Model Descriptions

1. **User**
    - **Purpose**: Stores user information, including personal details, authentication data, and relationships with other models.
    - **Fields**:
        - `id`: Unique identifier for each user.
        - `firstName` / `lastName`: User's name details.
        - `email`: Unique email used for login.
        - `password`: Encrypted password.
        - `avatar`: Optional profile image.
        - `phoneNumber`: Optional contact number.
        - `role`: Role of the user, defaulting to "user".
        - `createdAt`: Timestamp when the user was created.
    - **Relationships**:
        - `templates`: Code templates created by the user.
        - `blogPosts`: Blog posts authored by the user.
        - `comments`: Comments made by the user.
        - `reports`: Reports submitted by the user.
        - `codeExecutions`: Code executions run by the user.
        - `ratings`: Ratings given by the user.
        - `userPreferences`: User preferences (e.g., theme setting).
2. **Template**
    - **Purpose**: Stores code snippets saved by users, which can be reused and modified.
    - **Fields**:
        - `id`: Unique identifier for each template.
        - `title`: Name of the template.
        - `explanation`: Description of the template’s purpose.
        - `code`: Actual code content of the template.
        - `isFork`: Boolean indicating if the template is a forked version.
    - **Relationships**:
        - `forkedFrom`: (Optional) Original template from which this one is forked.
        - `forkedTemplates`: Forked templates derived from this template.
        - `user`: The user who created the template.
        - `tags`: Tags associated with the template for categorization.
        - `blogPosts`: Blog posts that reference this template.
3. **TemplateTag**
    - **Purpose**: Tags to categorize templates, allowing users to find related code snippets easily.
    - **Fields**:
        - `id`: Unique identifier for each tag.

        - `name`: The name or label of the tag.
    - **Relationships**:
        - `template`: The template to which the tag is associated.
4. **BlogPost**
    - **Purpose**: Stores blog posts created by users, with metadata and links to templates.
    - **Fields**:
        - `id`: Unique identifier for each blog post.
        - `title`: Title of the blog post.
        - `description`: Content or summary of the blog post.
        - `upvoteCount`: Tracks the number of upvotes.
        - `downvoteCount`: Tracks the number of downvotes.
        - `isHidden`: Marks if the blog post is hidden due to reporting.
    - **Relationships**:
        - `user`: The author of the blog post.
        - `tags`: Tags assigned to the blog post.
        - `templates`: Templates linked within the blog post.
        - `comments`: Comments on the blog post.
        - `reports`: Reports filed on the blog post.
        - `ratings`: Ratings associated with the blog post (upvotes or downvotes).
        - `BlogSearch`: Used for searching blog post content.
5. **BlogPostTag**
    - **Purpose**: Tags to categorise blog posts, aiding in search and discovery.
    - **Fields**:
        - `id`: Unique identifier for each tag.
        - `name`: Label for the tag.
    - **Relationships**:
        - `blogPost`: The blog post associated with this tag.
6. **Comment**
    - **Purpose**: Stores comments made by users on blog posts or as replies to other comments.
    - **Fields**:
        - `id`: Unique identifier for each comment.
        - `content`: Text content of the comment.
        - `upvoteCount`: Tracks the number of upvotes.
        - `downvoteCount`: Tracks the number of downvotes.
        - `isHidden`: Marks if the comment is hidden due to reporting.
    - **Relationships**:
        - `user`: The user who made the comment.
        - `blogPost`: (Optional) The blog post to which the comment belongs.
        - `parent`: (Optional) The comment that this comment replies to.
        - `replies`: Replies to this comment.
        - `reports`: Reports filed on this comment.

        - `ratings`: Upvote/downvote ratings on the comment.
7. **Report**
    - **Purpose**: Allows users to report inappropriate content in blog posts or comments.
    - **Fields**:
        - `id`: Unique identifier for each report.
        - `reason`: Explanation for the report.
    - **Relationships**:
        - `user`: The user who filed the report.
        - `blogPost`: (Optional) Blog post associated with the report.
        - `comment`: (Optional) Comment associated with the report.
8. **Rating**
    - **Purpose**: Allows users to rate (upvote or downvote) blog posts and comments.
    - **Fields**:
        - `id`: Unique identifier for each rating.
        - `isUpvote`: Boolean indicating if the rating is an upvote or downvote.
    - **Relationships**:
        - `user`: The user who submitted the rating.
        - `blogPost`: (Optional) The blog post associated with this rating.
        - `comment`: (Optional) The comment associated with this rating.
9. **UserPreferences**
    - **Purpose**: Done in PP2. Stores individual preferences for each user, such as theme settings.
    - **Fields**:
        - `id`: Unique identifier for user preferences.
        - `theme`: Stores the user's theme preference, defaulting to "light".
    - **Relationships**:
        - `user`: The user whose preferences are stored.
10. **CodeExecution**
    - **Purpose**: Stores information about code executions initiated by users.
    - **Fields**:
        - `id`: Unique identifier for each code execution.
        - `code`: The code being executed.
        - `language`: The programming language of the code.
        - `memoryLimit`: Memory limit for the execution.
    - **Relationships**:
        - `user`: (Optional) The user who executed the code.
        - `ExecutionEnvironment`: Configurations related to the execution environment.
11. **ExecutionEnvironment**
    - **Purpose**: Done in PP2. Holds configuration for specific code execution environments, such as memory limits.
    - **Fields**:
        - `id`: Unique identifier for the environment configuration.

        - `memoryLimit`: Limit on memory usage for executions.
    - **Relationships**:
        - `codeExecution`: Code execution associated with this environment.
12. **TemplateSearch**
    - **Purpose**: Stores data used to search through code templates based on title, tags, and content.
    - **Fields**:
        - `id`: Unique identifier for each search entry.
        - `title`: Title of the template being searched.
        - `tags`: Tags associated with the template.
        - `content`: Content to be searched.
    - **Relationships**:
        - `template`: Template associated with the search data.
13. **BlogSearch**
    - **Purpose**: Stores data for blog post search based on title, tags, and content.
    - **Fields**:
        - `id`: Unique identifier for each search entry.
        - `title`: Title of the blog post.
        - `tags`: Tags associated with the blog post.
        - `content`: Content of the blog post to be searched.
    - **Relationships**:
        - `blogPost`: Blog post associated with the search data.

User Endpoints

1. **Sign-Up**
    - **URL**: `/api/accounts/sign-up`
    - **Method**: `POST`
    - **Description**: Creates a new user with basic information.

- **Payload**:
  { "firstName": "string",

  "lastName": "string",

  "email": "string",

  "password": "string",

  "avatar": "string (optional)",

  "phoneNumber": "string (optional)"

  "role": "string (optional)"

  }

- **Example request**:

  {

  "firstName": "TA",

  "lastName": "Example",

  "email": "ta@example.com",

  "password": "password123",

  "avatar": "/avatars/avatar1.png",

  "phoneNumber": "123-456-7890",

  "role": "admin"

  }

- **Response:**

  {

  "message": "User created successfully",

  "user": {

  "id": 13,

  "firstName": "TA",

  "lastName": "Example",

  "email": "ta@example.com",

```
    "avatar": "/avatars/avatar1.png",
    "phoneNumber": "123-456-7890",
    "role": "admin"
  }
}
```


- **User story:** As a user, I want to sign up. Profile information should include first and last name, email, avatar, and phone number. Authentication should be handled with a proper JWT setup.

2. **Log-In**
   - **URL**: `/api/accounts/log-in`
   - **Method**: `POST`
   - **Description**: Authenticates a user by validating their email and password. If the credentials match an existing user, it generates an access token and a refresh token to manage session authentication.
   - **Headers**: None
   - **Payload**:


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


   - **Example request**: {


```
  "email": "kt@example.com",
  "password": "kt"
}
```


   - **Response:**


```
{
  "accessToken":
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VySWQiOjEzLCJlbWFpbCI
6ImV4YW1wbGUxQGV4YW1wbGUuY29tIiwiaWF0IjoxNzMwNTY2MzI2LCJle
```

HAiOjE3MzA1Njk5MjZ9.kWsKY8CsSzU4O8QWjKTOoVMmdIdQfady6414S8YjPJ0",

"refreshToken": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VySWQiOjEzLCJlbWFpbCI6ImV4YW1wbGUxQGV4YW1wbGUuY29tIiwiaWF0IjoxNzMwNTY2MzI2LCJleHAiOjE3MzExNzExMjZ9.g2R6asqfIDTl_TxFflMUJRd3yTdP8Cy6I0H2hLaOIhA"

}

- **User story:** As a user, I want to log in. Profile information should include first and last name, email, avatar, and phone number. Authentication should be handled with a proper JWT setup.

3. **Log-out**
   - **URL**:`/api/accounts/log-out`
   - **Method**: `POST`
   - **Headers:** `Authorization: Bearer {accessToken}`
   - **Description:** Verifies if the user is authenticated and logs them out. The endpoint expects an authorization token in the request headers to validate the user's session. A successful logout occurs by ending the user's session on the client-side (e.g., by removing the token from storage).
   - **Payload**: None required
   - **Example request:** blank
   - **Response:**

     {

     "message":" Successfully logged out."

     }

   - **User story:** As a user, I want to log out. Profile information should include first and last name, email, avatar, and phone number. Authentication should be handled with a proper JWT setup.

4. **Edit**
   - **URL**:`/api/accounts/edit?id={your user id}`
   - **Method**: `PUT`
   - **Parameters – Key:** id **Value:** {user id}
   - **Headers:** `Authorization: Bearer {accessToken}`
   - **Description:** Allows a user to edit their profile information
   - **Payload**: Note: at least one of the fields must have a value.

{

“firstName”: “string” (optional)

“lastName”: “string” (optional)

“email”: “string” (optional)

“avatar”: “string” (optional)

“phoneNumber”: “string” (optional)

}

- **Example request:**

{

“lastName”: “Surname”

}

- **Example Parameters – Key:** id **Value:** 3
- **Response:**

{

“message”: “User updated successfully.”,

“user”: {

“firstName”: “TA”,

“lastName”: “Surname” ,

“email”: “ta@example.com”,

“avatar”: “/avatars/avatar1.png”,

“phoneNumber”: "123-456-7890"

}

}

- **User story:** As a user, I want to edit my profile. Profile information should include first and last name, email, avatar, and phone number. Authentication should be handled with a proper JWT setup.

Code Writing and Execution Endpoints

1. **Write-Code**
    - **URL:** `/api/code-writing-and-execution/write-code`
    - **Method:** `POST`
    - **Description:** Compiles and executes code provided by the user in various programming languages (C, C++, Java, Python, JavaScript), returning the output or error messages.
    - **Payload**:
      {
        "code": "string (required)",
        "language": "string (required)"
      }

    - **Example request:**
      {
        "code": "a = 5\nb = 10\nprint(a + b)",
        "language": "python"
      }

    - **Response:**
      {
        "stdout": "15\n",
        "stderr": ""
      }
    - **User story:** As a visitor (unauthenticated user), I want to write code in various programming languages (e.g., C, C++, Java, Python, JavaScript) on Scriptorium. Also be able to execute my code on Scriptorium and see the output in real-time, and see error messages if my code fails to compile or run so that I can debug my code effectively. This includes compile errors, runtime errors, timeouts, or any warnings generated in the meantime.

2. **Highlight**
    - Doing this user story in PP2 as front-end
3. **Input**
    - **URL:** `/api/code-writing-and-execution/input`
    - **Method:** `POST`
    - **Headers:** None
    - **Description:** Allows a user to provide standard input to their code and execute it in various programming languages
    - **Payload**:
      {
      “code”: “string”
      “language”: “string”
      “input”: “string”
      }
    - **Example request**
      {

“code”: “print(input())”
“language”: “python”
“input”: “Hello World!”
}

- **Response:**
  {
  “output”: “Hello World!\n”,
  “error”: “”
  }
- **User story:** As a visitor, I want to provide input to my code via the standard input (stdin) before execution so that I can test programs that require user input.

Code Templates

1. **Save**
   - **URL:** `/api/code-templates/save`
   - **Method:** `POST`
   - **Headers:** `Authorization: Bearer {accessToken}`
   - **Description:** This endpoint allows authenticated users to create and save a code template, along with optional tags and related blog posts. The saved template is linked to the authenticated user who created it.
   - **Payload**:
     {
     "title": "string",
     "explanation": "string",
     "code": "string",
     "tags": ["string"] (optional),
     "blogPostIds": [int] (optional)
     }:
   - **Example request:**
     {
     "title": "C++ Sorting Algorithm",
     "explanation": "Template for implementing sorting algorithms in C++.",
     "code": "#include <iostream> #include <vector> void bubbleSort(std::vector<int>& arr) { for (size_t i = 0; i < arr.size(); i++) { for (size_t j = 0; j < arr.size() - i - 1; j++) { if (arr[j] > arr[j + 1]) std::swap(arr[j], arr[j + 1]); } } } int main() { std::vector<int> arr = {5, 1, 4, 2, 8}; bubbleSort(arr); for (int num : arr) std::cout << num << ' '; return 0; }",
     "tags": ["cpp", "algorithm", "sorting"],
     "blogPostIds": [2]
     }
   - **Response:**
     {
     "message": "Template saved successfully",
     "template": {
     "id": 6,

```
        "title": "C++ Sorting Algorithm",
        "explanation": "Template for implementing sorting algorithms in C++.",
        "code": "#include <iostream> #include <vector> void
    bubbleSort(std::vector<int>& arr) { for (size_t i = 0; i < arr.size(); i++) { for
    (size_t j = 0; j < arr.size() - i - 1; j++) { if (arr[j] > arr[j + 1]) std::swap(arr[j], arr[j
    + 1]); } } } int main() { std::vector<int> arr = {5, 1, 4, 2, 8}; bubbleSort(arr); for
    (int num : arr) std::cout << num << ' '; return 0; }",
        "isFork": false,
        "forkedFromId": null,
        "userId": 3,
        "tags": [
          {
            "id": 7,
            "name": "cpp",
            "templateId": 6
          },
          {
            "id": 8,
            "name": "algorithm",
            "templateId": 6
          },
          {
            "id": 9,
            "name": "sorting",
            "templateId": 6
          }
        ],
        "blogPosts": [
          {
            "id": 2,
            "title": "Basics of C++ Sorting",
            "description": "A blog post discussing various sorting algorithms
    implemented in C++.",
            "userId": 3,
            "upvoteCount": 0,
            "downvoteCount": 0
          }
        ]
      }
    }
```


- **User story:** As a user (authenticated), I want to save my code as a template with a title, explanation, and tags so that I can organize and share my work effectively.

2. **Search Saved**
   - **URL:** `/api/code-templates/search-saved?userId={your user id}`

- **Method:** `GET, POST`
- **Headers:** `Authorization: Bearer {accessToken}`
- **Parameters – Key:** userID **Value:** {user id}
- **Description:** Allows a user to view their saved templates (GET) or search through their saved templates using a keyword (POST)
- **Payload (for POST, payload for GET is empty)**:
  {
  “query”: “string”
  }
- **Example request (GET):** Empty
- **Example Parameters – Key:** userID **Value:** 2
- **Response (GET):**

```
{
   "savedTemplates": [
      {
         "title": "JavaScript Basics",
         "explanation": "A template for basic JavaScript syntax and examples.",
         "tags": [
            {
               "id": 1,
               "name": "javascript",
               "templateId": 1
            },
            {
               "id": 2,
               "name": "basics",
               "templateId": 1
            }
         ]
      }
   ]
}
```

- **Example request (POST):**

```
{
        “query”: “Python”
}
```

- **Example Parameters – Key:** userID **Value:** 2
- **Response (POST):**

```
{
   "savedTemplates": [
      {
         "title": "Python Loops",
         "explanation": "Template demonstrating basic loops in Python.",
         "tags": [
            {
               "id": 7,
               "name": "python",
```

```
                "templateId": 3
            },
            {
                "id": 8,
                "name": "loops",
                "templateId": 3
            },
            {
                "id": 9,
                "name": "beginner",
                "templateId": 3
            }
        ]
    },
    {
        "title": "Django Project Setup",
        "explanation": "Steps to set up a basic Django project.",
        "tags": [
            {
                "id": 19,
                "name": "django",
                "templateId": 7
            },
            {
                "id": 20,
                "name": "python",
                "templateId": 7
            },
            {
                "id": 21,
                "name": "webdevelopment",
                "templateId": 7
            }
        ]
    }
  ]
}
```


- **User story:** As a user, I want to view and search through my list of my saved templates, including their titles, explanations, and tags, so that I can easily find and reuse them.

3. **Edit Template**
   - **URL:** `/api/code-templates/edit-template?id={template id}`
   - **Method:** `PUT, DELETE`
   - **Headers:** `Authorization: Bearer {accessToken}`
   - **Parameters – Key:** templateId **Value:** {template id}

- **Description:** This endpoint allows authenticated users to update (PUT) or delete (DELETE) the code template with the given template id. Only authorised users with valid tokens can modify or delete a template.
- **Payload**: Note: At least one field must be provided for an update; in this example, the tags field is required, while all other fields are optional.

  ```
  {
    "title": "string” (optional),
    "explanation": "string” (optional),
    "tags": ["string"],
    "code": "string” (optional)
  }
  ```
- **Example request (PUT):**

  ```
  {
     "title": "Pythony Loops"
  }
  ```
- **Example Parameters – Key:** templateId **Value:** 7
- **Response (PUT):**

  ```
  {
     "message": "Code template successfully updated.",
     "template": {
        "id": 7,
        "title": "Pythony Loops",
        "explanation": "Steps to set up a basic Django project.",
        "code": "django-admin startproject myproject",
        "isFork": false,
        "forkedFromId": null,
        "userId": 2
     }
  }
  ```
- **Example request (DELETE):** Empty
- **Example Parameters – Key:** templateId **Value:** 7
- **Response (DELETE):**

  ```
  {
     "message": "Code template deleted successfully!"
  }
  ```

- **User story:** As a user, I want to edit an existing code template’s title, explanation, tags, and code, or delete it entirely.

4. **Run and Modify**
   - **URL:** `/api/code-templates/run-modify`
   - **Method:** `POST`
   - **Headers:** `Authorization: Bearer {accessToken} (optional)`
   - **Description:** allows authenticated users to run and modify an existing template. If saveAsFork is set to true, the modified template is saved as a fork of the original. Also allows visitors (non-authenticated users) to modify an existing template without saving it as a forked version.
   - **Payload**:

- {
  "templateId": "integer",
  "modifiedCode": "string",
  "saveAsFork": "boolean"
  }
- **Example request:**

```
{
   "templateId": 3,
   "modifiedCode": "console.log('testing template 3 fork save!');",
   "saveAsFork": true
}
```

- **Response:**

```
{
   "message": "Template saved as a forked version!",
   "template": {
      "id": 12,
      "title": "Node.js Basic Server (Fork)",
      "explanation": "Forked version of template ID 3",
      "code": "console.log('testing template 3 fork save!');",
      "isFork": true,
      "forkedFromId": 3,
      "userId": 16,
      "tags": [
         {
            "id": 25,
            "name": "nodejs",
            "templateId": 12
         },
         {
            "id": 26,
            "name": "server",
            "templateId": 12
         },
         {
            "id": 27,
            "name": "backend",
            "templateId": 12
         }
      ]
   }
}
```

- **User story:** As a visitor, I want to use an existing code template, run or modify it, and if desired, save it as a new template with a notification that it's a forked version, so I can build on others' work. Saving a template is only available to authenticated users.

5. **Search**

- **URL:** `/api/code-templates/search`
- **Method:** `POST`
- **Headers:** `None`
- **Description:** Searches for code templates based on a provided query. Matches are found within the title, explanation, code, or tags associated with each template.
- **Payload**:
- **Example request:**

```
{
  "query": "sorting algorithm"
}
```

- **Response:**


```
{
  "codeTemplates": [
    {
      "id": 4,
      "title": "Django Project Setup",
      "explanation": "Steps to set up a basic Django project.",
      "code": "django-admin startproject myproject",
      "tags": [
        {
          "name": "django"
        },
        {
          "name": "python"
        },
        {
          "name": "webdevelopment"
        }
      ]
    },
    {
      "id": 5,
      "title": "Django Project Setup",
      "explanation": "Steps to set up a basic Django project.",
      "code": "django-admin startproject myproject",
      "tags": [
        {
          "name": "django"
        },
        {
          "name": "python"
        },
        {
          "name": "webdevelopment"
        }
```

```
        ]
      },
      {
        "id": 11,
        "title": "Django Project Setup (Fork)",
        "explanation": "Forked version of template ID 4",
        "code": "console.log('ANOTHER FORK');",
        "tags": [
          {
            "name": "django"
          },
          {
            "name": "python"
          },
          {
            "name": "webdevelopment"
          }
        ]
      },
    ]
}
```


- **User story**: As a visitor, I want to search through all available templates by title, tags, or content so that I can quickly find relevant code for my needs.

Blog Posts Endpoints

1. **Create Blog**
   - **URL:** `/api/blog-posts/create-blog`
   - **Method:** `POST, PUT, DELETE`
   - **Headers:** `Authorization: Bearer {accessToken}`
   - **Description:** Handles blog post creation, updating, and deletion. Users can create a new blog post, edit an existing one, or delete a blog post. Only one operation can be performed per request. User authentication is required.
   - **Payload (POST)**:


```
{
  "title": "string",
  "description": "string",
  "tags": ["string"] (Optional),
  "templateIds": [“interger”]  (Optional)
}
```


   - **Example request (POST):**


```
{
  "title": "New: Introduction to JavaScript",
  "description": "This blog post provides an introduction to JavaScript, covering the basics of the language.",
```

```
  "tags": ["javascript", "basics", "programming"],
  "templateIds": [1, 2]
}
```

- **Example response** (**POST)**


```
{
   "message": "Blog post created successfully!",
   "blogPost": {
      "id": 3,
      "title": "New: Introduction to JavaScript",
      "description": "This blog post provides an introduction to JavaScript, covering the basics of the language.",
      "userId": 3,
      "upvoteCount": 0,
      "downvoteCount": 0,
      "tags": [
         {
            "id": 7,
            "name": "javascript",
            "blogPostId": 3
         }
      ],
      "templates": [
         {
            "id": 1,
            "title": "C++ Sorting Algorithm",
            "explanation": "Template for implementing sorting algorithms in C++.",
            "code": "#include <iostream> #include <vector> void bubbleSort(std::vector<int>& arr) { for (size_t i = 0; i < arr.size(); i++) { for (size_t j = 0; j < arr.size() - i - 1; j++) { if (arr[j] > arr[j + 1]) std::swap(arr[j], arr[j + 1]); } } } int main() { std::vector<int> arr = {5, 1, 4, 2, 8}; bubbleSort(arr); for (int num : arr) std::cout << num << ' '; return 0; }",
            "isFork": false,
            "forkedFromId": null,
            "userId": 3
         }
      ]
   }
}
```


- **Payload (PUT)**: Note: follow same structure as POST for PUT

```
{
   "title": "string",
   "description": "string",
   "tags": ["string"] (Optional),
   "templateIds": [1, 2]  (Optional)
}
```

- **Example request (PUT)**

```
{
    "id": 6,
  "title": "EDITED TITLE: REST API with Node.js",
  "description": "EDITED: REST API and Express and Node.js for developers",
  "tags": ["nodejs", "api", "rest"],
  "templateIds": [3, 5, 1]
}
```

Example response (PUT):


```
{
    "message": "Blog post updated successfully!",
    "blogPost": {
        "id": 6,
        "title": "EDITED TITLE: REST API with Node.js",
        "description": "EDITED: REST API and Express and Node.js for
developers",
        "userId": 16,
        "tags": [
            {
                "id": 25,
                "name": "nodejs",
                "blogPostId": 6
            },
            {
                "id": 26,
                "name": "api",
                "blogPostId": 6
            },
            {
                "id": 27,
                "name": "rest",
                "blogPostId": 6
            }
        ],
        "templates": [
            {
                "id": 1,
                "title": "My Sample Template",
                "explanation": "This is an example of a template with code and
tags.",
                "code": "console.log('Hello, world!');",
                "isFork": false,
                "forkedFromId": null,
                "userId": 16
            }
```

```
        ]
    }
}
```

- **Payload (DELETE)**:

```
{
  "id": “integer” // Replace with the blog post ID you want to delete
}
```

- **Example request (DELETE):**

```
{
  "id": 7
}
```

- **Example response (DELETE):**

```
{
    "message": "Blog post deleted successfully!"
}
```

- **User story:** As a user, I want to create/edit/delete blog posts. A blog post has title, description, and tag. It might also include links to code templates (either mine or someone else’s)

2. **Search Blog**
    - **URL**: `/api/blog-posts/search-blog`
    - **Method**: `POST`
    - **Description**: Searches for blog posts (that are not hidden) that match a given query in their titles, descriptions, tags, or associated templates. Supports pagination to limit the number of results returned per request. Each blog post includes links to associated code templates, enabling visitors to view, run, or fork the templates directly.
    - **Headers**: `None`
    - **Payload**:

```
{

    "query": "string",        // Required. The search keyword.

    "page": int, (optional, default: 1)

    "pageSize": int,  (optional default: 10).

}
```

- **Example request**:

```
{
   "query": "API",
   "page": 1,
   "pageSize": 2
}
```

- **Response:**


```
{
   "currentPage": 1,
   "pageSize": 2,
   "totalItems": 6,
   "totalPages": 3,
   "blogPosts": [
      {
         "id": 6,
         "title": "EDITED TITLE: REST API with Node.js",
         "description": "EDITED: REST API and Express and Node.js for developers",
         "user": {
            "firstName": "Test",
            "lastName": "This"
         },
         "tags": [
            {
               "name": "nodejs"
            },
            {
               "name": "api"
            },
```

```
        {
          "name": "rest"
        }
      ],
      "templates": [
        {
          "id": 3,
          "title": "Node.js Basic Server",
          "link": "/api/templateActions?id=3&action=view"
        }
      ]
    },
    {
      "id": 7,
      "title": "Building APIs with Django",
      "description": "Learn how to build RESTful APIs using Django and DRF.",
      "user": {
        "firstName": "Jane",
        "lastName": "Doe"
      },
      "tags": [
        {
          "name": "django"
        },
        {
          "name": "api"
        },
        {
          "name": "rest"
```

```
            }
        ],
        "templates": [
            {
                "id": 4,
                "title": "Django Basic API",
                "link": "/api/templateActions?id=4&action=view"
            }
        ]
    }
]
}
```


- **User story:** As a visitor, I want to browse and read blog posts so that I can learn from others' experiences and code examples. I want to search through blog posts by their title, content, tags, and also the code templates they contain. Also, I want to follow links from a blog post directly to the relevant code template so that I can view, run, or fork the code discussed.

3. **View Blog**
   - **URL:** `/api/blog-posts/view-blog?templateId={template id}`
   - **Method:** `GET`
   - **Parameters – Key:** templateId **Value:** {template id}
   - **Headers:** None
   - **Description:** allows a visitor to see each blog post that mentions the given code template
   - **Payload**: None
   - **Example request:** Empty
   - **Example parameters – Key:** templateId **Value:** 11
   - **Response:**


```
{
    "template": {
        "id": 11,
        "title": "C++ Sorting Algorithm",
        "explanation": "Template for implementing sorting algorithms in C++.",
```

"code": "#include <iostream> #include <vector> void bubbleSort(std::vector<int>& arr) { for (size_t i = 0; i < arr.size(); i++) { for (size_t j = 0; j < arr.size() - i - 1; j++) { if (arr[j] > arr[j + 1]) std::swap(arr[j], arr[j + 1]); } } } int main() { std::vector<int> arr = {5, 1, 4, 2, 8}; bubbleSort(arr); for (int num : arr) std::cout << num << ' '; return 0; }",
"isFork": false,
"forkedFromId": null,
"userId": 2,
"blogPosts": [
{
"id": 2,
"title": "Blog 1: Introduction to JavaScript",
"description": "This blog post provides an introduction to JavaScript, covering the basics of the language.",
"userId": 2
}
]
}
}

- **User story:** As a visitor, I want to see the list of blog posts that mention a code template on the template page.

4. **Create Comment**
   - **URL:** `/api/blog-posts/create-comment`
   - **Method:** `POST`
   - **Headers:** `Authorization: Bearer {accessToken}`
   - **Description:** Allows a user to comment or reply to existing comments on a blog post, as long as the blog post and parent comment are not hidden. If the blog post and/or the parent comment is hidden, the user must be the original author of the hidden content.
   - **Payload**: Note: only add a parentId when creating a reply
     {
     “content”: “string”,
     “blogPostId”: int,
     “parentId”: int (optional),
     “rating”: int
     }
   - **Example request (creating a comment):**
     {
     “content”: “This is a great blog post! Thanks for sharing!”,
     “blogPostId”: 3,
     “rating”: 4
     }
   - **Response (creating a comment):**
     {

```
    "message": "Comment created successfully!",
    "comment": {
        "id": 4,
        "content": "This is a great blog post! Thanks for sharing!",
        "userId": 2,
        "blogPostId": 3,
        "parentId": null,
        "rating": 4,
        "blogPost": {
            "id": 3,
            "title": "Blog 1: Introduction to JavaScript",
            "description": "This blog post provides an introduction to JavaScript,
covering the basics of the language.",
            "userId": 2
        }
    }
}
```


- **Example request (creating a reply):**


```
{
“content”: “Nice Post!”,
“blogPostId”: 3,
“parentId”: 4,
“rating”: 1
}
```


- **Response (creating a reply):**


```
{
    "message": "Comment created successfully!",
    "comment": {
        "id": 9,
        "content": "Nice post!",
        "userId": 2,
        "blogPostId": 3,
        "parentId": 4,
        "rating": 1,
        "blogPost": {
            "id": 3,
            "title": "Blog 1: Introduction to JavaScript",
            "description": "This blog post provides an introduction to JavaScript,
covering the basics of the language.",
            "userId": 2
        },
        "parent": {
            "id": 4,
            "content": "This is a great blog post! Thanks for sharing!",
            "userId": 2,
            "blogPostId": 3,
            "parentId": null,
            "rating": 4
```

```
    }
  }
}
```

- **User story:** As a user, I want to comment, or reply to existing comments on a blog post so that I can engage with the author and other readers by sharing feedback, asking questions, or starting discussions.

5. **Rating**
   - **URL:** `/api/blog-posts/ratings`
   - **Method:** `POST`
   - **Headers:** `Authorization: Bearer {accessToken}`
   - **Description:** Allows authenticated users to upvote or downvote either a blog post or a comment. Users can only rate one item (blog post or comment) per request, and each user can have only one rating per item, which can be modified afterward.
   - **Payload**:

```
{
  "blogPostId": "integer (required if rating a blog post)",
  "commentId": "integer (required if rating a comment)",
  "isUpvote": "boolean (required)"
}
```

   - **Example request:**

```
{
 "commentId": 5,
 "isUpvote": true
}
```

   - **Response:**

```
{
  "message": "Comment rating updated successfully!",
  "upvotes": 2,
  "downvotes": 0
}
```

   - **User story:** As a user, I want to rate blog posts and comments with upvotes or downvotes so that I can express my agreement or disagreement with the content.

6. **Sorted**
   - **URL**: `/api/blog-posts/sorted`
   - **Method**: `GET`
   - **Description**: Retrieves blog posts and their associated comments sorted by rating scores. This endpoint returns a list of blog posts with the highest-rated content appearing first.
   - **Headers**: `None`

- **Payload**: None (GET request does not require a body)
- **Example request**:None (GET request does not require a body)
- **Response:**


```
{
  "message": "Blog posts and comments sorted by rating retrieved successfully!",
  "blogPosts": [
    {
     "id": 6,
      "title": "REST API with Node.js",
      "description": "Overview of creating REST APIs with Node.js and Express.",
      "upvoteCount": 10,
      "downvoteCount": 2,
      "comments": [
        {
          "id": 2,
          "content": "Insightful post!",
          "upvoteCount": 5,
          "downvoteCount": 0,
          "ratingScore": 5
        },
        {
          "id": 3,
          "content": "Very helpful, thanks!",
          "upvoteCount": 3,
          "downvoteCount": 1,
          "ratingScore": 2
        }
      ],
      "ratingScore": 8
    }
```

```
  ]

}
```

- **User story:** As a visitor, I want to see the list of blog posts and comments sorted by their ratings so that I get exposed to the most valued or controversial discussions first.

Inappropriate Content Reports Endpoints

1. **Report**
   - **URL:** `/api/icr/report`
   - **Method:** `POST`
   - **Headers:** `Authorization: Bearer {accessToken}`
   - **Description:** Report an inappropriate blog post or comment so that the website is purged of abusive content. Must add additional explanation when submitting my report.
   - **Payload (reporting a blog post)**: Note: You can only report either a blog post or a comment, not both.
     {
     “blogPostId”: int,
     “reason”: “string”
     }
   - **Payload (reporting a comment)**:
   - {
     “commentId”: int ,
     “reason”: “string”
   - }
   - **Example request (reporting a blog post):**
     {
     “blogPostId”: 1,
     “reason”: “The blog has harmful intent.”
     }
   - **Response (reporting a blog post):**

```
{
   "message": "Report for blog post submitted successfully!",
   "report": {
      "id": 1,
      "reason": "The blog has harmful intent.",
      "userId": 2,
      "blogPostId": 1,
      "commentId": null
   }
}
```

   - **Example request (reporting a comment):**
     {
     “commentId”: 1,

“reason”: “The comment is abusive towards author.”
}

- **Response (reporting a comment):**
  {
  "message": "Report for comment submitted successfully!",
  "report": {
  "id": 2,
  "reason": "The comment is abusive towards author.",
  "userId": 2,
  "blogPostId": null,
  "commentId": 1
  }
  }

- **User story:** As a user, I want to report an inappropriate blog post or comment so that the website is purged of abusive content. I want to add additional explanation when submitting my report.

2. **Hide Content**
   - **URL:** `/api/icr/hide-content`
   - **Method:** `PUT`
   - **Headers:** `Authorization: Bearer {accessToken}`
   - **Description:** Allows a system admin to hide either an inappropriate blogPost or inappropriate comment for everyone except the author
   - **Payload**: Note: the ids are optional but at least one of either blog posts or comments must have a value to hide their content.
     {
     “blogPostId”: int (optional),
     “commentId”: int (optional)
     }
   - **Example request (hiding only a blog post):**
     {
     “blogPostId”: 2
     }
   - **Response (hiding only a blog post):**
     {
     "message": "Content hidden successfully!"
     }
   - **Example request (hiding only a comment):**
     {
     “commentId”: 4
     }
   - **Response (hiding only a comment):**
     {
      "message": "Content hidden successfully!"
     }

- **User story:** As the system administrator, I want to sort blog posts and comments based on the total number of reports they received so that I can find the inappropriate content easier.

3. **Admin Sort**
   - **URL:** `/api/icr/admin-sort`
   - **Method:** `GET`
   - **Headers:** `Authorization: Bearer {accessToken}`
   - **Description:** Allows an admin to retrieve and sort blog posts and comments based on the number of reports associated with each. This helps prioritize content moderation by listing potentially inappropriate content with the highest report counts at the top.
   - **Payload**: None required
   - **Example request:** GET
   - **Response:**

```
{
  "blogPosts": [
    {
      "id": 1,
      "title": "First Blog Post",
      "description": "This is the first blog post.",
      "reports": [
        { "id": 10, "reason": "Inappropriate language" },
        { "id": 12, "reason": "Off-topic content" }
      ],
      "_count": {
        "reports": 2
      }
    }
  ],
  "comments": [
    {
      "id": 5,
      "content": "This is an inappropriate comment.",
      "reports": [
        { "id": 20, "reason": "Harassment" },
        { "id": 21, "reason": "Inappropriate language" }
      ],
      "_count": {
        "reports": 2
      }
    }
  ]
}
```

   - **User story:** As the system administrator, I want to hide content that I deem inappropriate so that Scriptorium remains safe for everyone. This content

would then be hidden from everyone, except for the author. The author can still see the content (with a flag that indicates the reports), but cannot edit it.