if the user has permission to delete the comment.
- Path Parameters:
  - `id` *(required)*: The ID of the comment to delete.

return {
"message": "Comment deleted successfully"
}

AUTHORIZATION Bearer Token

Token	{{accessToken}}

POST create blog post

http://localhost:3000/api/blogs

- **Description**: Creates a new blog post with the provided details and associates it with code templates and tags. The request is authorized by checking the `userId` in the request with the authenticated user ID.
- **Authorization**: Verifies that the `userId` in the request body matches the authorized user.
- Request Body:
  - `title` *(required)*: Title of the blog post.
  - `description` *(required)*: Short description of the blog post.
  - `content` *(required)*: Main content of the blog post.
  - `userId` *(required)*: ID of the user creating the post.
  - `codeTemplateIds` *(optional)*: Array of code template IDs associated with the blog post.

- `tags` *(optional)*: Array of tags associated with the blog post.

return {
"message": "Blog Post created successfully",
"blogPost": {
"id": 1,
"userId": 2,
"title": "Getting Started with REST APIs",
"description": "A beginner's guide to REST APIs.",
"content": "Content of the blog post...",
"hidden": false,
"codeTemplateIds": [1, 2],
"createdAt": "2024-11-01T12:00:00Z",
"updatedAt": "2024-11-01T12:30:00Z",
"tags": ["REST", "API", "backend"],
"commentIds": []
}
}

AUTHORIZATION Bearer Token

Token	{{accessToken}}

Body raw (json)

json

```
{
    "title": "Exploring TIMELESS",
    "description": "A deep dive into async programming in JavaScript, covering callbacks, promi
    "content": "Asynchronous programming is crucial for developing responsive applications. In
    "userId": {{userId}},
    "codeTemplateIds": [],
    "tags": ["JavaScript", "Async", "Programming", "Web Development"]
}
```

GET get blog post

http://localhost:3000/api/blogs/{{blogPostId}}

- **Description**: Retrieves a specific blog post by ID. If the post is hidden, the endpoint verifies the user's authorization to view it.
- **Authorization**: If the blog post is hidden, the `userId` of the request must match the `userId` of the post.

Path Parameters:

- `id` *(required)*: The ID of the blog post to fetch.

returns {
"message": "Blog Post fetched successfully",
"blogPost": {
"id": 1,
"userId": 2,
"title": "Understanding REST APIs",
"description": "Introduction to REST principles",
"content": "Content of the post...",
"hidden": false,
"codeTemplateIds": [1, 2],
"createdAt": "2024-11-01T12:00:00Z",
"updatedAt": "2024-11-01T12:30:00Z",
"upVotes": 10,
"downVotes": 3,
"tags": ["API", "REST"],
"commentIds": [101, 102]
}
}

AUTHORIZATION Bearer Token

Token	{{accessToken}}

POST vote blog post

http://localhost:3000/api/blogs/{{blogPostId}}/rate

- **Description**: Allows a user to toggle their vote on a specific blog post. The `voteType` specifies whether the vote is an upvote, downvote, or a removal of an existing vote.
- **Authorization**: Requires an identity check by validating the `userId` in the request body against the authenticated user's ID in the authorization header.

Path Parameters:

- `id` *(required)*: The ID of the blog post to vote on.

Request Body:

- `userId` *(required)*: The ID of the user casting the vote.
- `voteType` *(optional)*: The type of vote, either `"upvote"`, `"downvote"`, or `null` to remove the vote.

returns {
"message": "Blog Post vote toggle successfully applied",
"vote": {
"id": 45

"id": 45,
"userId": 2,
"blogPostId": 1,

"voteType": "upvote"
}
}

if the user doesn't have a vote or canceled out then the vote would be returned null

AUTHORIZATION Bearer Token

Token	{{accessToken}}

Body raw (json)

```
json

{
    "userId": {{userId}},
    "voteType": "DOWN"
}
```

GET get vote for blog post

http://localhost:3000/api/blogs/{{blogPostId}}/rate

- **Description**: Retrieves the vote of the currently authenticated user on the specified blog post.
- **Authorization**: Requires authorization to fetch the current user's vote on the blog post.

Path Parameters:

- `id` *(required)*: The ID of the blog post to retrieve the vote for.

returns {
"message": "Blog Post vote successfully fetched for user",
"vote": {
"id": 45,
"userId": 2,
"blogPostId": 1,
"voteType": "downvote"
}
}

AUTHORIZATION Bearer Token

Token {{accessToken}}

POST report blog post

http://localhost:3000/api/blogs/{{blogPostId}}/report

- **Description**: Reports a blog post by its ID, providing a reason for the report. The endpoint creates a new report entry for the specified blog post.
- **Authorization**: Requires `userId` from the request body, which should be the ID of the user submitting the report.
- Path Parameters:Request Body:
  - `id` *(required)*: The ID of the blog post to report.
  - `userId` *(required)*: The ID of the user reporting the blog post.
  - `reason` *(required)*: A string describing the reason for reporting the blog post.

return {
"message": "Blog Post successfully reported",
"report": {
"id": 123,
"userId": 2,
"blogPostId": 1,
"reason": "Inappropriate content"
}
}

AUTHORIZATION Bearer Token

Token {{accessToken}}

Body raw (json)

json

```
{
    "userId": {{userId}},
    "reason": "hurt my feelings"
}
```

PUT update blog post

http://localhost:3000/api/blogs/{{blogPostId}}

- **Description**: Updates an existing blog post. Ensures the `userId` from the request header matches the `userId` of the blog post.
- **Authorization**: Only the post author (matching `userId`) can update the blog post.

Path Parameters:

- `id` *(required)*: The ID of the blog post to update.

Request Body:

- `title` *(optional)*: New title for the blog post.
- `description` *(optional)*: New description for the blog post.
- `content` *(optional)*: New content for the blog post.
- `codeTemplateIds` *(optional)*: List of associated code template IDs.
- `tags` *(optional)*: List of associated tags.

returns {
"message": "Blog Post updated successfully",
"blogPost": {
"id": 1,
"userId": 2,
"title": "Updated REST API Guide",
"description": "Updated description for REST guide",
"content": "Updated content...",
"hidden": false,
"codeTemplateIds": [1, 3],
"createdAt": "2024-11-01T12:00:00Z",
"updatedAt": "2024-11-01T13:00:00Z",
"tags": ["API", "REST", "Backend"],
"commentIds": [101, 102]
}
}

AUTHORIZATION Bearer Token

Token	{{accessToken}}

Body raw (json)

```json
{
    "description": "new description"
}
```

DELETE delete blog post

http://localhost:3000/api/blogs/{{blogPostId}}

- **Description**: Deletes a specific blog post by ID after verifying that the `userId` matches the post author's ID.
- **Authorization**: Only the post author (matching `userId`) can delete the blog post.

Path Parameters:

- `id` *(required)*: The ID of the blog post to delete.

return {

"message": "Blog Post deleted successfully"
}

AUTHORIZATION Bearer Token

Token	{{accessToken}}

---

admin

IN ORDER TO RUN THESE ADMIN ENDPOINTS YOU MUST BE LOGGED IN AS ADMIN

Before trying requests in this folder, make sure you have ran the admin login requests in the auth folder of this postman request - `login admin`

Running `login admin` request in the `auth` directory will set idTokens and userIds to an ADMIN role user to run these requests within this directory

---

PUT set hidden status for comment

http://localhost:3000/api/admin/comments/{{commentId}}/hide

- **Description**: This endpoint allows an administrator to toggle the `hidden` status of a comment. Setting a comment to `hidden` will make it invisible to regular users, while administrators will still be able to access it. Only users with admin privileges are authorized to perform this action.**Authorization**: Requires an admin-level authorization.

Path Parameters:

- `id` *(required)*: The ID of the comment whose hidden status is to be toggled.

Request Body:

- `hidden` *(required)*: A boolean value to set the comment as hidden (`true`) or visible (`false`).

return {
"message": "Comment hidden status has been updated",
"commentId": 123,
"hidden": true
}

AUTHORIZATION Bearer Token

Token	{{accessToken}}

Body raw (json)

```
json

{
    "hidden": true
}
```

PUT set hidden status for blog post

http://localhost:3000/api/admin/blogs/{{blogPostId}}/hide

- **Description**: Allows an administrator to update the `hidden` status of a blog post. This is intended to hide or unhide blog posts as needed. Only users with admin privileges are authorized to perform this action.
- **Authorization**: Requires an admin-level authorization.

Path Parameters:

- `id` *(required)*: The ID of the blog post whose hidden status is to be toggled.

Request Body:

- `hidden` *(optional)*: A boolean value to set the blog post as hidden (`true`) or visible (`false`). If not provided, the default is `false` .

return {
"message": "Blog Post hidden status has been updated",
"blogPostId": 123,
"hidden": true
}

AUTHORIZATION Bearer Token

Token {{accessToken}}

Body raw (json)

```
json

{
    "hidden": true
}
```

GET get most reported blog posts

http://localhost:3000/api/admin/blogs

- **Description**: Retrieves blog posts with the highest number of reports, allowing administrators to review potentially inappropriate or flagged content. Only posts that have received user reports are included in the response.
- **Authorization**: Requires an admin-level authorization.
- `page` *(optional)*: Specifies the page number of results to retrieve. Defaults to the first page if not provided.
- `limit` *(optional)*: Specifies the maximum number of blog posts to retrieve per page. Defaults to a preset limit if not provided.

returns {
"message": "Most reported blog posts fetched successfully",
"blogPosts": [
{
"id": 123,
"userId": 1,
"title": "Inappropriate Post",
"content": "Flagged content",
"reportCount": 10,
"createdAt": "2024-01-01T12:00:00Z"
},
{
"id": 456,
"userId": 3,
"title": "Reported Post",
"content": "Content flagged by multiple users",
"reportCount": 7,
"createdAt": "2024-01-03T09:20:00Z"
}
],
"totalCount": 2
}

AUTHORIZATION Bearer Token

Token	{{accessToken}}

GET get most reported comments

http://localhost:3000/api/admin/comments

- **Description**: This endpoint retrieves the most reported comments to assist administrators in reviewing flagged content. Only comments that have one or more reports will be included in the results. Access to this endpoint is restricted to users with admin privileges.
- **Authorization**: Requires an admin-level authorization.

Query Parameters:

- `page` *(optional)*: The page number to fetch. Defaults to the first page if not provided.
- `limit` *(optional)*: The maximum number of comments to return per page. Defaults to a set limit if not provided.

return {
"message": "Most reported comments fetched successfully",
"comments": [
{
"id": 123,
"userId": 1,
"content": "Inappropriate content",
"reportCount": 5,
"createdAt": "2024-01-01T12:00:00Z"
},
{
"id": 124,
"userId": 2,
"content": "Another reported comment",
"reportCount": 3,
"createdAt": "2024-01-02T15:30:00Z"
}
]
}

AUTHORIZATION Bearer Token

Token	{{accessToken}}