Person API Documentation

Creating comprehensive API documentation is essential for users to understand and utilize your API effectively. Below is an example of what my `DOCUMENTATION.md` file might look like for the Django REST API:

This document provides information on how to use the Person API, which allows for CRUD (Create, Read, Update, and Delete) operations on person records.

Description

A CRUD API (Create, Read, Update, and Delete) is a type of API that allows users to perform basic operations on data. These operations are typically performed on resources, which can be anything from simple data objects to complex entities. CRUD APIs are used in a variety of applications, including web applications, mobile apps, and desktop applications. They are also used by developers to build other APIs and services. The purpose of a CRUD API is to provide a simple and efficient way to manage data. By using a CRUD API, developers can avoid having to write their own code to perform basic operations on data. This can save time and effort, and it can also help to Here are some examples of how CRUD APIs can be used: A web application might use a CRUD API to allow users to create, read, update, and delete their accounts.

- A mobile app might use a CRUD API to allow users to add, remove, and edit items in their shopping cart.
- A desktop application might use a CRUD API to allow users to manage their contacts or appointments. CRUD APIs are a powerful tool that can be used to simplify the development and management of applications that need to interact with data.

Prerequisites

List of prerequisites or dependencies that users need to have installed before setting up and running your API. For example:

- Python (version 3.7 or higher)
- Django (Django==3.2.21)
- Django REST framework (djangorestframework==3.14.0)

Create Operation (POST):

To handle the creation of a new person,

Read Operation (GET):

To handle the list of a new person,

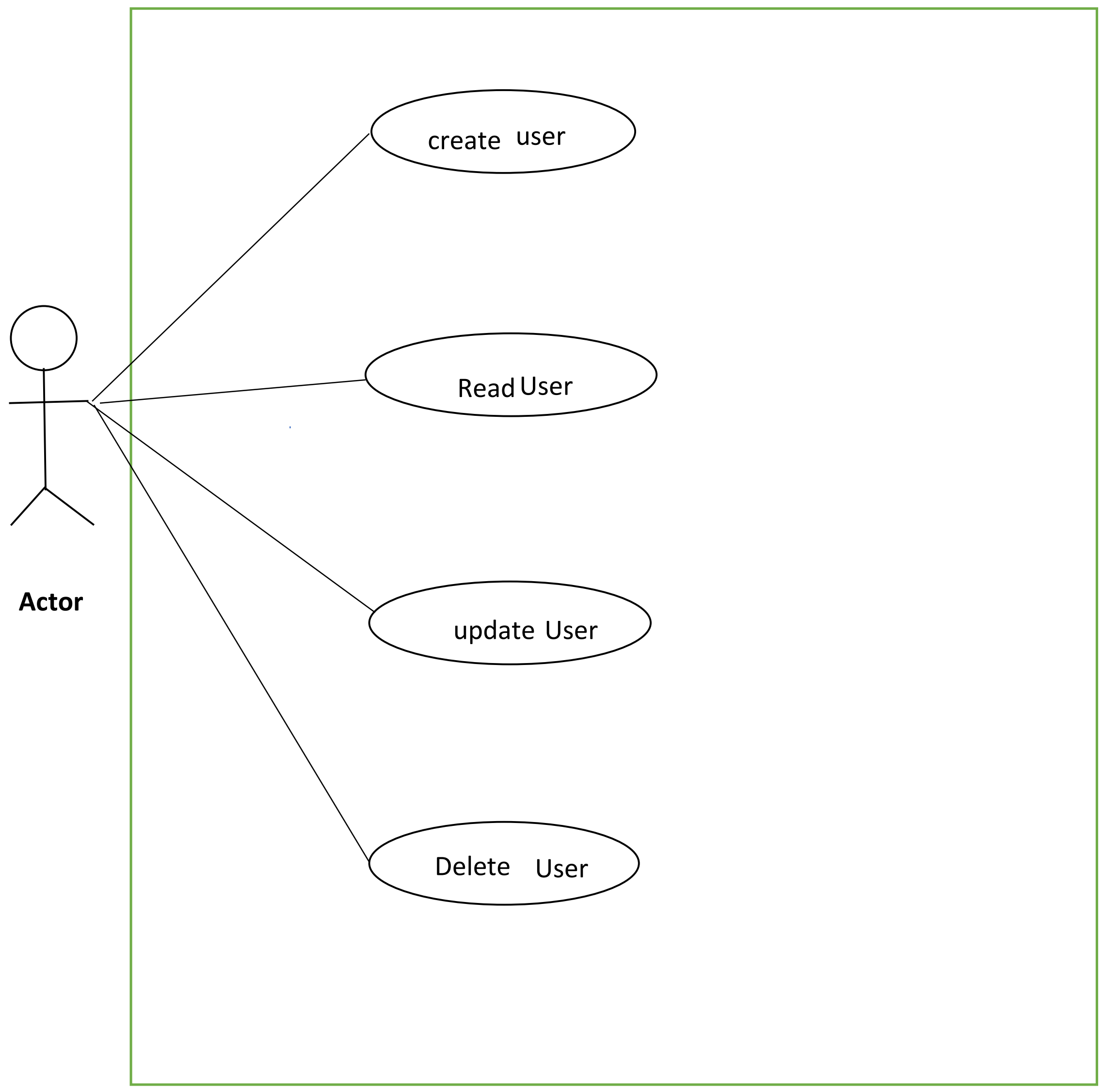
Update Operation (PUT/PATCH):

To handle updating a person's information

Delete Operation (DELETE):

To handle the deletion of a person,

USE -CASE DIAGRAM



API Base URL

The base URL for all API endpoints is `/api/`.

Endpoints

1. Create a New Person

Endpoint: `/api/`

Request Type: POST

Request Format:

```json
    {
      "name": "John Doe",

    }
```

Response Format:

```json
    {
      "id": 1,
        "name": "John Doe",
    }
```

2. Fetch Details of a Person

Endpoint:`/api/<user_id>/`

Request Type: GET

Response Format:

```json
    {
        "id": 1,
        "name": "John Doe",
    }
```

3. Modify Details of an Existing Person

Endpoint: `/api/<user_id>/`

Request Type: PUT or PATCH

Request Format:

```json
        {
            "name": "Updated Name",
            // Add other fields to update as needed
        }
```

Response Format:

```json
        {
            "id": 1,
            "name": "Updated Name",

        }
```

4. Remove a Person

Endpoint: `/api/<user_id>/`

Request Type: DELETE

Response Format:

```json
        {
           "message": "Person with id 1 has been deleted successfully."
        }
```

Sample Usage

Create a New Person

Request:

```http
        POST /api/ HTTP/1.1
        Host: https://hngxstagetwo-2ddm.onrender.com/api/
        Content-Type: application/json

        {
```

```
        "name": "Alice Smith",


    }
```

Response:

```json
    {
        "id": 2,
        "name": "Alice Smith",


    }
```

Fetch Details of a Person

Request:

```http
GET /api/2/ HTTP/1.1
Host: your-api-host.com
```

Response:

```json
        {
            "id": 2,
            "name": "Alice Smith",

        }
```

Modify Details of an Existing Person

Request:

```http
PUT /api/2/ HTTP/1.1
Host: your-api-host.com
Content-Type: application/json

        {
            "name": "Updated Name",
        }
```

Response:

```json
    {
       "id": 2,
       "name": "Updated Name",
    }
```

Remove a Person

Request:

```http
DELETE /api/2/ HTTP/1.1
Host: your-api-host.com
```

Response:

```json
```

```
    {
       "message": "Person with id 2 has been deleted successfully."
    }
```

Limitations and Assumptions

- This API assumes that you are working with a single "Person" resource and does not include authentication or authorization mechanisms. It's meant for educational purposes and should be extended for production use.

- The API is not deployed on a specific server in this documentation. You'll need to deploy it according to your hosting environment.

Local Setup and Deployment

To set up and deploy the API locally or on a server, follow these general steps:

1. Clone the project repository from GitHub.
2. Install the required dependencies using `pip install -r requirements.txt`.
3. Configure the database settings in `myproject/settings.py` to connect to your chosen database.

4. Apply database migrations using `python manage.py makemigrations` and `python manage.py migrate`.

5. Run the development server with `python manage.py runserver`.

Certainly! Here's a more detailed explanation of the local setup for your Django REST API:

1. Clone the Project Repository:

Assuming you have your Django project code hosted on a version control system like Git (e.g., GitHub), you'll want to clone the project repository to your local machine. Use the following command to clone the repository:

```bash
    git clone https://github.com/Abubakarauta/hngxStageTwo
```

2. Create a Virtual Environment:

It's a good practice to work within a virtual environment to isolate your project's dependencies. You can use `venv` to create a virtual environment:

```bash
    python -m venv myenv
```

Activate the virtual environment:

- On Windows:

  ```bash
        myenv\Scripts\activate
  ```

- On macOS and Linux:

  ```bash
         source myenv/bin/activate
  ```

3. Install Dependencies:

Navigate to your project's directory and install the required dependencies listed in the `requirements.txt` file:

```bash
      cd yourproject
```

```
    pip install -r requirements.txt
```

4. Configure Database:

Open the `myproject/settings.py` file and configure your database settings. By default, Django uses SQLite for development, which is simple to set up. You can change the database engine and connection details as needed.

Here's an example using SQLite:

```python
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': BASE_DIR / 'db.sqlite3',
    }
}
```

5. Apply Database Migrations:

Run the following commands to apply database migrations and create the database schema:

```bash
python manage.py makemigrations
python manage.py migrate
```

6. Run the Development Server:

Start the Django development server to run your API:

```bash
python manage.py runserver
```

By default, the server will run on `http://127.0.0.1:8000/`. You can access your API endpoints at this URL.

7. Test Your API:

You can now use tools like `curl`, `httpie`, or a web browser to make requests to your API endpoints. For example, to create a new person, you can use `httpie` as follows:

```bash
http POST http://127.0.0.1:8000/api/ name="John Doe" age:=30 email="johndoe@example.com"
```

8. Develop and Extend:

Now that your local setup is running, you can develop and extend your API by creating new views, serializers, and models in your Django app (`myapp` in this case) as needed. Refer to your Django project's structure and the documentation for more details on building out your API.

Remember that this local setup is for development purposes. For production deployment, you will need to follow a different set of steps, including configuring a production-ready web server, setting up a production database, and ensuring security measures are in place.

For deployment, consider using popular hosting platforms like Heroku, AWS, or a dedicated server, and follow their deployment guidelines.

Remember to secure your API and database for production use, including setting up authentication, authorization, and HTTPS if needed.