AI RBAC task

Introduction:

The FastAPI AI-RBAC project is a secure API with:

JWT-based authentication

Role-Based Access Control (RBAC)

MongoDB for user data storage

AI-powered question answering

PDF Upload & Querying

This API allows **user registration, login, AI-based question answering, and admin functionalities**.

Project Setup:

🔹 Install Dependencies

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

🔹 Start MongoDB

Ensure MongoDB is running:

```
mongod --dbpath /data/db
```

🔹 Run the Application

```
uvicorn app.main:app --reload
```

`After running, access Swagger UI:`

```
http://127.0.0.1:8000/docs
```

Architecture Overview:

app/
│── ai.py # AI Processing Logic
│── auth.py # Authentication (JWT, Password Hashing)
│── db.py # MongoDB Connection & CRUD
│── main.py # FastAPI Application & Routes
│── models.py # Pydantic Models
│── permissions.py # Role-Based Access Control (RBAC)
│── tests/ # Unit Tests
│── static/ # (Optional) Static files
│── templates/ # (Optional) HTML Templates
.env # Environment Variables
requirements.txt # Dependencies
README.md # Documentation

Database Configuration:

MongoDB is used for user management.

📍 **Database Name**: `fastapi_db`
📍 **Collection**: `users`

Field	Type	Description
_id	String	Unique Object ID
username	String	Unique Username
email	String	User Email
hashed_password	String	Encrypted Password
role	String	User role (`user`, `admin`)

Authentication & Authorization

- JWT Authentication
- OAuth2PasswordBearer (OAuth2 with Password Grant)
- Password Hashing using bcrypt
- Role-Based Access Control (RBAC)

How JWT Works:

1. User logs in → JWT Token generated
2. User makes requests → Sends JWT in `Authorization` header
3. API verifies token → Grants or denies access

API Endpoints

🔹 Authentication & User Management

Method	Endpoint	Description
POST	/register/	Register a new user
POST	/token/	User login (returns JWT)
GET	/users/me/	Get current user info

🔹 Role-Based Access

Method	Endpoint	Description
GET	/admin/	Admin-only access

🔹 AI-Powered Features

Method	Endpoint	Description
GET	/ask/	Ask AI a question
GET	/ai/protected/	AI Feature (Auth required)

🔹 PDF Handling

Method	Endpoint	Description
POST	/upload-pdf/	Upload a PDF file
GET	/ask-pdf/	Ask questions about uploaded PDFs

AI & PDF Handling

AI Processing

The `process_query(query)` function generates AI-based responses.

```
GET /ask/?query=What is AI?
```

Response

```
{
  "question": "What is AI?",
  "answer": "Artificial Intelligence is..."
}
```

PDF Upload & Question Answering

Upload a PDF via `/upload-pdf/`
Ask a question about the PDF using `/ask-pdf/?query=your-question`

Unit Testing

Test Framework: `unittest`

Run Tests:

```
pytest tests/
```

- Test Cases

Test File	Description
test_auth.py	Tests password hashing, JWT generation
test_db.py	Tests user creation, retrieval from MongoDB
test_main.py	Tests API endpoints: Register, Login, User Info
test_permissions.py	Tests Role-Based Access Control (RBAC)
test_ai.py	Tests AI response generation
test_pdf.py	Tests PDF upload and querying

How to Run the Project

- 1. Install Dependencies

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

- 2. Start MongoDB

Ensure MongoDB is running:

```
mongod --dbpath /data/db
```

🔹 3. Set Up Environment Variables

Create `.env` file:

```
SECRET_KEY=your_secret_key_here
MONGO_URI=mongodb://localhost:27017
DATABASE_NAME=fastapi_db
ACCESS_TOKEN_EXPIRE_MINUTES=30
```

🔹 4. Run FastAPI

```
uvicorn app.main:app --reload
```

After running, access **Swagger UI**:

```
http://127.0.0.1:8000/docs
```

🔹 5. Run Tests

Run all tests:

```
pytest tests/
```

API Testing & Execution Report

Step 1: User Registration

Description:

This screenshot represents the **user registration process** via the `/register/` endpoint.

Process:

1. The user submits a **POST request** to `http://127.0.0.1:8000/register/` with the following JSON payload:

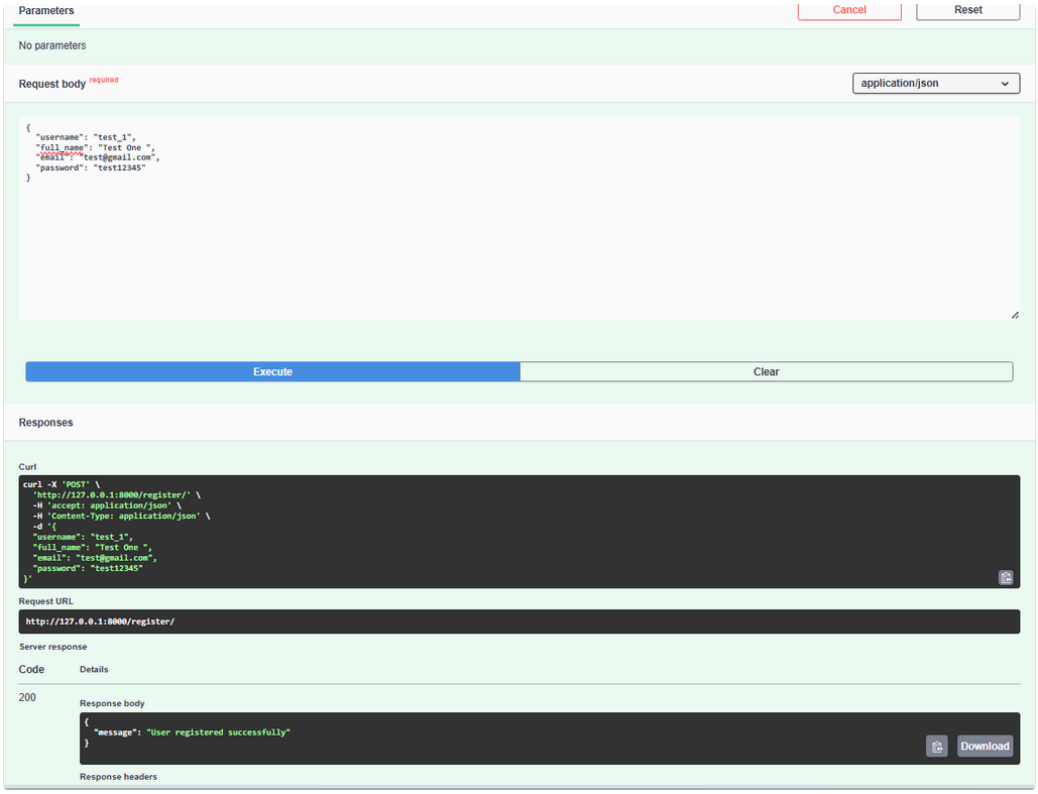
```
{
   "username": "test_1",
   "full_name": "Test One",
   "email": "test@gmail.com",
   "password": "test12345"
}
```

2. The server **processes the request** and registers the user in the database.
3. A **successful response (HTTP 200)** is returned with:

```
{
   "message": "User registered successfully"
}
```

Expected Outcome:

- The new user `test_1` is now stored in the database with a **hashed password**.
- The user can now **log in** using their credentials.



Step 2: User Login (Token Generation)

Description:

This screenshot represents the **user authentication process** via the `/token/` endpoint.

Process

1. The user submits a **POST request** to `http://127.0.0.1:8000/token/` with **form data**

```
grant_type: password
username: test_1
password: test12345
scope: (empty)
client_id: (empty)
client_secret: (empty)
```

2. The system verifies the user's **credentials** by checking:
   - If `test_1` exists in the database.
   - If the **hashed password** matches the stored hash.
3. If valid, the server responds with **HTTP 200 OK** and provides an **access token**:

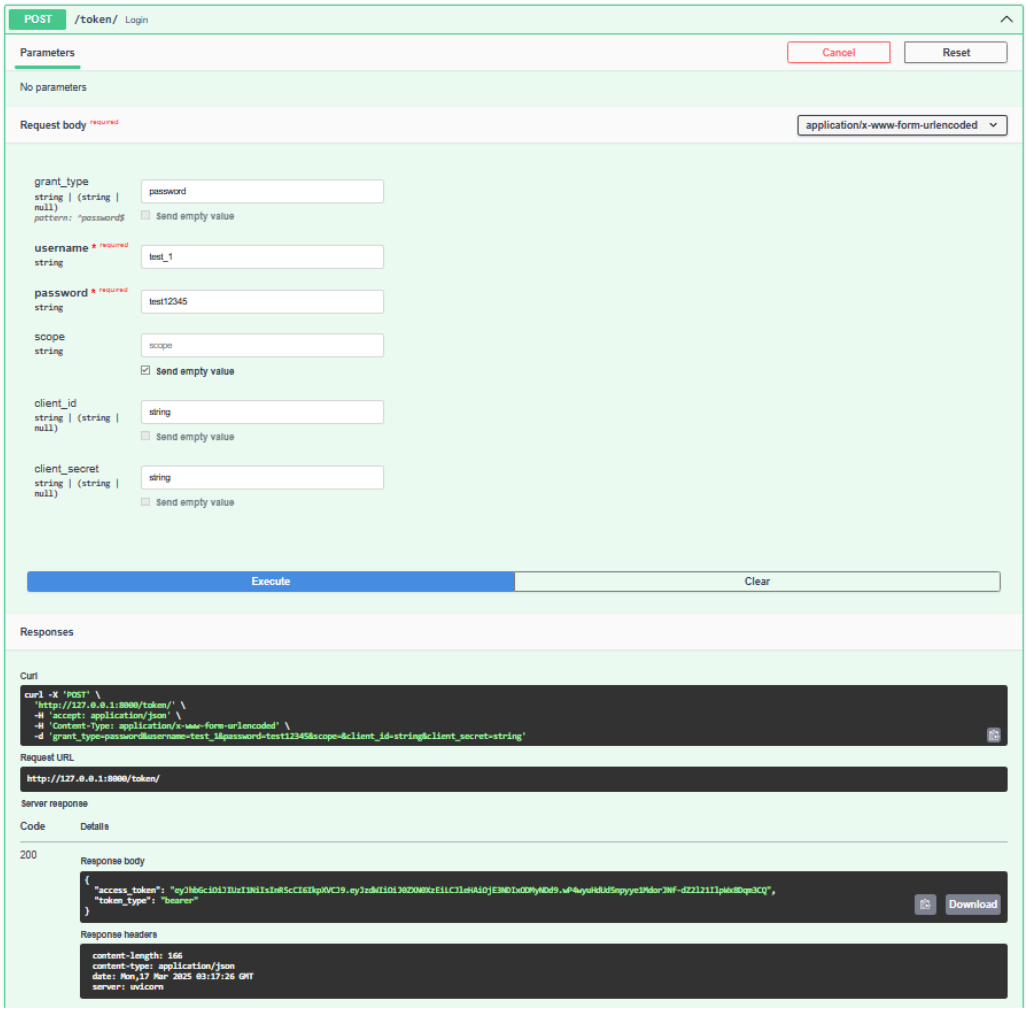
```
{
   "access_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...",
   "token_type": "bearer"
}
```

   This token is used for **authenticated requests**.

Expected Outcome:

- The user **successfully logs in**.

- The generated **JWT token** can be used to access **protected routes**.



Step 3: Authorization (OAuth2 Token Authentication)

Description:

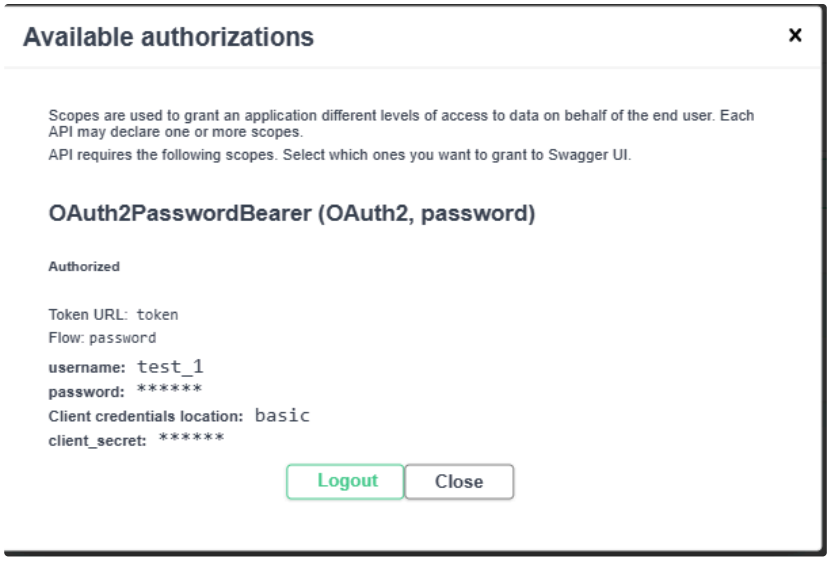
This screenshot shows the **Swagger UI authorization process**, where the user logs in using the **OAuth2 Password Flow**.

Process:

1. The user enters:
   - `username: test_1`
   - `password: ******`
   - **Client credentials location:** `basic`
   - Leaves `client_secret` empty (not needed for password-based authentication).
2. Clicks **Authorize**.
3. Swagger UI stores the **access token** from the `/token/` response and attaches it to **future API requests**.

Expected Outcome:

- The user is now **authenticated** and can **access protected endpoints**.
- The **lock icon** in Swagger UI should turn **green**, indicating successful authentication.



Step 4: Get Current User Info (`GET /users/me/`)

Description:

This screenshot verifies that the authenticated user can **retrieve their own profile information**.

Process:

1. The request is sent to:

```
GET http://127.0.0.1:8000/users/me/
```

2. The request includes an **Authorization** header:

```
Authorization: Bearer <your_access_token>
```

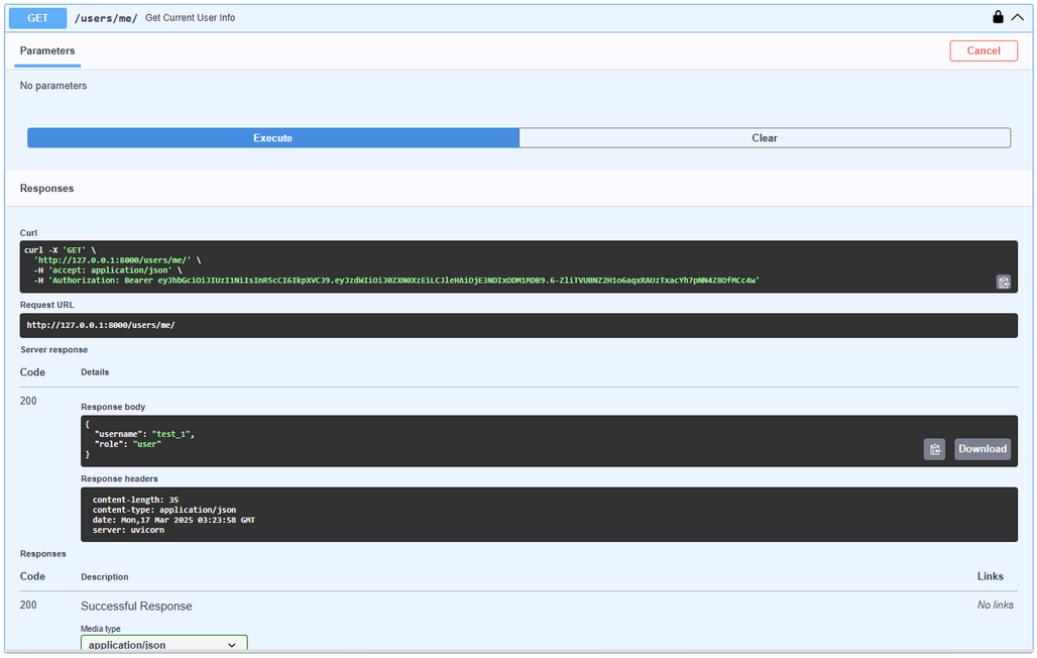
3. The response body confirms:

```
{
   "username": "test_1",
   "role": "user"
}
```

- The **username** is correctly fetched.
- The **role** is `"user"`, meaning the user has standard permissions.

Expected Outcome:

- **Status Code** `200 OK` confirms that authentication and data retrieval work as expected.
- The user can now **access protected routes**.



Step 5: Admin Dashboard Access (`GET /admin/`)

Description:

This screenshot verifies that **admin users can access the admin dashboard**, which is a **protected route requiring admin privileges**.

Process:

1. The request is sent to:

```
GET http://127.0.0.1:8000/admin/
```

2. The request includes an **Authorization** header with a valid **admin token**:

```
Authorization: Bearer <your_admin_access_token>
```

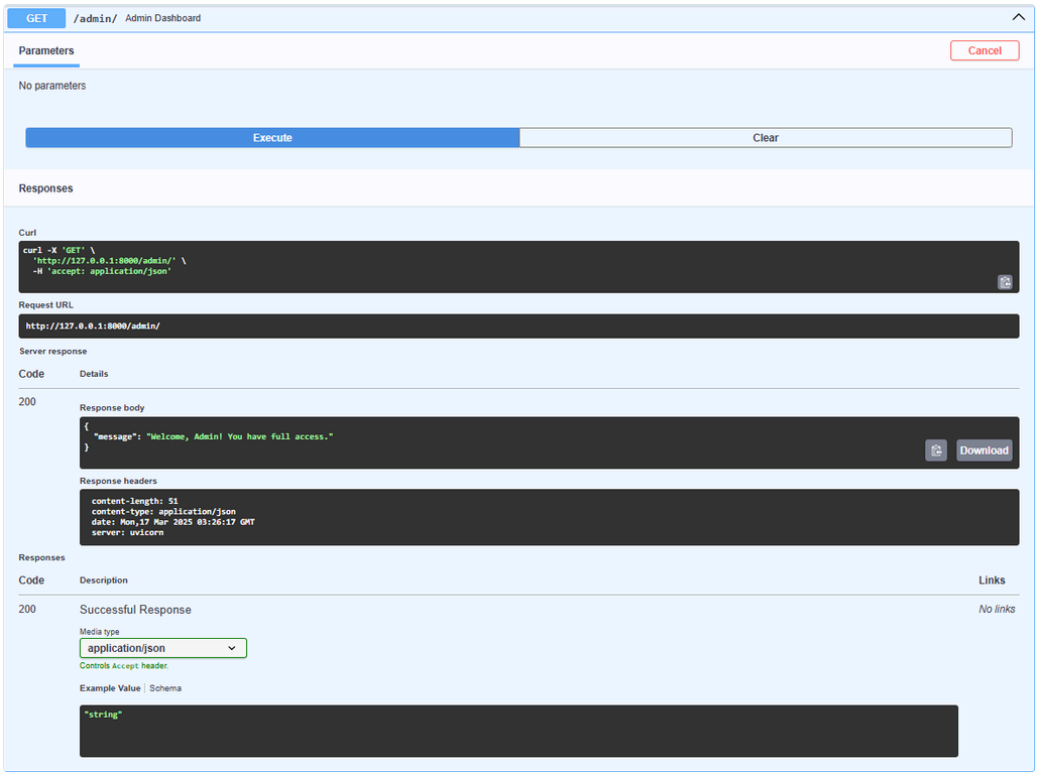
3. The response body confirms:

```
{
   "message": "Welcome, Admin! You have full access."
}
```

- This means that **RBAC successfully recognizes admin access**.
- If a **non-admin** tries to access this endpoint, they should receive a **403 Forbidden** error.

Expected Outcome:

- **Status Code** `200 OK` confirms successful admin authentication.
- Non-admin users should **not** be able to access this endpoint.



Step 6: Protected AI Feature Access (`GET /ai/protected/`)

Description:

This screenshot verifies that **only authenticated (logged-in) users** can access the protected AI feature.

Process:

1. The request is sent to:

```
GET http://127.0.0.1:8000/ai/protected/
```

2. The request includes an **Authorization** header with a valid **user token**:

```
Authorization: Bearer <your_access_token>
```

This ensures that **only authenticated users** can access this AI feature.

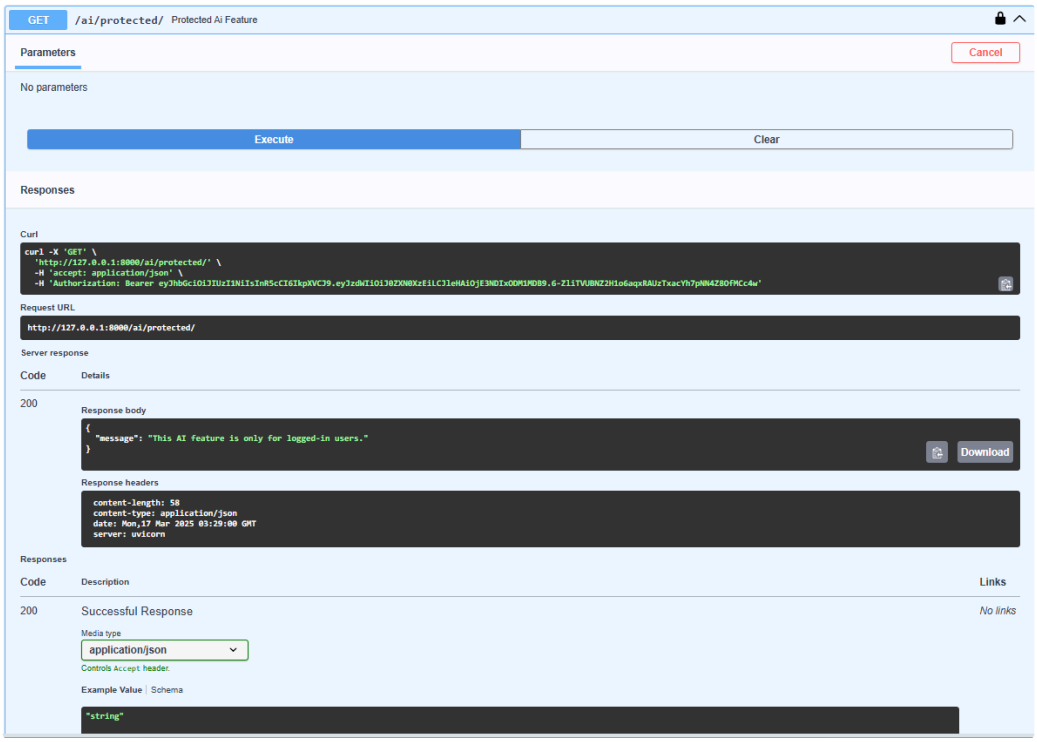
3. The response body confirms:

```
{
   "message": "This AI feature is only for logged-in users."
}
```

- This means the authentication mechanism is **working correctly**.
- If an **unauthenticated** user tries to access this, they should receive a **401 Unauthorized** error.

Expected Outcome:

- **Status Code** `200 OK` confirms successful access.
- If a user **without authentication** tries to access it, they should receive a **401 error**.



Step 7: PDF Upload (`POST /upload-pdf/`)

Description:

This endpoint allows users to upload a **PDF file**, which will be stored in **MongoDB** for later processing, such as text extraction and AI-based queries.

Process:

1. The user selects a PDF file (`AI-powered auto video generation system.pdf`) and sends a `POST` request.
2. The request is made to:

```
POST http://127.0.0.1:8000/upload-pdf/
```

3. The request contains:
   Headers:

```
Content-Type: multipart/form-data
Accept: application/json
```

File Data:

```
File: AI-powered auto video generation system.pdf (application/pdf)
```

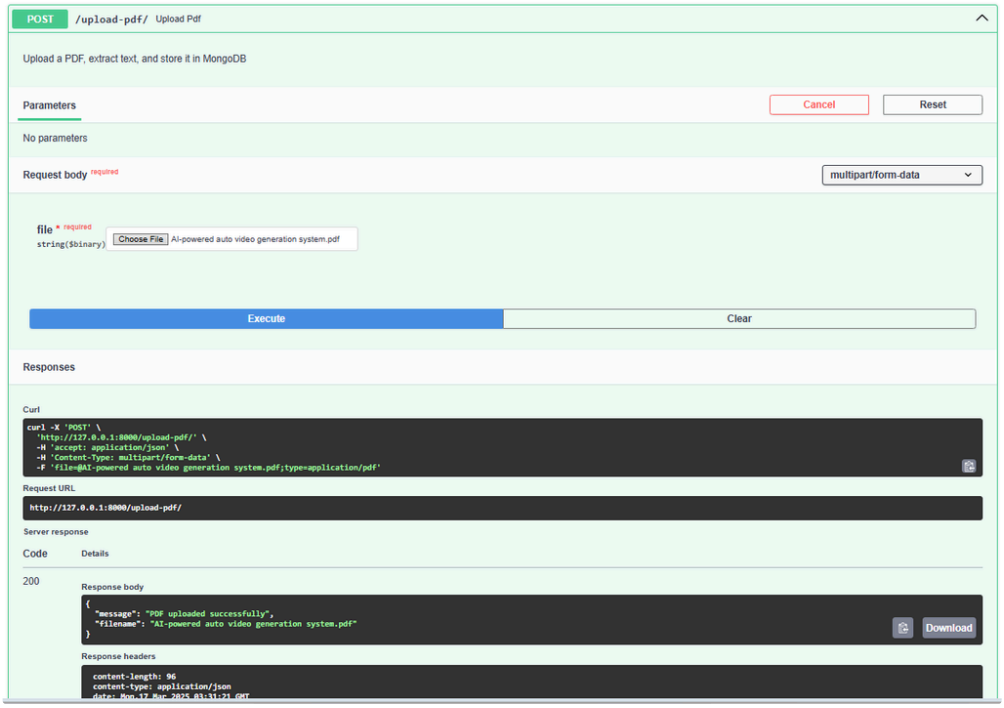
4. **Successful Response:**

```
 {
    "message": "PDF uploaded successfully",
    "filename": "AI-powered auto video generation system.pdf"
}
```

   The file is successfully received and stored.
5. The system acknowledges the upload.

Expected Outcome:

- **Status Code** `200 OK` confirms successful upload.
- The response provides the **stored filename** for further retrieval.



Step 8: AI-Based Question Answering from PDF (`GET /ask-pdf/`)

Description:

This endpoint allows users to query **previously uploaded PDF files**, extracting relevant text and providing AI-generated answers.

Process:

1. The user enters a **query**:

```
"How it will generate image?"
```

2. The user specifies the **filename** of the uploaded PDF:

```
"AI-powered auto video generation system.pdf"
```

3. A `GET` request is sent to:

```
GET http://127.0.0.1:8000/ask-pdf/?query=How%20it%20will%20generate%20image%3F&filename=AI-powered%20auto%20video%20generation%20system.pdf
```

4. The request includes:
   - **Headers:**

```
Accept: application/json
```

   - **Query Parameters:**

```
{  "query": "How it will generate image?",
"filename": "AI-powered auto video generation system.pdf" }
```

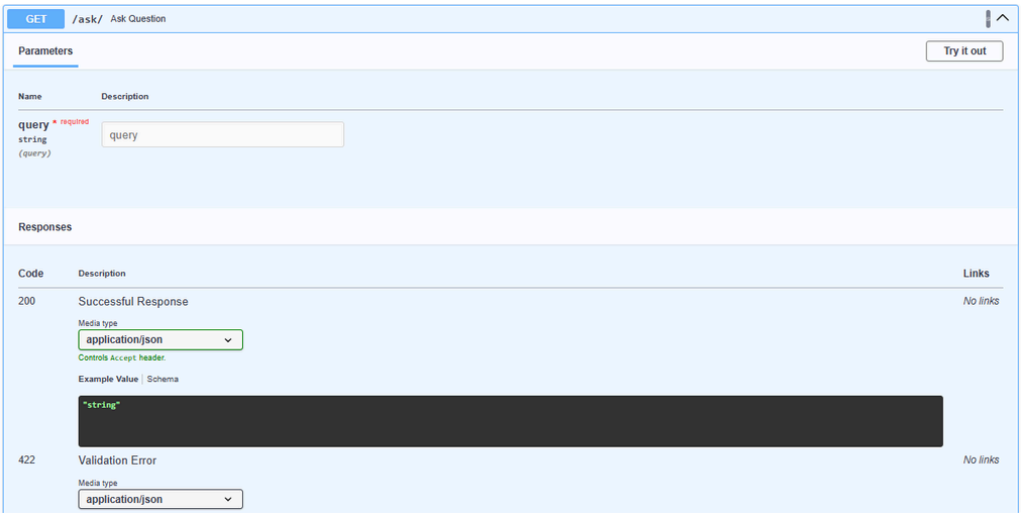
5. **Successful Response:**

```
{ "query": "How it will generate image?",
"answer": "Step 2: Image Creation for Each Scene (DALL-E 3)" }
```

   - The system retrieves the **relevant text from the PDF**.
   - AI processes the text and generates an **answer**.
   - The answer references **Step 2**, which involves **DALL-E 3 for image creation**.

Expected Outcome:

- **Status Code** `200 OK` confirms the query was successfully processed.
- The response **retrieves and interprets** information from the PDF using **AI**.



Step 9: AI-Based General Question Answering (`GET /ask/`)

Description:

This endpoint allows users to **ask any question**, and the system will return an AI-generated response.

How it Works:

1. The user enters a **query** (a question).
2. The system processes the query and generates an **AI-powered response**.
3. The response is returned in JSON format.

Parameters:

Parameter	Type	Required	Description
`query`	String	✅ Yes	The user's question

Example API Call:

```
GET http://127.0.0.1:8000/ask/?query=What%20is%20AI?

```

Request Example:

```
{     "query": "What is AI?" }
```

Expected Responses:

1. **Success (`200 OK`)**
   - The system processes the query and returns a valid answer.

```
{     "query": "What is AI?",
"answer": "Artificial Intelligence (AI) is the simulation of human intelligence in machines..." }
```

2. **Validation Error (`422 Unprocessable Entity`)**
   - If the query parameter is missing or invalid, it returns an error.

```
{     "detail": [
 {             "loc": ["query"],
 "msg": "field required",             "type": "value_error.missing"         }     ] }

```

Purpose of This Endpoint:

- This is a **general AI Q&A system**.
- It is **different from** `/ask-pdf/` because it does **not require a PDF**.
- The model **directly answers general queries**.

Unit Testing

test_auth.py - Authentication & Token Tests

This test file validates **user authentication, password hashing, and JWT token functionality**.

```
import unittest
from app.auth import hash_password, verify_password, create_access_token
from datetime import timedelta
import jwt
import os

SECRET_KEY = os.getenv("SECRET_KEY", "your_secret_key_here")
ALGORITHM = "HS256"

class TestAuth(unittest.TestCase):

    def test_password_hashing(self):
        """
        Test if a password is properly hashed and can be verified correctly.
        """
        password = "test123"
        hashed = hash_password(password)
        self.assertTrue(verify_password(password, hashed))

    def test_token_generation(self):
        """
        Test if a JWT token is correctly generated and contains the right payload.
        """
        data = {"sub": "testuser"}
        token = create_access_token(data, timedelta(minutes=30))
        decoded = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
        self.assertEqual(decoded["sub"], "testuser")

if __name__ == "__main__":
    unittest.main()
```

✅ **Tests:**

✔ Password Hashing & Verification

✔ JWT Token Encoding & Decoding

test_db.py - Database CRUD Tests

This test file ensures **MongoDB user creation and retrieval works correctly**.

```
import unittest
from app.db import get_user, create_user, users_collection

class TestDatabase(unittest.TestCase):

    def test_create_user(self):
        """
        Test if a new user can be successfully created in the database.
        """
        test_user = {
            "username": "testuser",
```

```
            "email": "test@example.com",
            "password": "test123"
        }
        create_user(test_user)
        user = get_user("testuser")
        self.assertIsNotNone(user)
        self.assertEqual(user["username"], "testuser")

    def test_get_nonexistent_user(self):
        """
        Test if trying to fetch a non-existent user returns None.
        """
        user = get_user("nonexistent")
        self.assertIsNone(user)

if __name__ == "__main__":
    unittest.main()
```

✅ **Tests:**

✔ User Creation

✔ Retrieving a Non-Existent User

test_main.py - API Endpoint Tests

This test file ensures **FastAPI endpoints for user registration and login are working**.

```
import unittest
from fastapi.testclient import TestClient
from app.main import app

client = TestClient(app)

class TestMainRoutes(unittest.TestCase):

    def test_register(self):
        """
        Test if a user can successfully register using the API.
        """
        response = client.post("/register/", json={
            "username": "testuser",
            "email": "test@example.com",
            "full_name": "Test User",
            "password": "test123"
        })
        self.assertEqual(response.status_code, 200)

    def test_login(self):
        """
        Test if a user can successfully log in and receive an access token.
        """
        response = client.post("/token/", data={
            "username": "testuser",
            "password": "test123"
        })
        self.assertEqual(response.status_code, 200)
        self.assertIn("access_token", response.json())

if __name__ == "__main__":
```

```
    unittest.main()
```

✅ **Tests:**

✔ User Registration

✔ User Login

test_permissions.py - Role-Based Access Tests

This test file ensures **RBAC (Role-Based Access Control) works correctly**.

```
import unittest
from app.permissions import check_permission

class TestPermissions(unittest.TestCase):

    def test_admin_permissions(self):
        """
        Test if an admin user has the correct permissions.
        """
        self.assertTrue(check_permission("admin", "delete"))
        self.assertTrue(check_permission("admin", "write"))

    def test_user_permissions(self):
        """
        Test if a normal user does NOT have admin-level permissions.
        """
        self.assertTrue(check_permission("user", "read"))
        self.assertFalse(check_permission("user", "delete"))

if __name__ == "__main__":
    unittest.main()
```

✅ **Tests:**

✔ Admin Access Control

✔ User-Level Restrictions

test_ai.py - AI-Based Question Answering

This test file ensures **AI-powered answers are generated properly**.

```
import unittest
from app.ai import process_query

class TestAI(unittest.TestCase):

    def test_ai_response(self):
        """
        Test if the AI processing function returns a valid response.
        """
        response = process_query("What is AI?")
        self.assertIsInstance(response, str)

if __name__ == "__main__":
    unittest.main()
```

✅ **Tests:**

✔ AI-Based Answer Generation

test_pdf.py - PDF Upload & Querying Tests

```python
import unittest
from fastapi.testclient import TestClient
from app.main import app

client = TestClient(app)

class TestPDF(unittest.TestCase):

    def test_pdf_upload(self):
        """
        Test if a PDF file can be uploaded successfully.
        """
        files = {"file": ("test.pdf", b"Fake PDF Content", "application/pdf")}
        response = client.post("/upload-pdf/", files=files)
        self.assertEqual(response.status_code, 200)

    def test_ask_pdf(self):
        """
        Test if the API correctly answers questions based on an uploaded PDF.
        """
        response = client.get("/ask-pdf/", params={"query": "What is in the PDF?"})
        self.assertEqual(response.status_code, 200)

if __name__ == "__main__":
    unittest.main()
```

✅ **Tests:**

✔ PDF Upload API

✔ Querying Uploaded PDF

Running the Tests

```
python tests/test_auth.py
python tests/test_db.py
python tests/test_main.py
python tests/test_permissions.py
python tests/test_ai.py
python tests/test_pdf.py
```

```
pytest tests/
```