Model Design

```
model User {
  id               Int        @id @default(autoincrement())
  firstName        String
  lastName         String
  email            String     @unique
  password         String
  phoneNumber      String?
  profilePicture   String?
  templates        Template[]
  reports          Report[]
  createdAt        DateTime   @default(now())
  updatedAt        DateTime   @updatedAt
  liked            BlogPost[] @relation("likedBlogPosts")
  disliked         BlogPost[] @relation("dislikedBlogPosts")
  likedComments    Comment[]  @relation("likedComments")
  dislikedComments Comment[]  @relation("dislikedComments")
  role             String     @default("user")
}
```

User model explanation: id is the user's id, firstName and lastName are the user's name, email is obviously the email they used to sign up, password holds their hashed password, phoneNumber is optional to hold their phone number, profilePicture holds the URL of their profile picture, templates is a list of their code templates, reports is a list of their content reports, created and updated at are when they signed up and updated their profile, liked and disliked are the blog posts that they have liked and disliked, likedComments and dislikedComments are the same but for comments, and role is whether they are an admin or just a user.

```
model Template {
  id          Int        @id @default(autoincrement())
  title       String
  code        String
  language    String
  explanation String     @default("")
  tags        Tag[]      @relation("TemplateTags")
  authorID    Int
  author      User       @relation(fields: [authorID],
references: [id])
  forked      Boolean    @default(false)
  forkedId    Int?
  forkedFrom  Template?  @relation("ForkRelation", fields:
[forkedId], references: [id])
  forks       Template[] @relation("ForkRelation")
  blogPosts   BlogPost[] @relation("BlogPostTemplates")
}
```

Title is the tile of the template, code is the code of the template, language is the language the code is written in, explanation is the explanation, tags is a list of tags provided by the usr, authorID is the the author's ID, forked is true if this template is forked for a previous one, and forkedId is the Id of the template it was forked from, if applicable.

```
model Tag {
  id        Int        @id @default(autoincrement())
  name      String     @unique
  templates Template[] @relation("TemplateTags")
  blogPosts BlogPost[] @relation("BlogPostTags")
}
```

Tags are used for blog posts and templates. They have names which must be unique, and then a list of the templates and blog posts that they are associated with.

```
model BlogPost {
  id          Int        @id @default(autoincrement())
  title       String
  description String
  tags        Tag[]      @relation("BlogPostTags")
  authorId    Int
  comments    Comment[]
  rating      Int        @default(0)
  hidden      Boolean    @default(false)
  reportCount Int        @default(0)
  createdAt   DateTime   @default(now())
  updatedAt   DateTime   @updatedAt
  liked       User[]     @relation("likedBlogPosts")
  disliked    User[]     @relation("dislikedBlogPosts")
  templates   Template[] @relation("BlogPostTemplates")
}
```

Blog posts have a title, description, tags, authorId, and then comments is a list of comments on the blog post, rating is calculated by doing the length of liked - length of disliked. Hidden is true if the blog has been hidden by an admin. reportCount is the number of reports about the post.

Liked and disliked are the users that have liked/disliked the post, and templates holds a list of the templates mentioned in the blog post.

```
model Comment {
  id          Int       @id @default(autoincrement())
  content     String
  authorId    Int
  postId      Int
  parentId    Int? // Nullable, since top-level comments don’t
have a parent
  post        BlogPost  @relation(fields: [postId], references:
[id], onDelete: Cascade)
  rating      Int       @default(0)
  liked       User[]    @relation("likedComments")
  disliked    User[]    @relation("dislikedComments")
  createdAt   DateTime  @default(now())
  parent      Comment?  @relation("CommentReplies", fields:
[parentId], references: [id])
  Replies     Comment[] @relation("CommentReplies")
  hidden      Boolean   @default(false)
  reportCount Int       @default(0)
}
```

Comments have content which is the content of the comment, parentId is the id of the comment this comment replies to, post is the blog post this comment was made on, rating is calculated in the same way as blog posts. Replies is a list of comments that have this comment as their parent.

```
model Report {
  id          Int      @id @default(autoincrement())
  explanation String
  userId      Int
  user        User     @relation(fields: [userId], references:
[id])
  contentId   Int //Id of reported content (BlogPost or Comment)
  contentType String //Either post or comment
  createdAt   DateTime @default(now())
}
```

Reports have an explanation explaining what the report is about, contentId is the Id of the comment/blog post that this report is about, and contentType specifies whether the content is a blog or a comment.

API Endpoints

/api/templates

- Allowed request types: GET, POST
    - GET: Returns a list of code templates that match the parameters in the request body. The allowed parameters in a request are: id (a template id), title (template title), code (code in the template), tags (tags of the template, must be separated by commas), authorId (id of a user), and page (the page number, 10 templates will be sent at a time). Note that when searching by title, code, or tags, all templates that contain the query will be returned. For example, for a request with title: “Java”, all templates whose title contains the word “Java” will be returned.
      Example GET request body:
      ```
      { "title": "Python", "code": "print", "tags":
      "interesting, great"}
      ```
      Response:
      ```
      {"results": [{
                  "id": 1,
                  "title": "Python Script",
                  "code": "print(\"Hello World\")",
                  "language": "python",
                  "explanation": "A cool Python script",
                  "authorID": 1,
                  "forked": false,
                  "forkedId": null,
                  "author": {
                      "id": 1
                  },
                  "tags": [
                      {
                          "id": 3,
                          "name": "interesting"
                      },
                      {
                          "id": 4,
                          "name": "great"
                      }
                  ]}]}
      ```
    - POST: Uploads a new code template and returns the new template. This is an authenticated route, so the user must send their token in the authorization header. Requests must include title (title of template), code (code in the template), language (the language the code is written in), explanation (an explanation of the

template), tags (tags for this template, separated by commas), forked (true if this template is a fork of another template, false otherwise), and forkedId (the id of the template this one was forked from, if applicable).
Example POST request body:
```
{

    "title": "First Template",
    "code": "print('Hello World')",
    "language": "python",
    "explanation": "My first template ever",
    "tags": "simple,awesome,cool",
    "forked": "false"

}
```
Response:
```
{
    "id": 2,
    "title": "First Template",
    "code": "print('Hello World')",
    "language": "python",
    "explanation": "My first template ever",
    "authorID": 1,
    "forked": false,
    "forkedId": null
}
```

/api/templates/[id]

- Allowed request types: GET, PUT, DELETE
    - GET: Returns the template with the id from the URL.
      Example GET request with id = 2 response:
```
{
    "id": 2,
    "title": "First Template",
    "code": "print('Hello World')",
    "language": "python",
    "explanation": "My first template ever",
    "authorID": 1,
    "forked": false,
    "forkedId": null,
    "author": {
        "id": 1
    },
    "tags": [
        {
            "id": 5,
            "name": "simple"
        },
        {
            "id": 6,
            "name": "awesome"
        },
        {
            "id": 7,
            "name": "cool"
        }
    ],
    "blogPosts": []
}
```

- PUT: Updates the template with the id from the URL. This is an authenticated route, so the user must send their token in the authorization header. Obviously users can only update templates they created themselves. The fields that can be updated are title, explanation, tags and code.
  Example PUT request body with id = 2:
  ```
  { "title": "First Template Updated"}
  ```
  Response:
  ```
  {
      "id": 2,
      "title": "First Template Updated",
      "code": "print('Hello World')",
      "language": "python",
      "explanation": "My first template ever",
      "authorID": 1,
      "forked": false,
      "forkedId": null
  }
  ```
- DELETE: Deletes the template with the id from the URL. This is an authenticated route, so the user must send their token in the authorization header. Obviously users can only delete templates they created themselves. Returns the deleted template.
  Example DELETE request with id = 2 response:
  ```
  {
  ```

```
        "id": 2,
        "title": "First Template Updated",
        "code": "print('Hello World')",
        "language": "python",
        "explanation": "My first template ever",
        "authorID": 1,
        "forked": false,
        "forkedId": null
    }
```

/api/blog

- Allowed request types: GET, POST
    - GET: Returns a list of blog posts that match the parameters in the request body. The allowed parameters in a request are: id (a blog post id), title (post title), description (content in the blog post), tags (tags of the blog post, must be separated by commas), templateIds (the ids of templates the blog posts should mention), authorId (id of a user), and page (the page number, 10 blog posts will be sent at a time). Note that when searching by title, code, or tags, all posts that contain the query will be returned. For example, for a request with title: “Java”, all posts whose title contains the word “Java” will be returned. The blog posts will be sorted by rating, from best to worst.
      Example GET request body:
      `{"title": "Post"}`
      Response:

```
[
    {
        "id": 2,
        "title": "Next Post",
        "description": "skfjlgfslkgj",
        "authorId": 1,
        "rating": 0,
        "createdAt": "2024-11-02T20:35:52.514Z",
        "updatedAt": "2024-11-02T20:35:21.246Z",
        "comments": [],
        "tags": [
            {
                "id": 1,
                "name": "coding"
            }
        ]
    },
    {
        "id": 1,
        "title": "First Blog Post",
        "description": "This is the description of my
first blog post.",
        "authorId": 1,
        "rating": -1,
        "createdAt": "2024-11-01T22:46:52.919Z",
        "updatedAt": "2024-11-02T20:32:52.831Z",
        "comments": [
            {
                "id": 1,
                "content": "This is a top-level
comment.",
                "authorId": 1,
                "postId": 1,
                "parentId": null,
                "rating": 0,
                "createdAt":
"2024-11-01T22:52:59.097Z"
            },
            {
                "id": 2,
                "content": "This is a top-level
comment.",
                "authorId": 1,
                "postId": 1,
                "parentId": null,
                "rating": 0,
                "createdAt":
"2024-11-01T22:59:12.391Z"
            },
            {
                "id": 3,
                "content": "This is a reply to the
top-level comment.",
                "authorId": 2,
                "postId": 1,
                "parentId": 1,
                "rating": 0,
                    "createdAt":
"2024-11-01T23:00:21.208Z"
                }
            ],
            "tags": [
                {
                    "id": 1,
                    "name": "coding"
                },
                {
                    "id": 2,
                    "name": "tutorial"
                }
            ]
    }
]
```

- POST: Creates a new blog post and returns the new post. This is an authenticated route, so the user must send their token in the authorization header. The body of the request should include: title (title of the post), description (the content of the blog post), tags (tags of the post, separated by commas), and templateIds (the ids of the templates mentioned in the blog post).
  Example POST request body:
  ```
  {
      "title": "Great post",
      "description": "This post is really awesome and
  cool.",
      "tags": "cool",
      "templateIds": "1",
  }
  ```
  Response:
  ```
  {
      "id": 3,
      "title": "Great post",
      "description": "This post is really awesome and
  cool.",
      "authorId": 1,
      "rating": 0,
      "createdAt": "2024-11-02T23:22:17.778Z",
      "updatedAt": "2024-11-02T23:22:17.778Z",
      "tags": [
          {
  ```

```
                "id": 7,
                "name": "cool"
            }
        ]
    }
```

/api/blog/[id]:

- Allowed request types: GET, POST, PUT, DELETE
    - GET: Returns the blog post with the specified id.
      Example GET request with id = 1 response:

```
{
    "id": 1,
    "title": "First Blog Post",
    "description": "This is the description of my
first blog post.",
    "authorId": 1,
    "rating": -1,
    "createdAt": "2024-11-01T22:46:52.919Z",
    "updatedAt": "2024-11-02T20:32:52.831Z",
    "comments": [
        {
            "id": 1,
            "content": "This is a top-level comment.",
            "authorId": 1,
            "postId": 1,
            "parentId": null,
            "rating": 0,
            "createdAt": "2024-11-01T22:52:59.097Z"
        },
        {
            "id": 2,
            "content": "This is a top-level comment.",
            "authorId": 1,
            "postId": 1,
            "parentId": null,
            "rating": 0,
            "createdAt": "2024-11-01T22:59:12.391Z"
        },
        {
            "id": 3,
            "content": "This is a reply to the
top-level comment.",
            "authorId": 2,
            "postId": 1,
            "parentId": 1,
            "rating": 0,
            "createdAt": "2024-11-01T23:00:21.208Z"
        }
    ],
    "tags": [
        {
            "id": 1,
            "name": "coding"
        },
        {
            "id": 2,
            "name": "tutorial"
        }
    ],
    "liked": [],
    "disliked": [
        {
            "id": 1
        }
    ]
}
```

- POST: Likes or dislikes the post. This is an authenticated route, so the user must send their token in the authorization header. Request bodies must include rating (1 for like, -1 for dislike). If the user has already liked the post and tries to like it again, they will unlike it (same for dislike). Users can also choose to like posts they previously disliked, or vice versa.
  Example POST request with id = 1:
  ```
  {
      "rating": "1"
  }
  ```
  Response:
  ```
  {
      "id": 1,
      "title": "First Blog Post",
      "description": "This is the description of my
  first blog post.",
  ```

```
    "authorId": 1,
    "rating": 1,
    "createdAt": "2024-11-01T22:46:52.919Z",
    "updatedAt": "2024-11-02T23:36:22.086Z"
}
```

- PUT: Updates the post and returns the updated post. This is an authenticated route, so the user must send their token in the authorization header. Obviously users can only update posts they created themselves. Updateable fields are title, description, tags and templateIds.
  Example PUT request with id = 1:
  ```
  { "title": "New title" }
  ```
  Response:
  ```
  {
      "id": 1,
      "title": "New title",
      "description": "This is the description of my
  first blog post.",
      "authorId": 1,
      "rating": 1,
      "createdAt": "2024-11-01T22:46:52.919Z",
      "updatedAt": "2024-11-03T00:00:41.438Z"
  }
  ```
- DELETE: Deletes the post. This is an authenticated route, so the user must send their token in the authorization header. Obviously users can only delete posts they created themselves.
  Example DELETE request with id = 1 response:
  Success message, returns with 200 code.

/api/blog/[id]/comments:

- Allowed request types: GET, POST
  - GET: Returns all comments on the blog post with the specified id, sorted from highest to lowest rating.
    Example GET request with id = 2 response:
    ```
    [
        {
            "id": 4,
            "content": "Cool post bro",
            "authorId": 1,
            "postId": 2,
            "parentId": null,
            "rating": 0,
    ```

```
            "createdAt": "2024-11-03T00:21:13.751Z",
            "Replies": []
        }
    ]
```
  - POST: Uploads a new comment to the post with the specified id. The request body must contain content (the content of the comment). If this comment is a reply to an existing comment, it should also include parentId (id of the comment which it is replying to). User needs to be authenticated.
    Example POST request body with id = 2:
    ```
    {
        "content": "Cool post bro"
    }
    ```
    Response:
    ```
    {
        "id": 4,
        "content": "Cool post bro",
        "authorId": 1,
        "postId": 2,
        "parentId": null,
        "rating": 0,
        "createdAt": "2024-11-03T00:21:13.751Z"
    }
    ```

/api/blog/[id]/comments/[commentId]:

- Allowed request types: POST
  - POST: Likes or dislikes the comment with the specified commentId and returns the comment. This is an authenticated route, so the user must send their token in the authorization header. Request body must contain a rating (1 for like, -1 for dislike). If the user has already liked, trying to like again will unlike the post (same for dislike). Users can also like a comment they previously disliked, and vice versa.
    Example POST request body:
    ```
    {
        "rating": "1"
    }
    ```
    Response:
    ```
    {
        "id": 4,
        "content": "Cool post bro",
        "authorId": 1,
        "postId": 2,
    ```

```
        "parentId": null,
        "rating": 1,
        "createdAt": "2024-11-03T00:21:13.751Z"
    }
```

/api/auth/signup:

- Allowed request types: POST
    - POST: Creates new user with given user fields: firstName, lastName, email, password, phoneNumber, profilePicture. First checks if the user already exists and if not, the password is hashed and the user is created. Token is also generated.
      Example POST request body:
      ```
      {
        "firstName": "John",
        "lastName": "Doe",
        "email": "john.doe@example.com",
        "password": "yourpassword123",
        "phoneNumber": "1234567890"
      }
      ```
      Response:
      ```
      {
          "token":
      "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VySWQiOjEs
      ImlhdCI6MTczMDYwOTY2OSwiZXhwIjoxNzMxMjE0NDY5fQ.qvPNWb2
      jRJ_Kf5dUt0P32aQUzuhwu2nFjMscApL38z4",
          "user": {
              "id": 1,
              "email": "john.doe@example.com"
          }
      }
      ```

/api/auth/login:

- Allowed request types: POST
    - POST: Matches request email with user to find a unique account and compares password with the password for that user. If there is a successful match a token is generated.
      Example POST request body:
      ```
      {
        "email": "john.doe@example.com",
        "password": "yourpassword123"
      ```

```
}
```
Response:
```
{
    "token":
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VySWQiOjEs
ImlhdCI6MTczMDYxMDIxNiwiZXhwIjoxNzMxMjE1MDE2fQ.FqLIb9C
Ut7UfFg7GZ8bW1l0dUsxrwRZuZCM7eFPEy9Q",
    "user": {
        "id": 1,
        "email": "john.doe@example.com"
    }
}
```

/api/auth/profile:

- Allowed request types: PUT
  - PUT: Allows users to edit any field in their profile (firstName, lastName, phoneNumber, profilePicture - if they have a URL). The user must be logged. Use token from login response in authorization header (Token Bearer <token>)
    Example PUT request body:
```
{
  "firstName": "John",
  "lastName": "Doe",
  "phoneNumber": "0987654321",
  "profilePicture": "url_to_profile_picture" // This could
be the Cloudinary URL after uploading
}
```

Response:
```
{
    "message": "User profile updated successfully",
    "user": {
        "id": 1,
        "firstName": "John",
        "lastName": "Doe",
        "email": "john.doe@example.com",
        "password":
"$2a$10$kgDIaXZNuCIXyUkSkwrc5OYVZ6MHr/aN4s2pN.jOuBPgUg1kxv
C.G",
        "phoneNumber": "0987654321",
        "profilePicture": "url_to_profile_picture",
        "createdAt": "2024-11-03T04:54:29.459Z",
        "updatedAt": "2024-11-03T14:46:07.761Z",
        "role": "user"
    }
}
```

/api/user/upload-avatar:

- Allowed request types: POST
    - POST: Allows authenticated users to upload and set a profile picture for their account. Users must include a valid JWT token in the Authorization header. This token verifies the user's identity. The route accepts a POST request with a profile picture file uploaded as profilePicture in the form data. The file is first stored locally. The local file is uploaded to Cloudinary, a cloud-based image hosting service, which returns a secure URL for the uploaded image. The user's profile in the database is updated with the new image URL, linking the Cloudinary image to the user account. After a successful upload, the local copy of the image file is deleted to save storage space.
      Response:
```
{
    "message": "User avatar updated successfully",
    "user": {
        "id": 1,
        "firstName": "John",
        "lastName": "Doe",
        "email": "john.doe@example.com",
        "password":
"$2a$10$kgDIaXZNuCIXyUkSkwrc5OYVZ6MHr/aN4s2pN.jOuBPgUg1kxv
C.G",
        "phoneNumber": "0987654321",
        "profilePicture":
"https://res.cloudinary.com/dvvz3o66x/image/upload/v173064
7449/y4a5iwf6hdsxpciejvgi.png",
        "createdAt": "2024-11-03T04:54:29.459Z",
        "updatedAt": "2024-11-03T15:24:09.497Z",
        "role": "user"
    }
}
```

/api/reports/reportContent:

- Allowed request types: POST
    - POST: This route allows a user to report inappropriate content whether that is a blog post or a comment. The user must be authorized using the token from login.
      Example POST request body:
      ```
      {
        "contentId": 1,
        "contentType": "post",
        "explanation": "This content is inappropriate because it
      contains offensive language."
      }
      ```

      Response:
      `201 Created`

/api/reports/fetchReports:

- Allowed request types: GET
    - GET: This route allows an admin to fetch a list of reported blog posts and comments, sorted by report count. Users must be authorized by including the token in the header.
      Response:
      ```
      201 OK
      Body: A list of reported blog posts and comments, sorted
      by the number of reports.
      ```

/api/reports/hideContent:

- Allowed request types: PUT
    - PUT: This route allows an admin to hide inappropriate content.
      Example POST request body:
      ```
      {
        "contentId": 1,
        "contentType": "post"
      }
      ```
      Response:
      ```
      201 OK
      Body: Confirmation message and details of the content that
      was hidden.
      ```

/api/execute

- Allowed request types: POST

- This endpoint allows the user to execute code in different languages (Python, JavaScript, C, C++, and Java). The output or the error is returned in the response. The user needs to specify the language and the code, and if there is stdin required.

```
{
  "code": "#include <stdio.h>\nint main() { printf(\"Hello, C
World!\\n\"); return 0; }",
 "language": "c",
 "stdin": ""
}
```

Response:

```
{
 "output": "Hello, C World!\n"
}
```