Backend Documentation

Introduction

C-LARA is a platform designed to create multimodal texts for language learners with integrated audio and translations. The current API allows users to register, log in, and manage user profiles, user can also manage their audio recordings and access different content of text to record their own audio.

Documentation Structure

For our backend documentation, we have two separate parts that offer a comprehensive overview of our system's architecture and functionality. The first part, titled "API Documentation," delves into the intricate details of our application programming interfaces, explaining the various endpoints, expected behaviors, and authentication mechanisms. This section is instrumental for developers and technical stakeholders to understand the interactions and capabilities of our services.

The second part, "Project Structure," provides an in-depth exploration of the backend's codebase. It introduces each file's purpose, its role within the larger system, and the logic it encapsulates. Whether you're a new developer getting acquainted with our project or a reviewer aiming to understand our coding practices, this section offers a roadmap to navigate the backend's structural landscape.

Together, these segments aim to provide a holistic understanding of our backend's design, functionality, and best practices, ensuring clarity and transparency for all stakeholders involved.

Repository Reference

For the complete source code, and backend design of our project, please visit our GitHub Repository.

API Documentation

Authentication

CLARA uses JWT for authentication. Upon logging in, you will receive a JWT token. This token should be included in the `Authorization` header for all subsequent requests.

Example header:

```
Authorization: Bearer <token>
```

HTTP Methods

Our API supports the following HTTP methods:

- GET
- POST
- PUT
- DELETE

Endpoints

User

1. User Registration

- **URL**: `api/user/register`
- **Method**: `POST`
- **Parameters**:
  - `username`: The desired username.
  - `email`: User's email address.
  - `password`: A strong password.
- **HTTP Responses**:
  - 200: Successfully registered.
  - 400: Incorrect data format in request, need to fill all the required fields.
  - 400: Username or email already exists.
- **Example**:
  - Request:

```
{
    "username": "Alice",
    "email": "Alice@gmail.com",
    "password": "Alice"
}
```

  - Response:

```
{
    "id": 1,
    "email": "Alice@gmail.com"
    "username": "Alice",
    "password":"Alice (SHA encrypted)"

}
```

2. User Login

- **URL**: `api/user/login`
- **Method**: `POST`
- **Parameters**:
  - `email`: Your email address.
  - `password`: Your password.
- **Response**:
  - 200: Successfully logged in.
  - 400: Incorrect data format in request, need to fill all the required fields.
  - 401: Invalid username or password, no active account found with the given credentials.
- **Example**:
  - Request:

```
{
    "username": "Alice",
```

```
    "password": "Alice"
}
```

- Response: `refresh token` and `access token`

```
{
    "refresh": "string"
    "access": "string"
}
```

3. Update User Password

- **URL**: `api/user/change-password`
- **Method**: `PUT`
- **Headers**:
  - `Authorization`: Your JWT token.
- **Parameters**:
  - `username`: Current username.
  - `email`: email address.
  - `oldpassword`: Old password (optional).
  - `newpassword`: New password (optional).
- **Response**:
  - 200: Password updated successfully.
  - 400: Incorrect data format.
  - 401: Unauthorized, token might be invalid or expired.
- **Example**:
  - Request:

```
{
    "oldpassword": "password"
    "newpassword": "newpassword"
}
```

  - Response:

```
{
    "id": 1,
    "username": "Alice",
    "email": "Alice@gmail.com"
    "newpassword":"newpassword (SHA encrypted)"
}
```

4. Edit Profile

- **URL**: (`api/user/edit-profile`)
- **Method**: PUT
- **Headers**:
  - Content-Type: application/json
  - Authorization: Your JWT access token.
- **Parameters**:
  - **username**: String, the new name for the user.
  - **email**: String, the new email address for the user.
- **Response**:

- **200**: Profile updated successfully, returns the updated user details.
- **400**: Incorrect data format, and validation errors.

- **Example**:
  - **Request**:

```
{
    "username": "Alice Johnson",
    "email": "alice.johnson@gmail.com"
}
```

  - **Response**:

```
{
    "username": "Alice Johnson",
    "email": "alice.johnson@gmail.com"
}
```

5. Refresh Token

- **URL**: `api/user/login/refresh`
- **Method**: `POST`
- **Parameters**:
  - `refresh`: Your refresh token.
- **Response**:
  - 200: New token provided.
  - 401: Invalid or expired refresh token.
- **Example**:
  - Request:

```
{
    "refresh": "string"
}
```

  - Response:

```
{
    "refresh": "string",
    "access": "string"
}
```

6. UserRecordInfo

- **URL**: `api/user/data/records/`
- **Method**: `GET`
- **Parameters**:
  - `username: string`
- **Response**:
  - 200: New token provided.
  - 401: Invalid or expired refresh token.
- **Example**:
  - Request:

```
{
    "username": "string"
}
```

- Response:

```
{
    "username": string,
    "info": [] // can contain null i.e. no recordings matched
}
```

CLARA Intermediary

1. ~~Filter Users (To be used by CLARA to filter users with specified language)~~

- **~~URL:~~** `api/clara/users`
- **~~Method:~~** `GET`
- **~~Parameters:~~**
  - `language: string`
- **~~Response:~~**
  - ~~200: Task successfully submitted and updated in database.~~
- **~~Example:~~**
  - ~~Request:~~
  - ```
    {
        "language": "English"
    }
    ```
  - ~~Response:~~

```
{
  "users": ["MannyCLARA", "Bill123"]
}
```

1. ~~Add Batch Job (To be used by CLARA to add tasks)~~

- **~~URL:~~** `api/clara/add-batch-job`
- **~~Method:~~** `POST`
- **~~Parameters:~~**
  - `username: string`
  - `task_id: string`
  - `data: JSON`
- **~~Response:~~**
  - ~~200: Task successfully submitted and updated in database.~~
  - ~~400: Bad request; missing required parameters.~~
- **~~Example:~~**
  - ~~Request:~~
  - ```
    {
        "username": "manny1"
        "task_id": "alice_in_wonderland"
        "data": [
          {"text": "Hello  my name is Alice",
    ```

```
        "file": "alice_in_wonderland_1"}
    ]
}
```

- ~~Response:~~

```

```

5. AddTask

- **URL**: `api/task/`
- **Method**: `PUT`
- **Parameters**:
  - `request: JSON`
    - task_id: string
    - user: string
    - tag_id: int
    - data [(text: string, file: string) *pairs*]
- **Response**:
  - 200: Task added.
- **Example**:
  - Request:

```
request = {
    "task_id": "Alice in Wonderland",
    "user": "user_example",
    "tag_id": 123,
    "data": ["Once upon a time there lived a girl called Alice",
             "She lives in a Wonderland","where she falls into a rabbit hole.",
    ],
    "upload_time": N/A
}
```

  - Response:

```
empty
```

6. ChangeUserTask

- **URL: api/task/user**
- **Method**: `POST`
- **Parameters**:
  - `request: JSON`
    - task_id: string
    - user: string
- **Response**:
  - 200: Task changed.
  - 404: Task not found.
- **Example**:
  - Request:

```
request = {
    "task_id": "Alice in Wonderland",
```

```
    "user": "user_example",
}
```

- Response:

```
empty
```

Status Codes

Understanding the HTTP status codes can be crucial in diagnosing issues and understanding the API's behavior. Below are the status codes that our API may return:

- **200 OK**: The request was successful.
- **201 Created**: The resource was successfully created.
- **400 Bad Request**: The server could not understand the request, possibly due to incorrect data format or missing fields.
- **401 Unauthorized**: The user is not authenticated, which might be due to missing or invalid authentication credentials.
- **403 Forbidden**: The user is authenticated but does not have the necessary permissions to access the requested resource.
- **404 Not Found**: The requested resource could not be found. This occurs when requesting a non-existent endpoint or resource.
- **409 Conflict**: The request could not be completed due to a conflict, such as a duplicate resource.
- **500 Internal Server Error**: The server encountered an unexpected error that prevented it from fulfilling the request, indicating an issue with the server itself.

Error Handling

Our API is designed to return specific HTTP status codes and messages to help identify and resolve issues efficiently. Below we detail various error scenarios and how they are handled:

1. Authentication Errors

- **HTTP Status Code:** 401 (Unauthorized)
- **Scenario:** Occurs when a request lacks valid authentication credentials
- **Description:** If a request to a protected endpoint does not include a valid JWT token, a 401 Unauthorized response is returned, indicating the need for valid authentication credentials.

2. Permission Errors

- **HTTP Status Code:** 403 (Forbidden)
- **Scenario:** Occurs when the authenticated user does not have permission to access the requested resource
- **Description:** If an authenticated user tries to access a resource they don't have permission to access, a 403 Forbidden response is returned, signaling insufficient permissions.

3. Not Found Errors

- **HTTP Status Code:** 404 (Not Found)
- **Scenario:** Occurs when the requested resource cannot be found
- **Description:** If a request is made to a non-existent endpoint or to access a resource that doesn't exist, a 404 Not Found response is returned, highlighting the unavailability of the requested resource.

4. Method Not Allowed Errors

- **HTTP Status Code:** 405 (Method Not Allowed)
- **Scenario:** Occurs when the HTTP method used in the request is not supported by the endpoint

- **Description:** If a request uses an HTTP method that is not allowed for the endpoint (e.g., using POST on an endpoint that only supports GET), a 405 Method Not Allowed response is returned, signifying an incorrect HTTP method usage.

5. Server Errors

- **HTTP Status Code:** 500 (Internal Server Error)
- **Scenario:** Occurs when the server encounters an error that prevents it from fulfilling the request
- **Description:** If the server encounters an unexpected error that prevents it from processing the request, a 500 Internal Server Error response is returned, indicating a server malfunction.

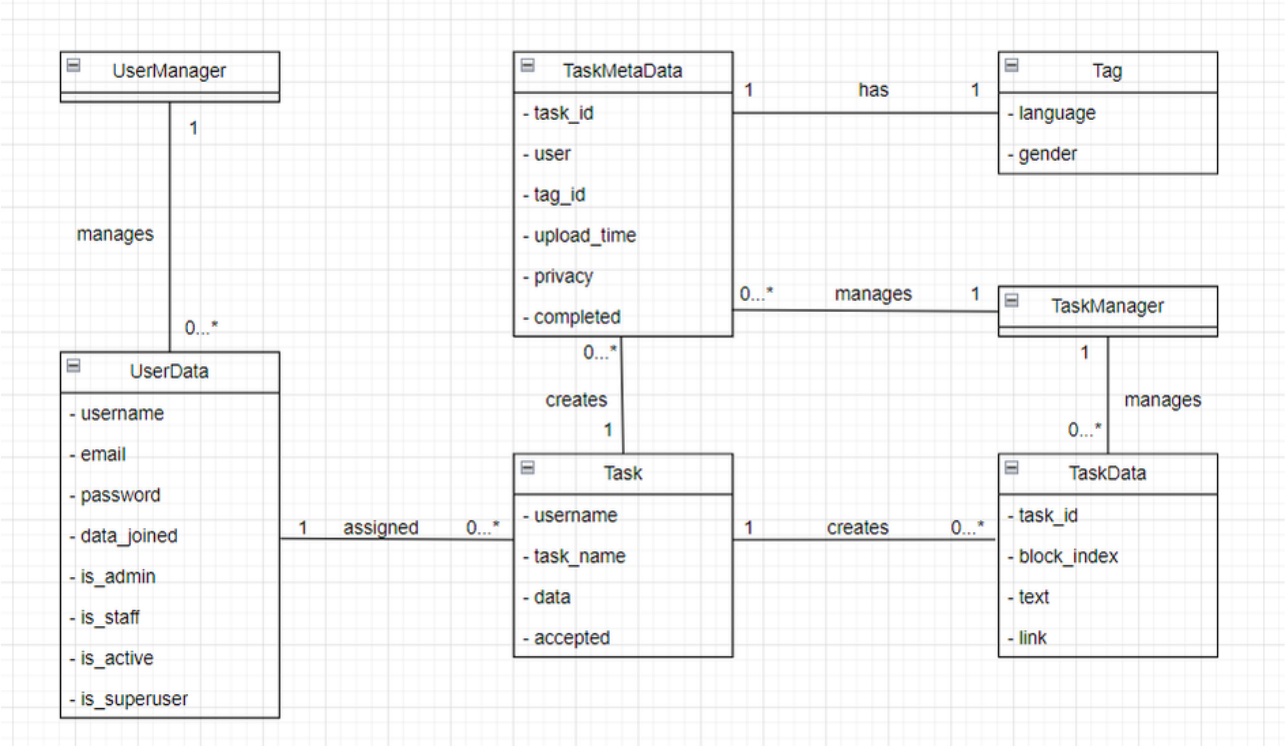
Project Structure

Folder Structure

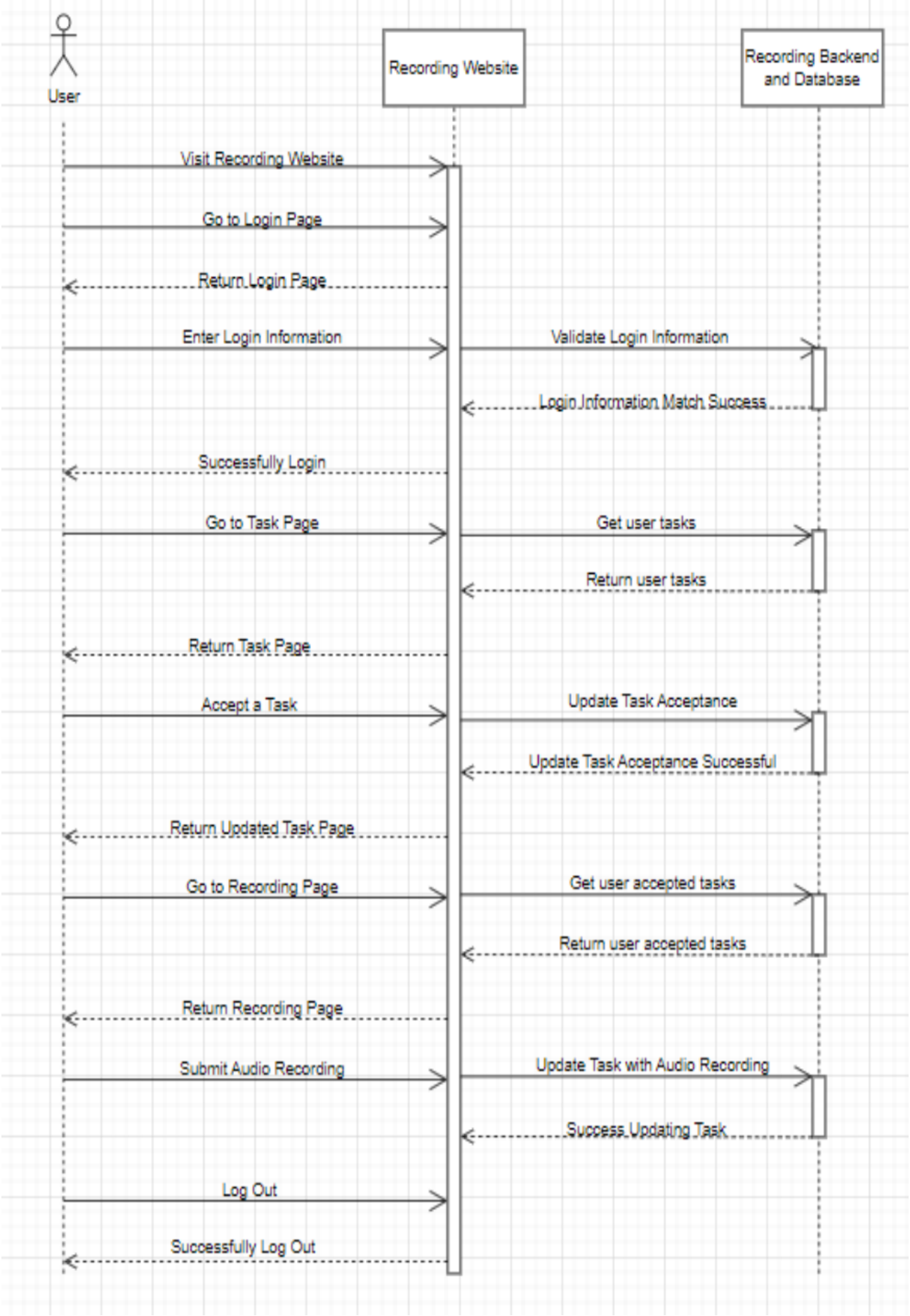
```
djangoProject1/                  # Root directory of the project
├── djangoProject1/              # Contains default Django files
│   ├── __init__.py              # Initialization file
│   ├── asgi.py                  # ASGI configuration
│   ├── settings.py              # Project settings
│   ├── urls.py                  # Project URL mappings
│   └── wsgi.py                  # WSGI configuration
│
└── user/                        # Newly created app for User
    ├── migrations/              # Data migration files
    │   └── __init__.py          # Initialization file
    ├── __init__.py              # Initialization file
    ├── admin.py                 # Admin interface configuration
    ├── apps.py                  # App configuration
    ├── models.py                # Data models
    ├── serializers.py           # Data serialization
    ├── tests.py                 # Test files
    ├── urls.py                  # App's URL mappings
    └── views.py                 # View functions or classes
│
└── data/                        # Newly created app for Data
    ├── migrations/              # Data migration files
    │   └── __init__.py          # Initialization file
    ├── __init__.py              # Initialization file
    ├── models.py                # Data models
    ├── serializers.py           # Data serialization
    ├── tests.py                 # Test files
    ├── urls.py                  # App's URL mappings
    └── views.py                 # View functions or classes
└── claraintermediary/           # Newly created app for Intermediary
    ├── migrations/              # Data migration files
    │   └── __init__.py          # Initialization file
    ├── __init__.py              # Initialization file
    ├── urls.py                  # App's URL mappings
    └── views.py                 # View functions or classes


```

Domain Class Diagram

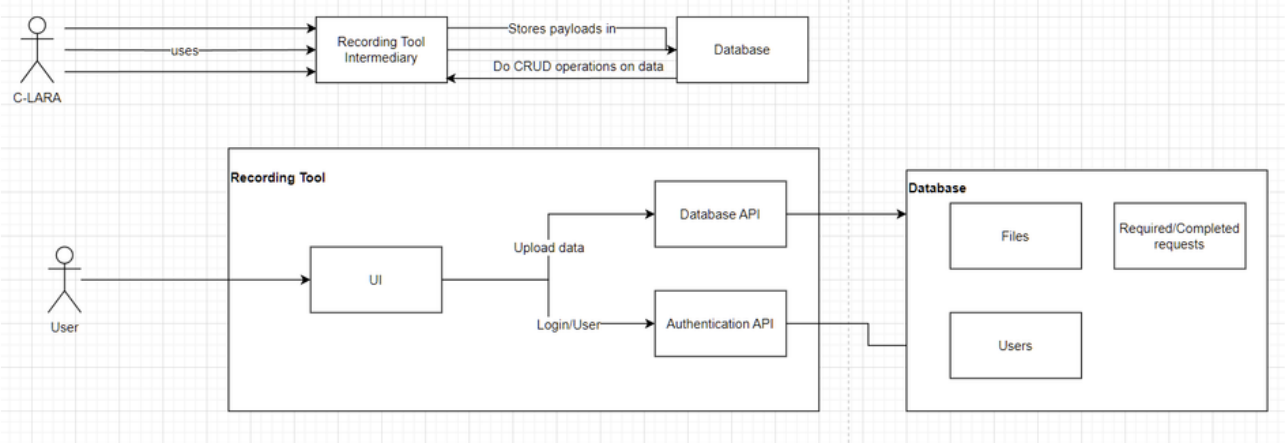


Sequence Diagram



Project Structure for User

UML Diagram for C-LARA and User Interactions



Views

Our Django application leverages a variety of views to facilitate different functionalities. These views are designed using Django's class-based views framework. Here, we outline each view along with its designated endpoint and functionality:

1. RegisterView

- **Endpoint:** `/signup`
- **Method:** POST
- **Description:** Allows new users to register by providing the necessary details such as username, email, and password.

2. EditProfileView

- **Endpoint:** `/edit-profile`
- **Method:** PUT
- **Description:** Enables authenticated users to update their profile details including username and email.

3. ChangePasswordView

- **Endpoint:** `/change-password`
- **Method:** PUT
- **Description:** Facilitates password changes for authenticated users, requiring them to provide both current and new passwords.

Serializers

Our Django application utilizes serializers to validate data and convert complex data types to native Python data types that can be rendered into JSON. Below, we highlight the serializers employed in the application along with their associated fields and methods:

1. UserSerializer

- **Model:** UserData
- **Fields:**
  - `id`: The user's ID (automatically generated by Django)
  - `email`: The user's unique email address
  - `username`: The user's unique username
  - `password`: The user's password
- **Methods:**
  - `validate_email(value)`: Ensures the email is unique
  - `validate_username(value)`: Ensures the username is unique

- `create(validated_data)`: Creates a new user with the validated data

2. EditProfileSerializer

- **Model:** UserData
- **Fields:**
  - `name`: The user's name (Please note: this field appears to be missing in your UserData model; consider updating your model or serializer accordingly)
  - `email`: The user's email address
- **Methods:**
  - `update(instance, validated_data)`: Updates the user profile with the validated data

3. ChangePasswordSerializer

- **Fields:**
  - `current_password`: The user's current password (CharField, write-only, required)
  - `new_password`: The user's new password (CharField, write-only, required)
- **Methods:**
  - `validate_current_password(value)`: Validates the correctness of the current password
  - `validate_new_password(value)`: Validates the new password using Django's password validation rules

Authentication

In our application, we have adopted JSON Web Token (JWT) authentication, a compact and URL-safe means to represent claims between two parties. In this section, we detail how JWT authentication has been implemented in our Django application:

1. JWT Authentication Setup

- **Library:** Django Rest Framework SimpleJWT
- **Views:**
  - `TokenObtainPairView`: To obtain JWT tokens
  - `TokenRefreshView`: To refresh JWT tokens
  - `TokenVerifyView`: To verify JWT tokens

2. Obtaining JWT Tokens

- **Endpoint:** `/login`
- **Method:** POST
- **Parameters:**
  - `username`: The user's username
  - `password`: The user's password
- **Response:** Access and refresh JWT tokens (provided the credentials are correct)
- **Description:** Allows users to obtain JWT tokens by submitting their username and password

3. Refreshing JWT Access Tokens

- **Endpoint:** `/login/refresh`
- **Method:** POST
- **Parameters:**
  - `refresh`: The refresh JWT token
- **Response:** A new access JWT token
- **Description:** Users can acquire a new access token using the refresh token

4. Verifying JWT Access Tokens

- **Endpoint:** `/verify`
- **Method:** POST
- **Parameters:**
  - `token`: The JWT token to be verified
- **Response:** Success message (if the token is valid)
- **Description:** This endpoint allows users to verify the authenticity of their JWT tokens

5. Protected Endpoints

- **Permissions:** `IsAuthenticated` (ensures that only authenticated users can access certain endpoints)
- **Description:** Certain endpoints in the application are protected, requiring users to authenticate to gain access

6. Sending JWT Tokens in Requests

- **Headers:** `Authorization` (to include the JWT token in requests)
- **Description:** To access protected endpoints, users must include their JWT token in the Authorization header of their requests, formatted as:

```
Authorization: Bearer <access_token>
```

Project Structure for Data

Views

Our Django application leverages a variety of views to facilitate different functionalities. These views are designed using Django's class-based views framework. Here, we outline each view along with its designated endpoint and functionality:

1. Get User Tasks

- **URL**: `api/task/user`
- **Method**: `GET`
- **Response**:
  ```
  [
    {
      "task_id": "",
      "block_id": "",
      "text": "",
      "file": "",
      "has_existing": boolean,
    },
    ...
  ]
  ```
- **Example**:
  - Request:

  ```
  empty
  ```

  - Response:

  ```
  [
      {
          "task_id": "Alice somehow in Wonderland",
  ```

```
        "block_id": 1,
        "text": "We went into the Wonderland somehow",
        "file": "Alice somehow in Wonderland 1",
        "has_existing": false
    },
    {
        "task_id": "Alice somehow in Wonderland",
        "block_id": 2,
        "text": "She lives in a magical lovely Wonderland",
        "file": "Alice somehow in Wonderland 2",
        "has_existing": false
    },
    {
        "task_id": "Alice somehow in Wonderland",
        "block_id": 3,
        "text": "where she falls into a big big rabbit hole.",
        "file": "Alice somehow in Wonderland 3",
        "has_existing": false
    },
    {
        "task_id": "Alice not in Wonderland",
        "block_id": 2,
        "text": "She lives in a Wonderland",
        "file": "Alice not in Wonderland 2",
        "has_existing": false
    },
    {
        "task_id": "Alice not in Wonderland",
        "block_id": 3,
        "text": "where she falls into a rabbit hole.",
        "file": "Alice not in Wonderland 3",
        "has_existing": false
    },
    {
        "task_id": "Alice not in Wonderland",
        "block_id": 1,
        "text": "Once upon a time there lived a girl called Alice",
        "file": "Alice not in Wonderland 1",
        "has_existing": false
    }
]
```


2. Submit Task

- **URL**: `api/task/submit/{task_id}/{block_id}`
- **Method**: `POST`
- Parameters:
  - binary: *binary data*
- **Response**:
  - 200: Task successfully submitted and updated in database.
  - 404: `Task_id/Block_id is none.`
  - 400: `Attempted to submit empty/null file.`
- **Example**:
  - Request:


```
010101010010100...
```

- Response:

```
empty
```

3. Get Audio

- **URL**: `api/audio/{task_id}/{block_id}`
- **Method**: `GET`
- **Path parameter**:
  - `task_id: string`
  - `block_id: int`
- **Response**:
  - 200: Audio file retrieved successfully.
  - 404: `Audio file not found`
- **Example**:
  - Request:

```
empty
```

  - Response:

```
010
```

4. Delete Task

- **URL**: `api/task/{task_id}`
- **Method**: `DELETE`
- **Parameters**:
  - `task_id: string`
- **Response**:
  - 200: Task and related files and data deleted successfully.
  - 400: `Missing task_id value.`
- **Example**:
  - Request:

```
empty
```

  - Response:

```
empty
```

5. Clear Task

- **URL**: `api/task/clear/{task_id}/{block_id}`
- **Method**: `POST`
- **Parameters**:
  - `task_id: string`
  - `block_id: int`
- **Response**:
  - 200: Task's block audio file deleted/cleared.
  - 400: `Task_id does not exist.`

- **Example**:
  - Request:

```
empty
```

  - Response:

```
empty
```

Serializers

Our Django application utilizes serializers to validate data and convert complex data types to native Python data types that can be rendered into JSON. Below, we highlight the serializers employed in the application along with their associated fields and methods:

1. `NewMetaDataAudioSerializer`

- **Model:** TaskMetaData
- **Fields:**
  - `task_id`: The unique task id.
  - `user`: The username
  - `tag_id`: Tags include language, gender, other attributes of the requested task model
  - `upload_time`: The time of upload
- **Methods:**
  - `validate_task_id(value)`: Ensures the task_id is unique
  - `validate_username(value)`: Ensures the username is unique

2. `NewDataAudioSerializer`

- **Model:** TaskData
- **Fields:**
  - `block_id`: The block id is used to organise a task's set of text which allows
  - `text`: The user's email address
  - `file`: The proposed pathname

3. `TaskUserSerializer`

- **Model:** TaskMetaData
- **Fields:**
  - `task_id`: The task id
  - `user`: The username

Database Entity-Relation Diagram