# README.md

## Overview

This project is a URL shortening service that allows users to shorten long URLs and redirect from the shortened URL to the original URL. It includes features for creating shortened URLs, redirecting to the original URLs, and tracking the number of views for each short URL. Additionally, it supports specifying an expiration period for the short URLs.

## Essential Features

1. **Create Shortened URL**
   - **Endpoint**: `POST /shorten`
   - **Request Body**: `{"url": "<original_url>"}`
   - **Response Body**: `{"short_url": "<shortened_url>"}`
   - Converts the received long URL into a unique short key and stores it in the database.
2. **Redirect to Original URL**
   - **Endpoint**: `GET /<short_key>`
   - **Response**:
     - If the key exists, redirects to the original URL with a 301 status code.
     - If the key doesn't exist, returns an error message with a 404 status code.

## Additional Features

1. **URL Key Expiration**
   - Allows specifying an expiration period when creating a key and deletes expired keys.
   - **Endpoint**: `POST /shorten`
   - **Request Body**: `{"url": "<original_url>", "expiration": <seconds>}`

2. **Statistics**
    - Tracks the number of views for each short key and provides a statistics endpoint to return this information.
    - **Endpoint**: `GET /stats/<short_key>`
    - **Response Body**: `{"views": <view_count>}`
3. **Test Code**
    - Includes unit tests and integration tests to ensure the functionality of the service.

## Technology stack

1.**FastAPI** ： The web framework used for building the API.

2.**SQLAlchemy** : Python SQLite database interactions(ORM)

3.**SQLite**: : SQLite is a serverless database, extremely easy to set up and use, especially for small-scale applications or development and testing environments. It has minimal external dependencies, SQLite stores the entire database in a single file on disk, making it lightweight and easy to manage. It uses fewer resources compared to other and For read-heavy operations, SQLite can be very fast. Since our application primarily performs read operations (redirecting short URLs), SQLite's performance is adequate for our needs. SQLite provides full ACID compliance, ensuring that transactions are processed reliably. In short it is perfect for initial development and testing.

Note :

For Scalability considerations, in a production environment or if more users concurrently access, then we might need RDS like PostgreSQL or NoSQL like MongoDB, Amazon Dynamo DB. PostgreSQL has advanced features and versatile. It can handle high number of concurrent connections and  horizontally scalable. Known for its reliability and integrity features. MongoDB/DynamoDB is also known for its flexibility for handling unstructured data and horizontal scalability.It can handle large volumes of data with high throughput.

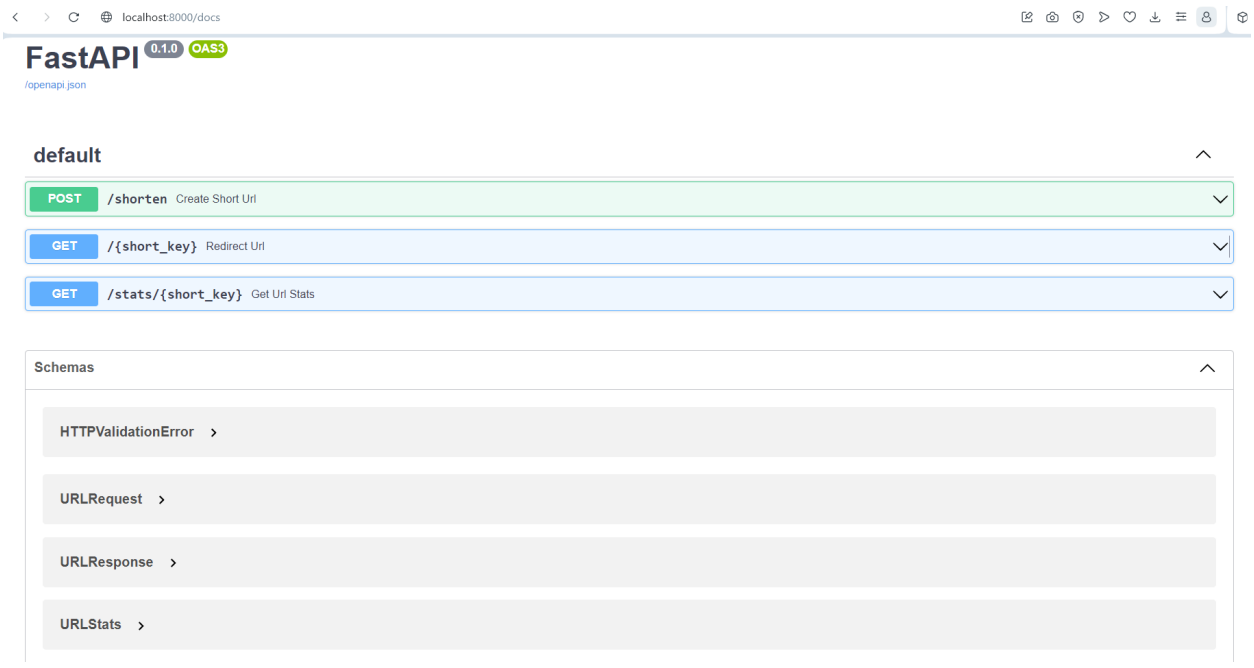
4.**Pydantic** : used by FastAPI, for the data validation and settings management

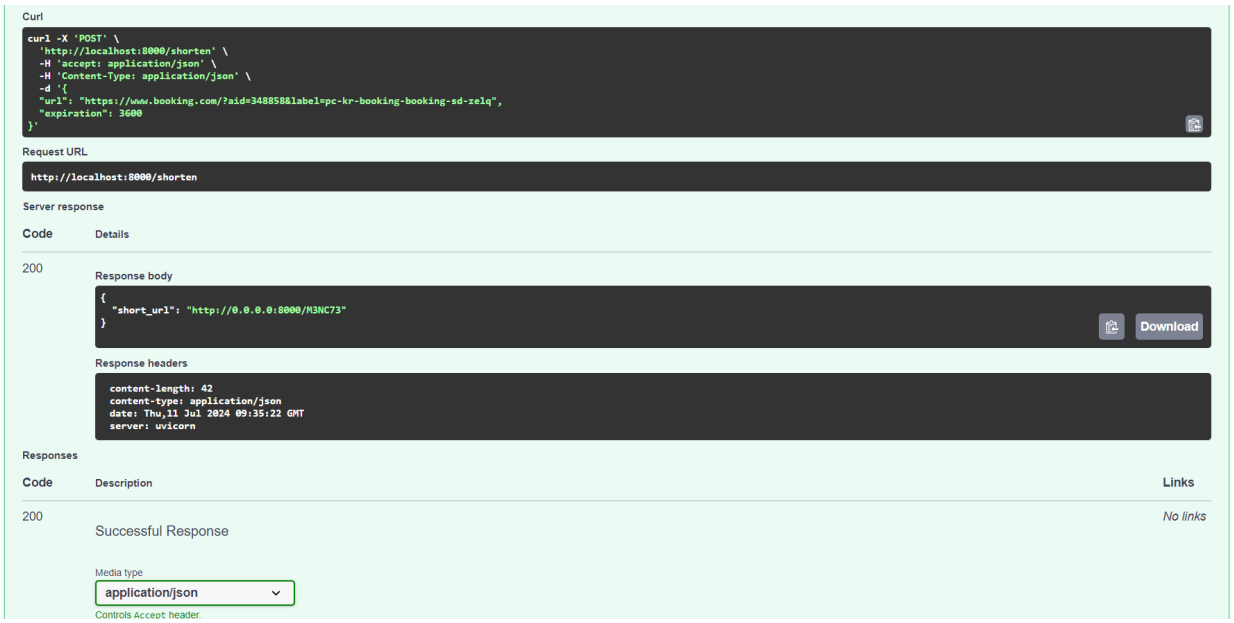
5. **Docker Compose** for development

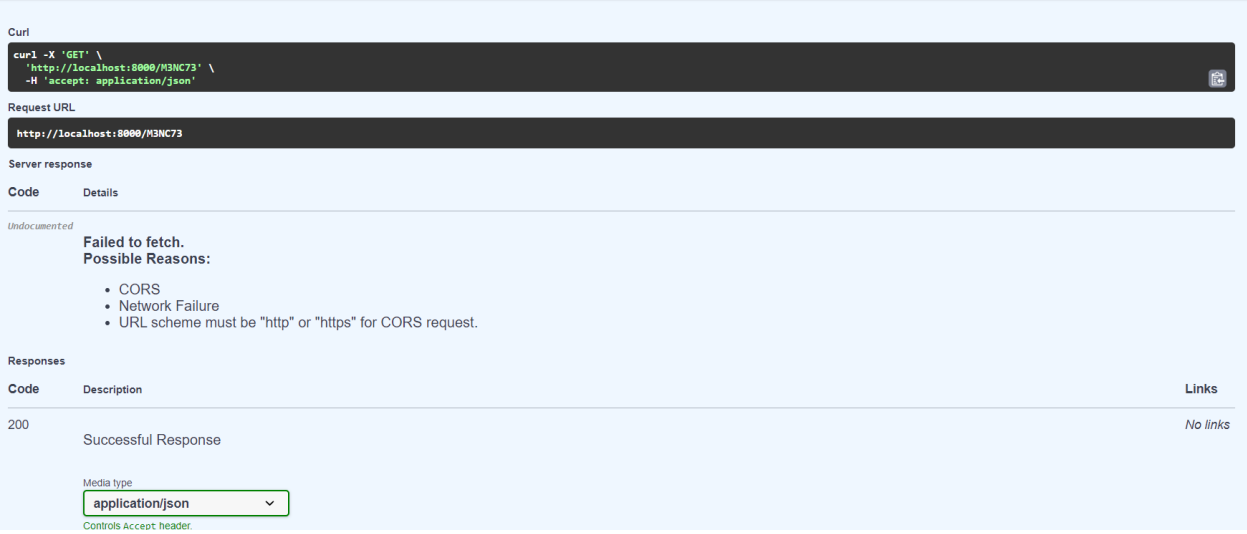
6.Tests with **Pytest**

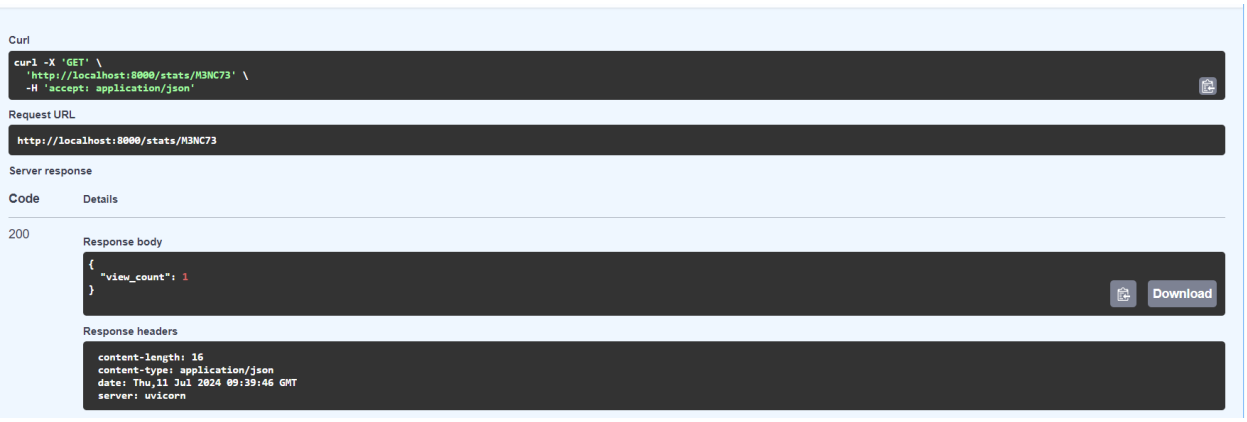
7.**Uvicorn** : An ASGI server for serving the FastAPI app.

## Interactive API Documentation:






## How to use it :

Step 1:Clone the repository and set up project folder

```
git clone https://github.com/yolotalkies123/url_shortner.git
cd url_shortner
```

Step2: Activate your Conda environment in interpreter. Make sure to use VScode or PyCharm IDE

Step3:Install dependencies

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

```
cd app
```

Step4:Run the application in command terminal as,

```
uvicorn app:app --reload --port 8000 --host 0.0.0.0
```

if you prefer docker compose, then

```
docker-compose up --build
```

Step5:The database file (`test.db`) will be created automatically in the project root directory when you run the application. In our case /app/test.db .

Step6:Swagger documentation is available at `http://localhost:8000/docs`.

Step7:Run testing scripts with pytest

```
cd tests
pytest test_app.py
pytest test_crud.py
pytest test_utils.py
```

## Folder Structure:

url_shortner/
├── app/ # Main application directory
│ ├──
**pycache**/ # Python cache directory
│ ├── .pytest_cache/ # Pytest cache directory
│ ├── tests/ # Directory for test cases
│ │ └──
**init**.py # Makes `tests` a package
│ ├──
**init**.py # Makes `app` a package
│ ├── .env # Environment variables file

```
│   ├──
app.py                  # Main FastAPI application file
│   ├──
crud.py                 # CRUD operations for database
│   ├──
database.py             # Database connection and session setup
│   ├──
models.py               # SQLAlchemy models for database tables
│   ├──
schemas.py              # Pydantic schemas for request and response models
│   ├── test.db                 # SQLite database file for testing
│   └──
utils.py                # Utility functions
├── .dockerignore               # Files and directories to ignore in Docker build
├── docker-compose.yml          # Docker Compose configuration
├── Dockerfile                  # Dockerfile for building the Docker image
├──
README.md               # Project documentation
└── requirements.txt            # Python dependencies
```

Note : There are many functions for generating shortkeys like Base62 Encoding, Hash, NanoID, Hashing ,Autoincrement, UUID. It is based on the requirement purely.

## Access the database:

We don't need a separate tool to view the SQLite database, but you can use tools like DB Browser for SQLite if you need a graphical interface to inspect the database contents.

For basic operations, you can use command-line tools or Python scripts to interact with the database.

**Steps :**

Step1: Open the command prompt or Linux terminal

Step2: Navigate to the directory containing database file

Step3:Start SQLite Command-Line Interface

```
sqlite3 test.db
```

Step4:View tables in the database

```
.tables
```

Step5:View Schema in a table

```
.schema urls_hub
```

Step6:Query a table

```
SELECT * FROM urls_hub;
```

Step7:Exit CLI

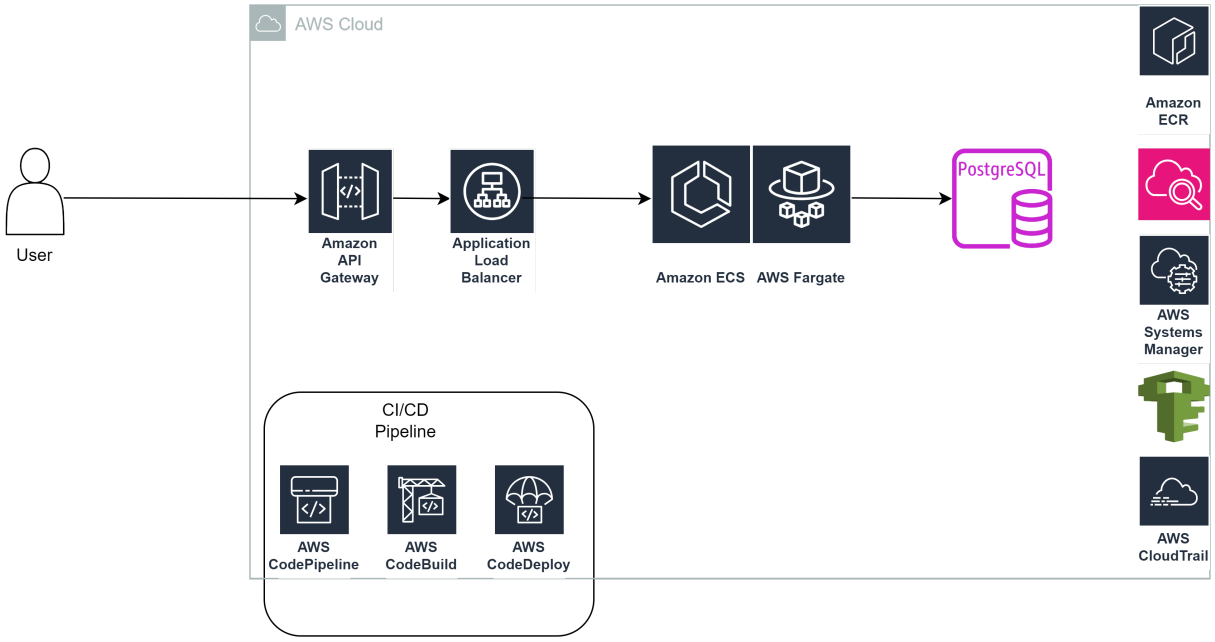
```
.exit
```

## Future Enhancements:

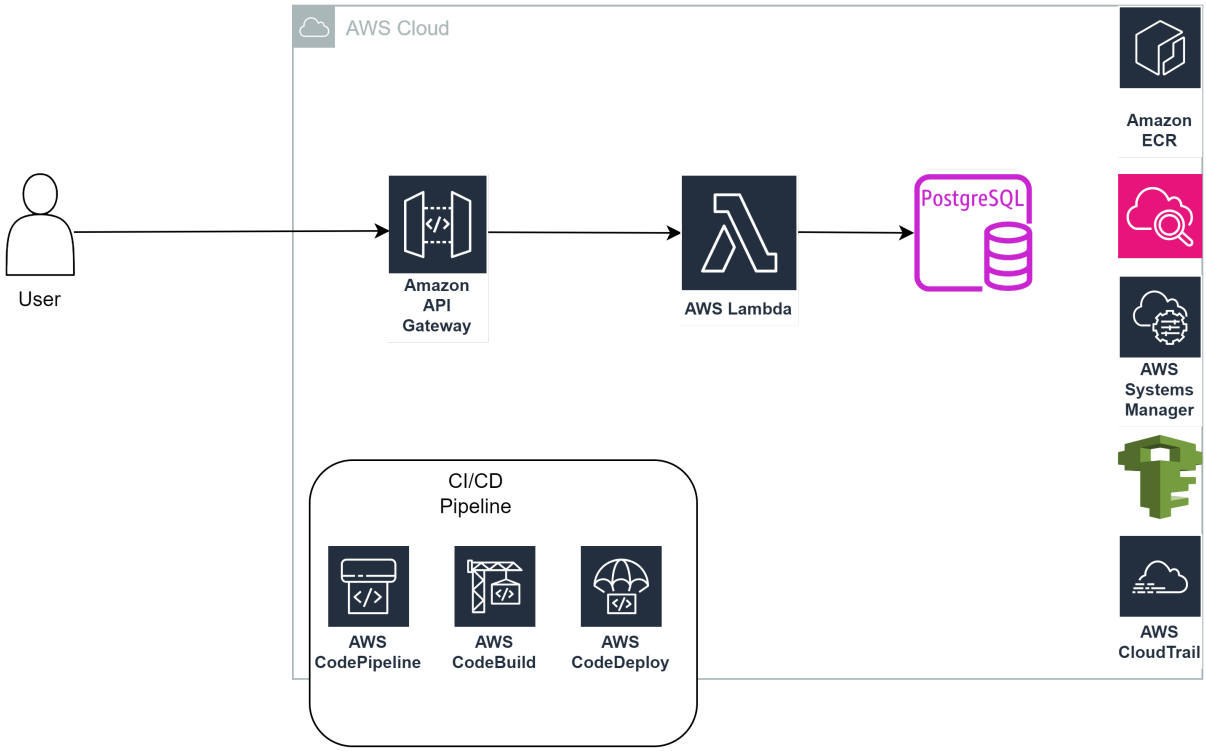
- Implementing support for PostgreSQL or MongoDB for better scalability.
- Enhancing security features, such as rate limiting and authentication.
- Improving the key generation algorithm for better efficiency and uniqueness.
- Implementing in any cloud service like AWS in Cloud native architecture  using ECS/EKS or in serverless fashion using Lambda
- Can add a codepipeline, CI (continuous integration) and CD (continuous deployment) based on GitHub Actions for DevOps methodology.
- In production mode, we need to consider scalability, availability, reliability, Integrity ,monitoring ,performance and cost factors to implement a dynamic web application

## Architecture Diagram/System Design :

Scenario 1: In Cloud Native Architecture




Scenario 2: In Serverless Architecture




Note:When selecting architecture for an application, it's important to balance availability, security, performance, and cost. High availability requires redundant systems across multiple zones to minimize downtime. Security involves using VPC isolation, IAM roles, and encryption. For performance, scalable services like auto-scaling ECS tasks and managed databases handle varying loads efficiently. To manage costs, use pricing models like spot instances and serverless services. Additionally, consider ease of management, monitoring, and integration with existing systems to ensure the solution meets both technical and business needs.