# **ToDo List API with User Authentication :**

## **1. Introduction**

### **Overview of the Project**

The **ToDo List API** is a RESTful web service that allows users to create, read, update, and delete (CRUD) tasks securely. It implements **JWT-based authentication** to ensure that each user can only manage their own tasks.

### **Objective and Significance**

* Secure user authentication with JWT tokens
* Task management with CRUD functionality
* Secure password storage using bcrypt
* Database migrations with Flask-Migrate
* Well-documented API with Swagger (Flasgger)
* Deployment-ready with Gunicorn & Nginx

### **Technologies Used**

* **Flask** - Web framework
* **Flask-RESTful** - API development
* **Flask-JWT-Extended** - JWT authentication
* **Flask-SQLAlchemy** - ORM for database
* **Flask-Migrate** - Database migrations
* **Flask-Bcrypt** - Password hashing
* **Flasgger** - API documentation
* **Python-Dotenv** - Environment variable management

## **2. Project Setup**

### **Installing Dependencies**

Run the following command to install all required dependencies:

| pip install -r requirements.txt |
| --- |

### **Project Directory Structure**

| /todo-api │── /app *# Main application folder* │ │── /resources *# Contains API-related functions* │ │ ├── auth.py *# Handles user authentication (Login, Register)* │ │ ├── task.py *# Handles task operations (CRUD)* │ │── /utils *# Helper functions (like password security)* │ │ ├── security.py *# Password hashing & checking* │ │── models.py *# Defines database tables (User, Task)* │ │── routes.py *# Connects API URLs to functions* │── config.py *# Stores configurations like database setup* │── run.py *# Starts the Flask server* │── requirements.txt *# Lists all necessary Python libraries* │── .env *# Stores secret keys & database settings* |
| --- |

## **2.1 How Does This API Work?**

### Think of this API as a digital notebook where each user can log in and create personal tasks. But only the logged-in user can see or edit their own tasks. Here’s how the API works:

### User signs up – We store their username & hashed password.

### User logs in – They get a JWT token for secure access.

### User can create tasks – They add tasks like "Buy groceries" with details.

### User can view, update, and delete tasks – Only their own tasks.

### Security features – Passwords are encrypted, and access is protected with JWT tokens.

### 

### **3. Code Explanation**

### **run.py**

| """ This is the main entry point for the Flask application. It handles application initialization, database setup, and server startup. """  *# Import the application factory function and database instance from the app package* from app import create\_app, db  *# Initialize the Flask application using the factory function* *# This creates the app instance with all configurations, blueprints, and extensions* app = create\_app()  *# Define a custom Flask CLI command to create database tables* @app.cli.command("create-db") def create\_db():  """  Purpose: Creates all database tables defined in the models  Outcome:   - Creates SQL tables for User and Task models if they don't exist  - Prints confirmation message when successful  - Usage: Run 'flask create-db' in terminal  """  db.create\_all() *# SQLAlchemy method to create tables*  print("Database tables created!") *# CLI feedback*  *# Main execution block - only runs when file is executed directly* if \_\_name\_\_ == '\_\_main\_\_':  """  Purpose: Start the Flask development server  Outcome:  - Starts the WSGI server on default port 5000  - Enables debug mode if FLASK\_ENV=development  - Accessible at http://localhost:5000  """ |
| --- |

### **config.py**

| import os from datetime import timedelta  *# Get the absolute path of the current directory (where this script is located)* basedir = os.path.abspath(os.path.dirname(\_\_file\_\_))  class Config:  """  Purpose:  This class defines the configuration settings for the application.   Outcome:  - Sets up the database URI, either from an environment variable or as a fallback to a local SQLite database.  - Disables SQLAlchemy event tracking to improve performance.  - Configures JWT (JSON Web Token) settings for authentication security.  """   *# Define the database URI, using an environment variable if available,*  *# otherwise defaulting to an SQLite database file named 'todo.db' in the current directory.*  SQLALCHEMY\_DATABASE\_URI = os.environ.get('DATABASE\_URL') or \  'sqlite:///' + os.path.join(basedir, 'todo.db')   *# Disable modification tracking feature of SQLAlchemy to reduce overhead.*  SQLALCHEMY\_TRACK\_MODIFICATIONS = False   *# Secret key for JWT authentication, retrieved from environment variables.*  JWT\_SECRET\_KEY = os.environ.get('JWT\_SECRET\_KEY')   *# Set JWT access token expiration time to 1 hour.*  JWT\_ACCESS\_TOKEN\_EXPIRES = timedelta(hours=1) |
| --- |

**app/resources/auth.py**

| from flask\_restful import Resource from flask import request, jsonify from flask\_jwt\_extended import create\_access\_token, jwt\_required, get\_jwt\_identity from app.models import User from app import db  class RegistrationResource(Resource):  def post(self):  """  Purpose: Handles user registration.  Outcome:  - Validates if the username already exists.  - Creates a new user if the username is unique.  - Returns a success message or an error if the username exists.  """  data = request.get\_json()  if User.query.filter\_by(username=data['username']).first():  return {'message': 'Username already exists'}, 400    user = User(username=data['username'])  user.set\_password(data['password'])  db.session.add(user)  db.session.commit()  return {'message': 'User created successfully'}, 201  class AuthResource(Resource):  def post(self):  """  Purpose: Handles user login and generates a JWT token.  Outcome:  - Validates user credentials.  - Returns a JWT token if credentials are valid.  - Returns an error if credentials are invalid.  """  username = request.json.get('username')  password = request.json.get('password')  user = User.query.filter\_by(username=username).first()    if not user or not user.check\_password(password):  return {'message': 'Invalid credentials'}, 401    access\_token = create\_access\_token(identity=str(user.id))  return {'access\_token': access\_token}, 200  class LogoutResource(Resource):  @jwt\_required()  def post(self):  """  Purpose: Handles user logout (placeholder for future token invalidation).  Outcome:  - Returns a success message indicating logout.  """  return {'message': 'Successfully logged out'}, 200 |
| --- |

### **app/resources/task.py**

| from flask\_restful import Resource, reqparse from flask\_jwt\_extended import jwt\_required, get\_jwt\_identity from flask import request from app.models import Task, User, db from datetime import datetime  task\_parser = reqparse.RequestParser() task\_parser.add\_argument('title', type=str, required=True) task\_parser.add\_argument('description', type=str) task\_parser.add\_argument('due\_date', type=str) task\_parser.add\_argument('priority', type=str) task\_parser.add\_argument('status', type=str)  class TaskResource(Resource):  @jwt\_required()  def post(self):  """  Purpose: Creates a new task for the authenticated user.  Outcome:  - Validates input data.  - Creates a new task in the database.  - Returns the created task details.  """  args = task\_parser.parse\_args()  user\_id = int(get\_jwt\_identity())    try:  due\_date = datetime.fromisoformat(args['due\_date']) if args['due\_date'] else None  except ValueError:  return {'message': 'Invalid date format. Use ISO format'}, 400    task = Task(  title=args['title'],  description=args['description'],  due\_date=due\_date,  priority=args.get('priority', 'medium'),  user\_id=user\_id  )    db.session.add(task)  db.session.commit()  return task.to\_dict(), 201   @jwt\_required()  def get(self):  """  Purpose: Retrieves all tasks for the authenticated user.  Outcome:  - Returns a list of tasks associated with the user.  """  user\_id = get\_jwt\_identity()  tasks = Task.query.filter\_by(user\_id=user\_id).all()  return [task.to\_dict() for task in tasks]  class TaskDetailResource(Resource):  @jwt\_required()  def get(self, task\_id):  """  Purpose: Retrieves a specific task by ID for the authenticated user.  Outcome:  - Returns task details if found.  - Returns a 404 error if the task is not found.  """  user\_id = get\_jwt\_identity()  task = Task.query.filter\_by(id=task\_id, user\_id=user\_id).first\_or\_404()  return task.to\_dict()    @jwt\_required()  def put(self, task\_id):  """  Purpose: Updates a specific task by ID for the authenticated user.  Outcome:  - Validates input data.  - Updates the task in the database.  - Returns the updated task details.  """  task = Task.query.filter\_by(  id=task\_id,  user\_id=get\_jwt\_identity()  ).first\_or\_404()    args = task\_parser.parse\_args()    if args['title']:  task.title = args['title']  if args['description']:  task.description = args['description']  if args['due\_date']:  try:  task.due\_date = datetime.fromisoformat(args['due\_date'])  except ValueError:  return {'message': 'Invalid date format. Use ISO format'}, 400  if args['priority']:  task.priority = args['priority']  if args['status']:  task.status = args['status']    db.session.commit()  return task.to\_dict()   @jwt\_required()  def delete(self, task\_id):  """  Purpose: Deletes a specific task by ID for the authenticated user.  Outcome:  - Deletes the task from the database.  - Returns a 204 status code indicating success.  """  task = Task.query.filter\_by(  id=task\_id,  user\_id=get\_jwt\_identity()  ).first\_or\_404()    db.session.delete(task)  db.session.commit()  return '', 204 |
| --- |

**app/utils/security.py**

| from flask\_bcrypt import Bcrypt  bcrypt = Bcrypt()  def hash\_password(password):  """  Purpose: Hashes a password using bcrypt.  Outcome:  - Returns a hashed password string.  """  return bcrypt.generate\_password\_hash(password).decode('utf-8')  def check\_password(hashed\_password, password):  """  Purpose: Verifies a password against its hashed version.  Outcome:  - Returns True if the password matches the hash, otherwise False.  """  return bcrypt.check\_password\_hash(hashed\_password, password) |
| --- |

**app/init.py**

| from flask import Flask from flask\_sqlalchemy import SQLAlchemy from flask\_jwt\_extended import JWTManager from flask\_migrate import Migrate from flasgger import Swagger from config import Config  db = SQLAlchemy() jwt = JWTManager() migrate = Migrate() swagger = Swagger()  def create\_app(config\_class=Config):  """  Purpose: Initializes the Flask application and its extensions.  Outcome:  - Configures the app with settings from the Config class.  - Initializes database, JWT, migrations, and Swagger.  - Registers blueprints for API routes.  - Creates database tables if they don't exist.  """  app = Flask(\_\_name\_\_)  app.config.from\_object(config\_class)   *# Add Swagger configuration*  app.config['SWAGGER'] = {  'title': 'ToDo API',  'uiversion': 3,  'securityDefinitions': {  'BearerAuth': {  'type': 'apiKey',  'name': 'Authorization',  'in': 'header'  }  },  'security': [  {'BearerAuth': []}  ]  }   *# Initialize extensions*  db.init\_app(app)  jwt.init\_app(app)  migrate.init\_app(app, db)  swagger.init\_app(app)   *# Import models to ensure they are registered with SQLAlchemy*  from app.models import User, Task   *# Create tables if they don't exist (for development only)*  with app.app\_context():  db.create\_all()   *# Register blueprints*  from app.routes import api\_blueprint  app.register\_blueprint(api\_blueprint)   return app |
| --- |

### **app/models.py**

| from app import db from app.utils.security import hash\_password, check\_password from datetime import datetime  class User(db.Model):  """  Purpose: Defines the User model for the database.  Outcome:  - Represents a user with username, password, and tasks.  - Provides methods to set and check passwords.  """  id = db.Column(db.Integer, primary\_key=True)  username = db.Column(db.String(64), unique=True, nullable=False)  password\_hash = db.Column(db.String(128), nullable=False)  tasks = db.relationship('Task', backref='author', lazy=True)   def set\_password(self, password):  self.password\_hash = hash\_password(password)   def check\_password(self, password):  return check\_password(self.password\_hash, password)  class Task(db.Model):  """  Purpose: Defines the Task model for the database.  Outcome:  - Represents a task with title, description, due date, priority, and status.  - Provides a method to convert task details to a dictionary.  """  id = db.Column(db.Integer, primary\_key=True)  title = db.Column(db.String(100), nullable=False)  description = db.Column(db.Text)  due\_date = db.Column(db.DateTime)  priority = db.Column(db.String(20), default='medium')  status = db.Column(db.String(20), default='pending')  created\_at = db.Column(db.DateTime, default=datetime.utcnow)  user\_id = db.Column(db.Integer, db.ForeignKey('user.id'), nullable=False)   def to\_dict(self):  return {  'id': self.id,  'title': self.title,  'description': self.description,  'due\_date': self.due\_date.isoformat() if self.due\_date else None,  'priority': self.priority,  'status': self.status,  'created\_at': self.created\_at.isoformat()  } |
| --- |

**app/routes.py**

| from flask import Blueprint from app.resources.auth import RegistrationResource, AuthResource, LogoutResource from app.resources.tasks import TaskResource, TaskDetailResource  api\_blueprint = Blueprint('api', \_\_name\_\_)  *# Authentication routes* api\_blueprint.add\_url\_rule('/register', view\_func=RegistrationResource.as\_view('register')) api\_blueprint.add\_url\_rule('/login', view\_func=AuthResource.as\_view('login')) api\_blueprint.add\_url\_rule('/logout', view\_func=LogoutResource.as\_view('logout'))  *# Task routes* api\_blueprint.add\_url\_rule('/tasks', view\_func=TaskResource.as\_view('tasks')) api\_blueprint.add\_url\_rule('/tasks/<int:task\_id>', view\_func=TaskDetailResource.as\_view('task\_detail')) |
| --- |

## **4. User Authentication & Management**

### **User Registration API (POST /register)**

Allows users to register by providing a username and password.

### **User Login API (POST /login)**

Users receive a **JWT token** upon successful authentication.

### **Secure API Access**

All task-related endpoints require authentication using **JWT Bearer tokens**.

### **Authorization**

Users can only manage their own tasks.

## **5. ToDo Task Management**

### **Creating Tasks (POST /tasks)**

Users can create new tasks with fields like title, description, due\_date, priority, and status.

### **Viewing Tasks (GET /tasks and GET /tasks/<id>)**

Users can retrieve their own tasks.

### **Updating Tasks (PUT /tasks/<id>)**

Allows modifying task details.

### **Deleting Tasks (DELETE /tasks/<id>)**

Users can delete their tasks.

## **6. Security Implementation**

### **JWT Authentication**

* Secure user authentication using **JWT tokens**
* Tokens expire in **1 hour**

### **Password Hashing**

* Uses **bcrypt** to hash passwords before storing them in the database.

### **Role-Based Access Control (Optional)**

* Can be extended to include admin roles for user management.

## **7. Database & Data Handling**

### **Defining Models with SQLAlchemy**

class User(db.Model):

id = db.Column(db.Integer, primary\_key=True)

username = db.Column(db.String(64), unique=True, nullable=False)

password\_hash = db.Column(db.String(128), nullable=False)

tasks = db.relationship('Task', backref='author', lazy=True)

### **Database Migrations with Flask-Migrate**

flask db init

flask db migrate

flask db upgrade

### **Data Validation & Error Handling**

* Validate input formats
* Handle missing fields
* Return proper error messages

## **10. How to Run the Code?**

### **1. Setup Virtual Environment**

| conda create -n venv python=3.11 -y conda activate venv/ |
| --- |

**2.Install Dependencies**

| pip install -r requirements.txt |
| --- |

### **3. Initialize the Database**

Run these commands one by one at cmd.

| flask db init flask db migrate flask db upgrade |
| --- |

### **3. Start the Flask Server**

| python run.py |
| --- |

**4.Now, you can test the API using Postman or Swagger at:**

```

| http://localhost:5000/apidocs |
| --- |

```

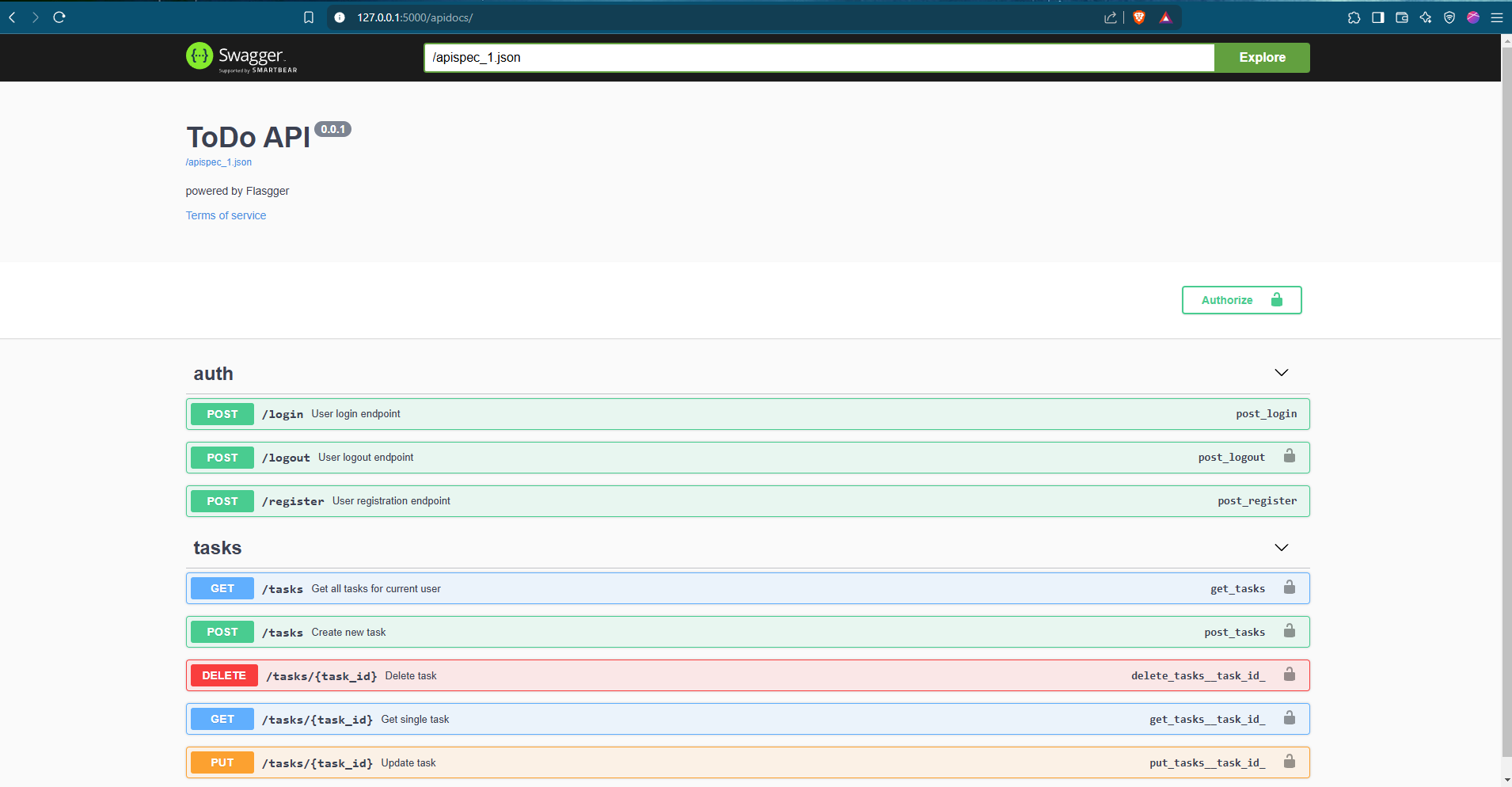
### **5. Use JWT Tokens for API Requests**

* Register/Login to get a token
* Use the token in **Authorization header** (Bearer <token>) for API requests

## **11. The Output**

**1. After running the python run.py :**

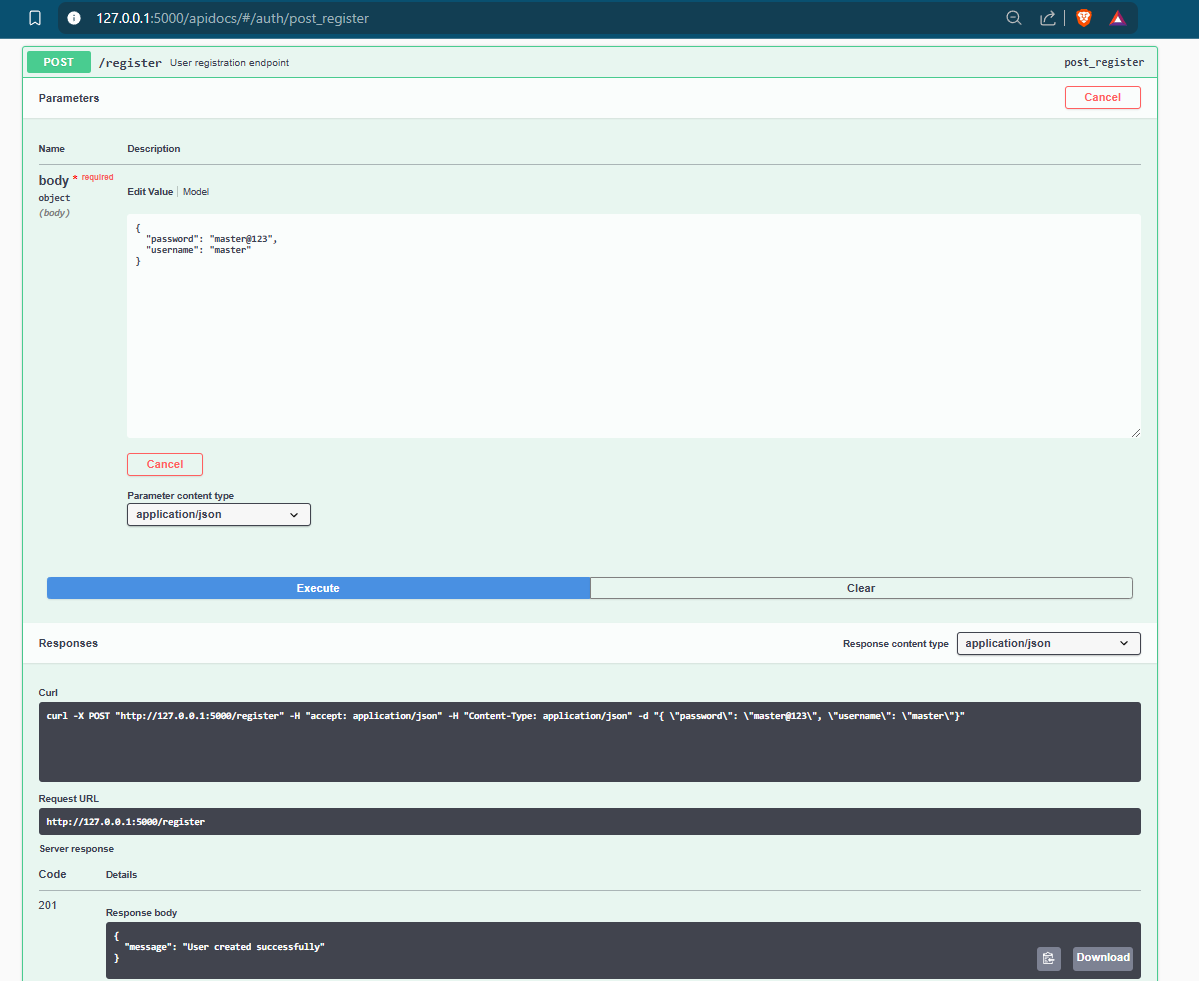
| **http://127.0.0.1:5000/apidocs/** |
| --- |



**2. Now you have to first do register:**

In side of body , give password and username :

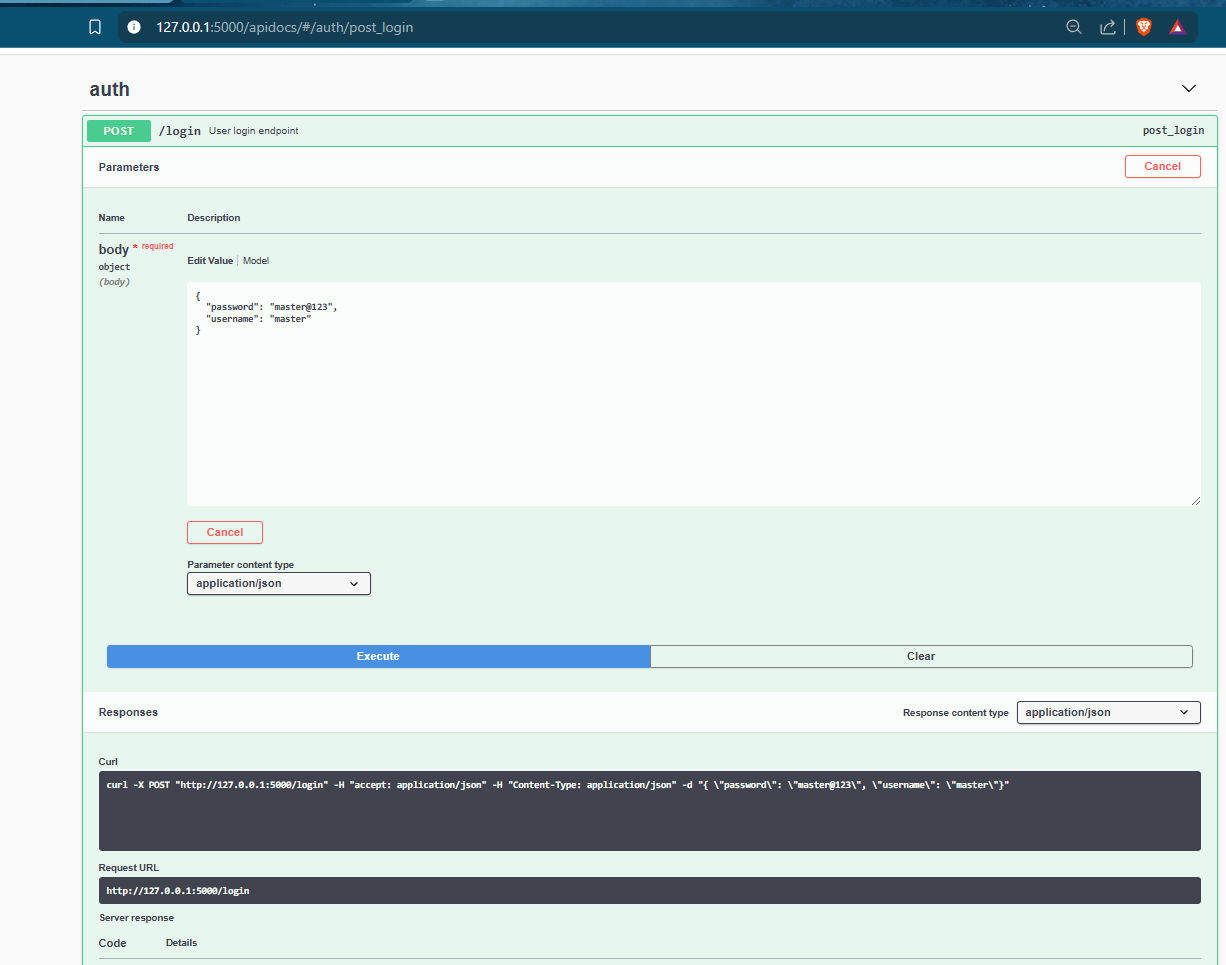
| {  "password": "master@123",  "username": "master" } |
| --- |

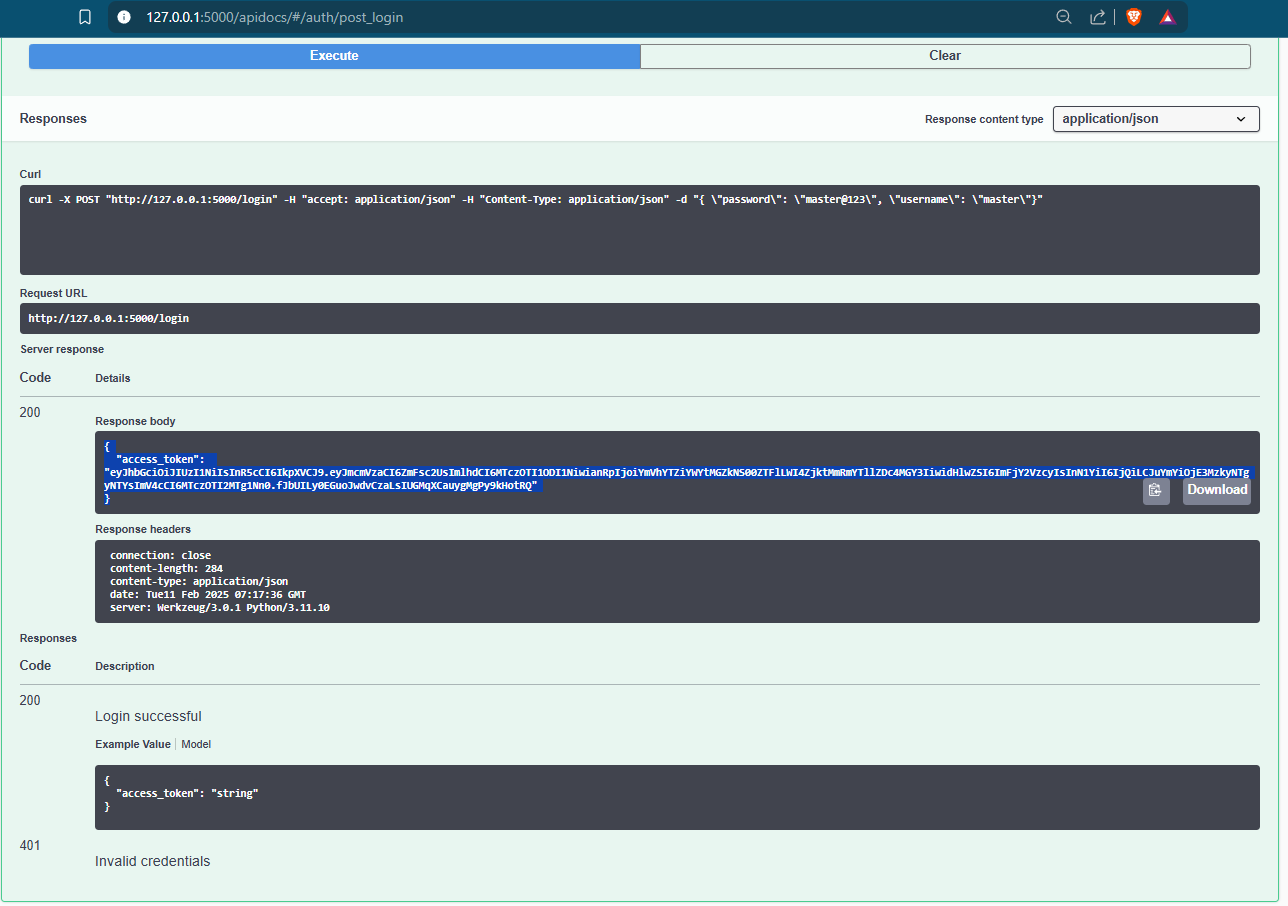


**3. Now click on login:**

**Try it out:**

**Enter the username and password , and click on execute; after successful login you will get access token , you have to copy access token for further use.**

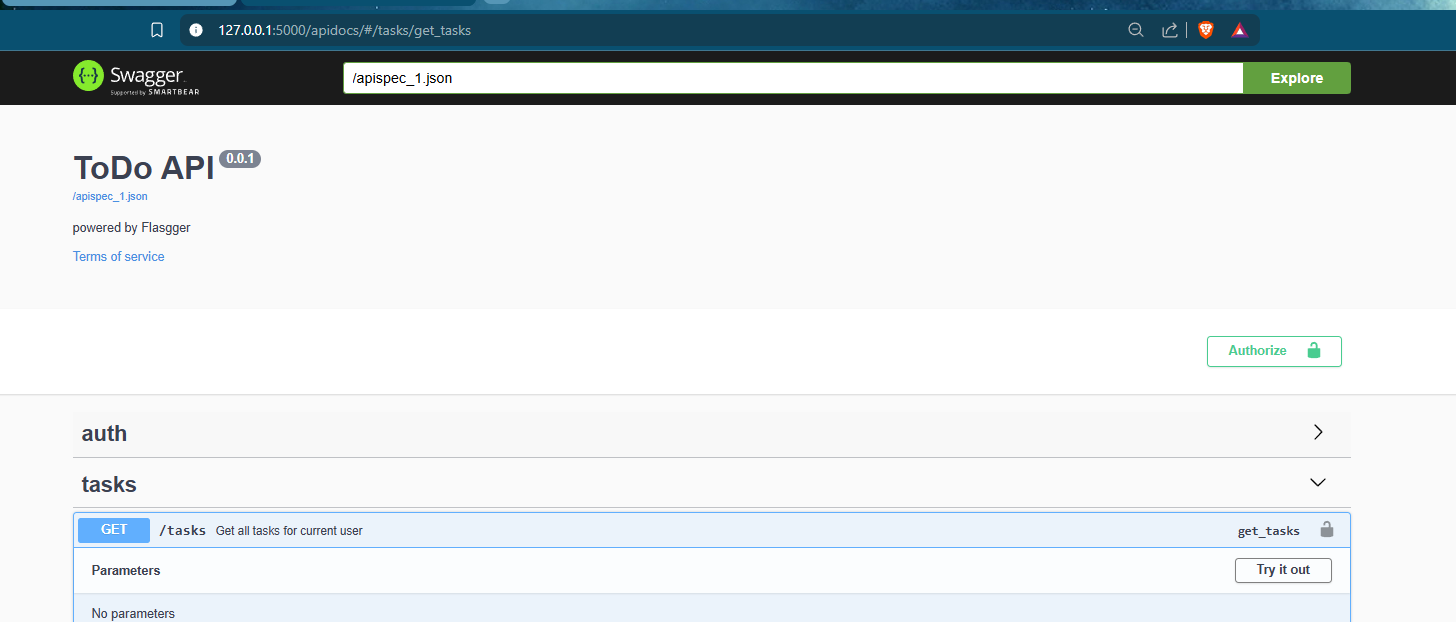




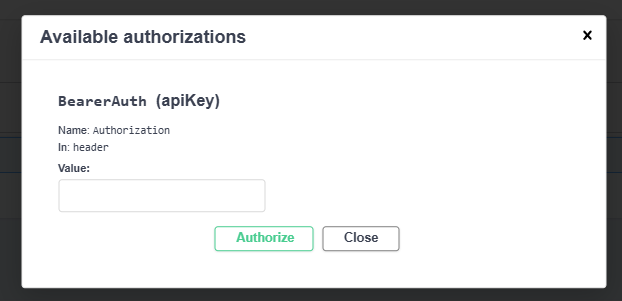
Note :Copy the access token for further use.

**3. Lets create the task:**

**Before the task creation you have to give access token:**



Click on Authorize Button:



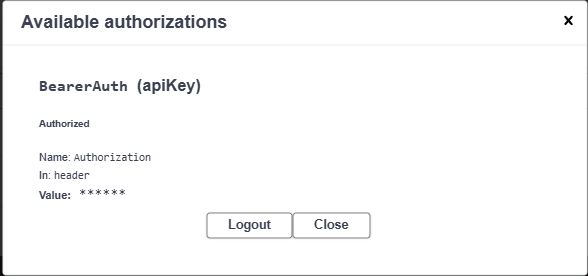
**Here you have to enter your access token:**

**bearer<space>access\_token**

Example:

| Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJmcmVzaCI6ZmFsc2UsImlhdCI6MTczOTI1OTIzMywianRpIjoiNjU2NjYwNTUtYmEzNC00ZGRjLThlNmYtY2UwOTFjZjk3NjQ2IiwidHlwZSI6ImFjY2VzcyIsInN1YiI6IjUiLCJuYmYiOjE3MzkyNTkyMzMsImV4cCI6MTczOTI2MjgzM30.bMApuYVTRXZvnavuS92m3W\_HW2yw4F-K59Nyx5udiiM |
| --- |

After this you will get this:



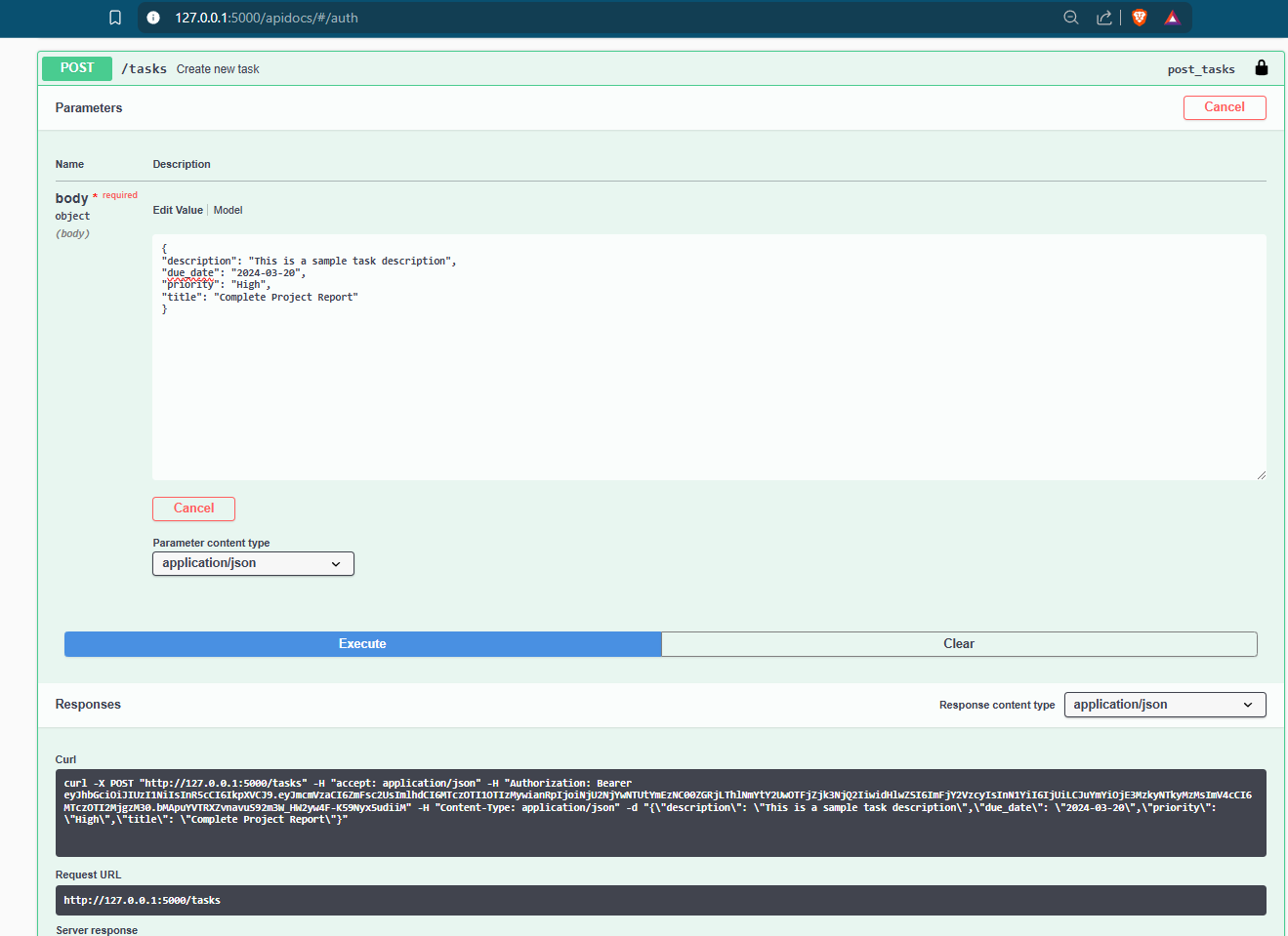
**Click on close.**

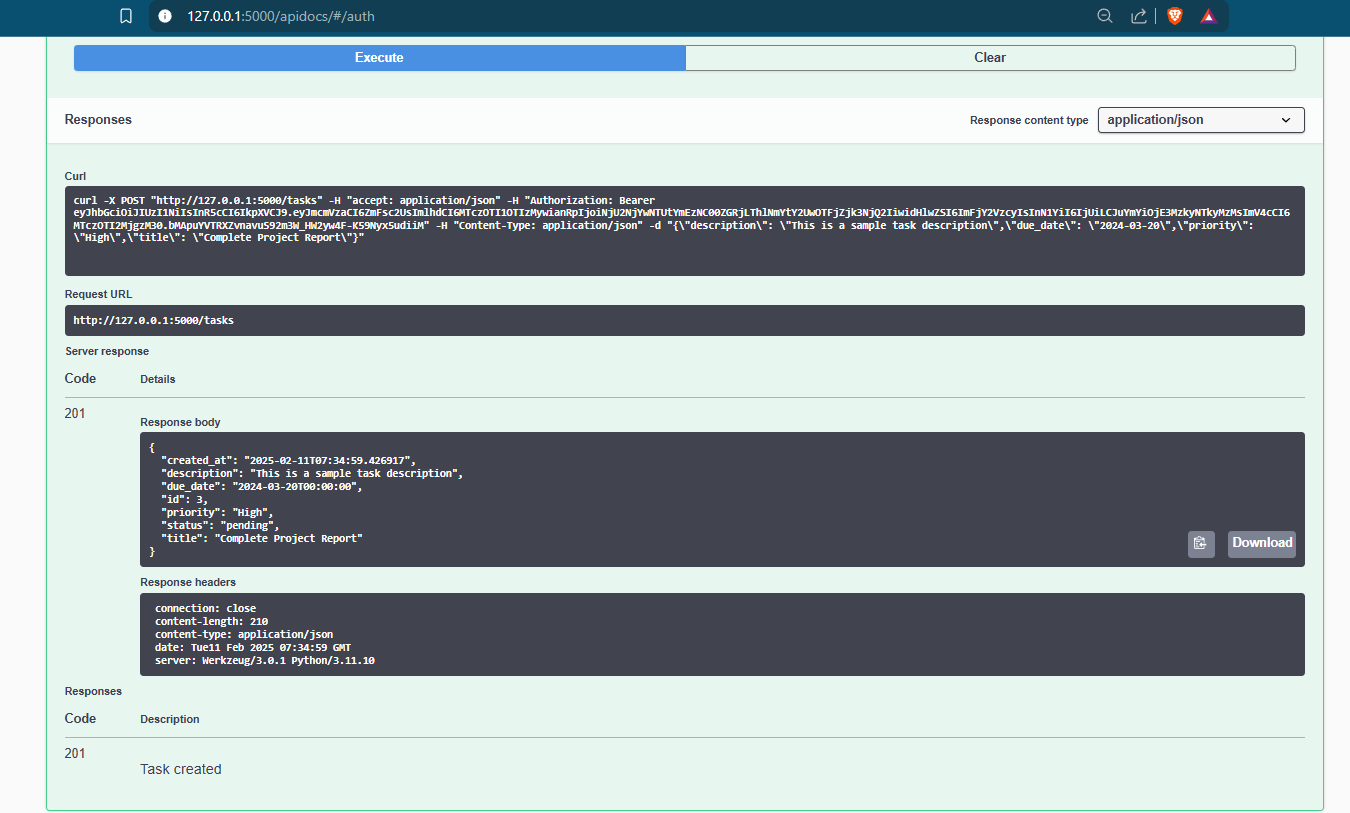
**Now click on tasks to create new task:**

**in the body mention this :**

| {  "title": "Buy groceries",  "description": "Milk, eggs, and bread",  "due\_date": "2023-12-31T23:59:59",  "priority": "medium"  } |
| --- |

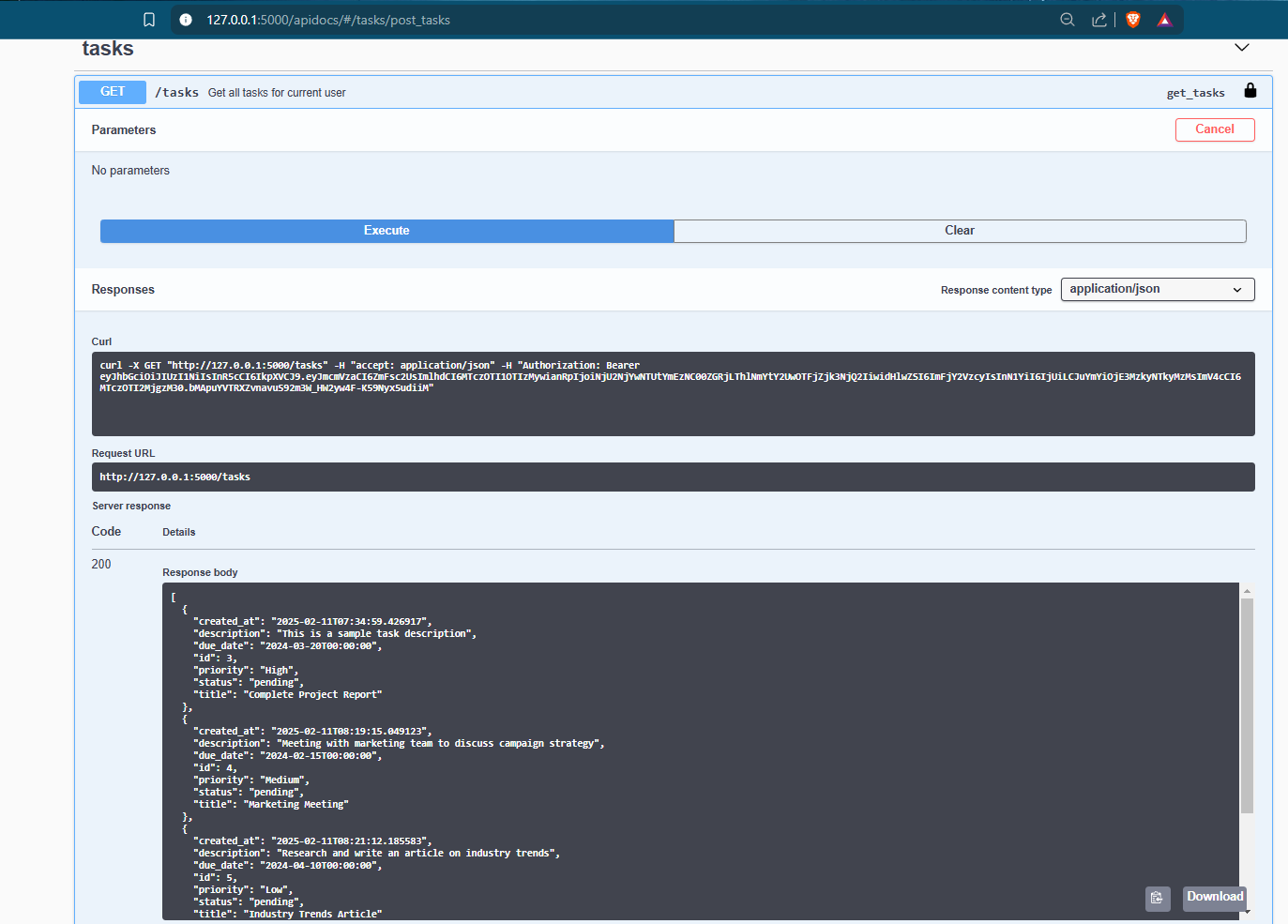
**Then execute :**



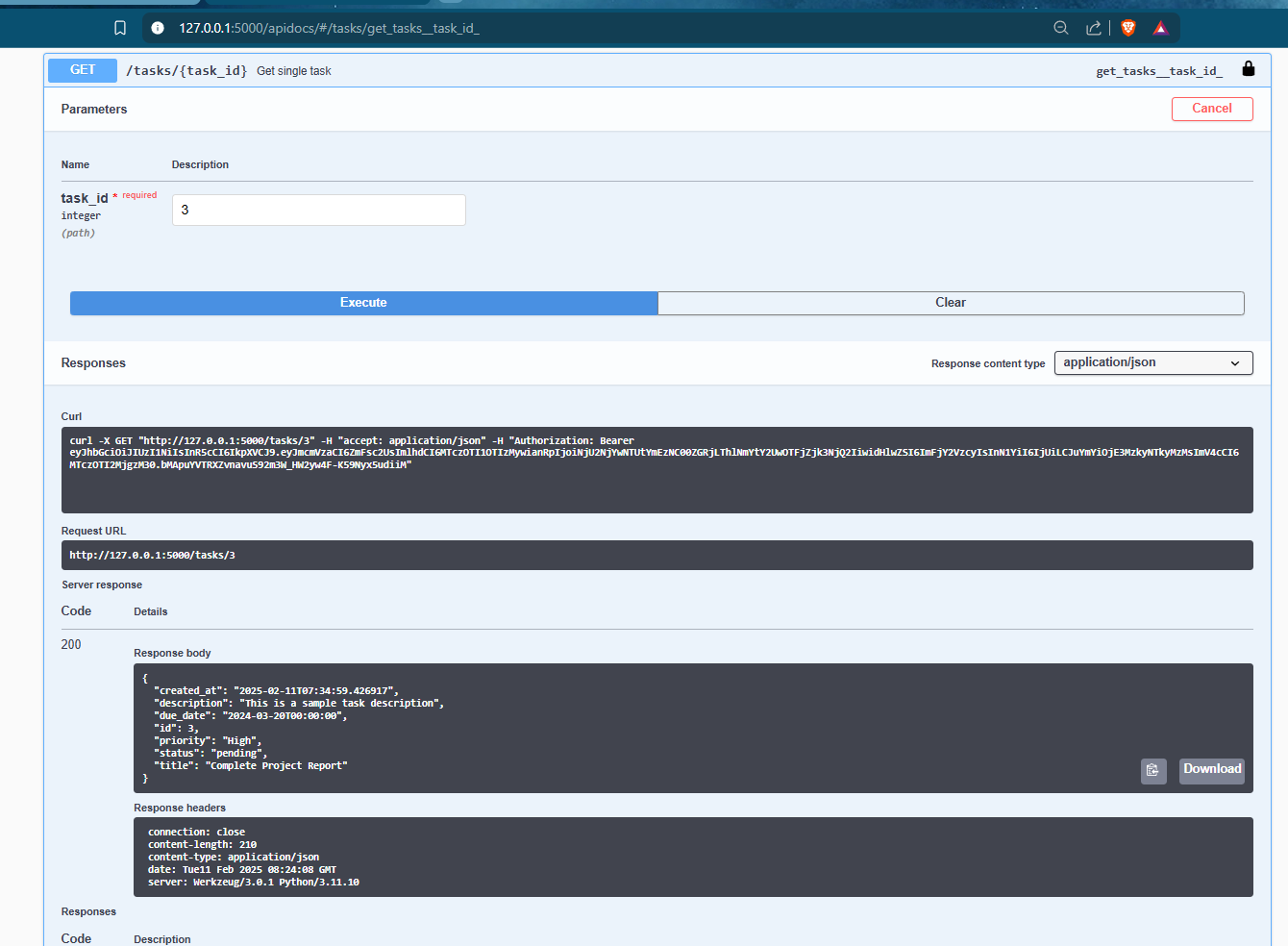


**4. Lets check all the task created :**

**Click on execute button:**



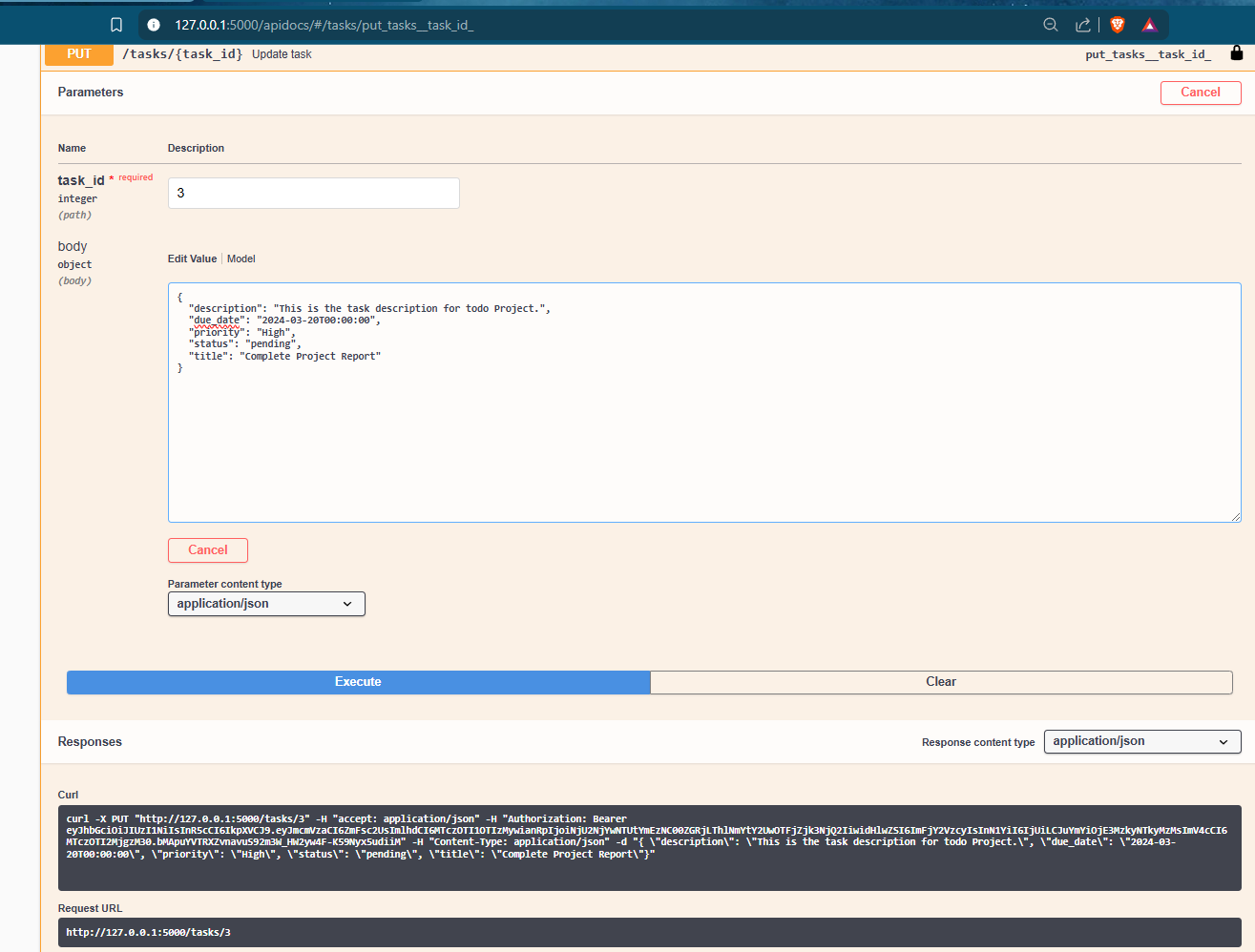
**5. Get the tasks details with the help of task\_id:**

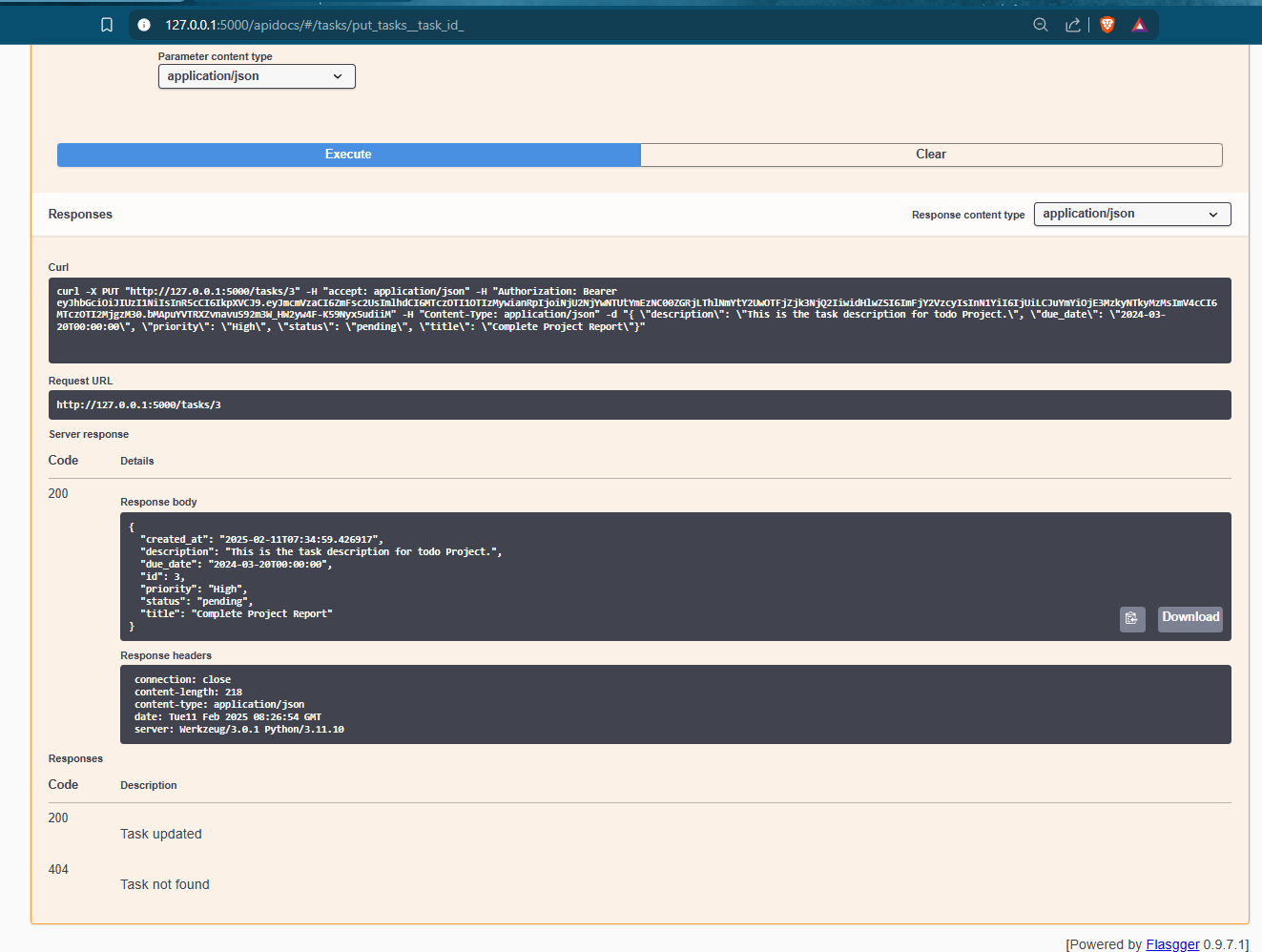


**6.Lets Update the task:**

**Enter the task\_id , and update the input.**

| {  "description": "This is the task description for todo Project.",  "due\_date": "2024-03-20T00:00:00",  "priority": "High",  "status": "pending",  "title": "Complete Project Report" } |
| --- |

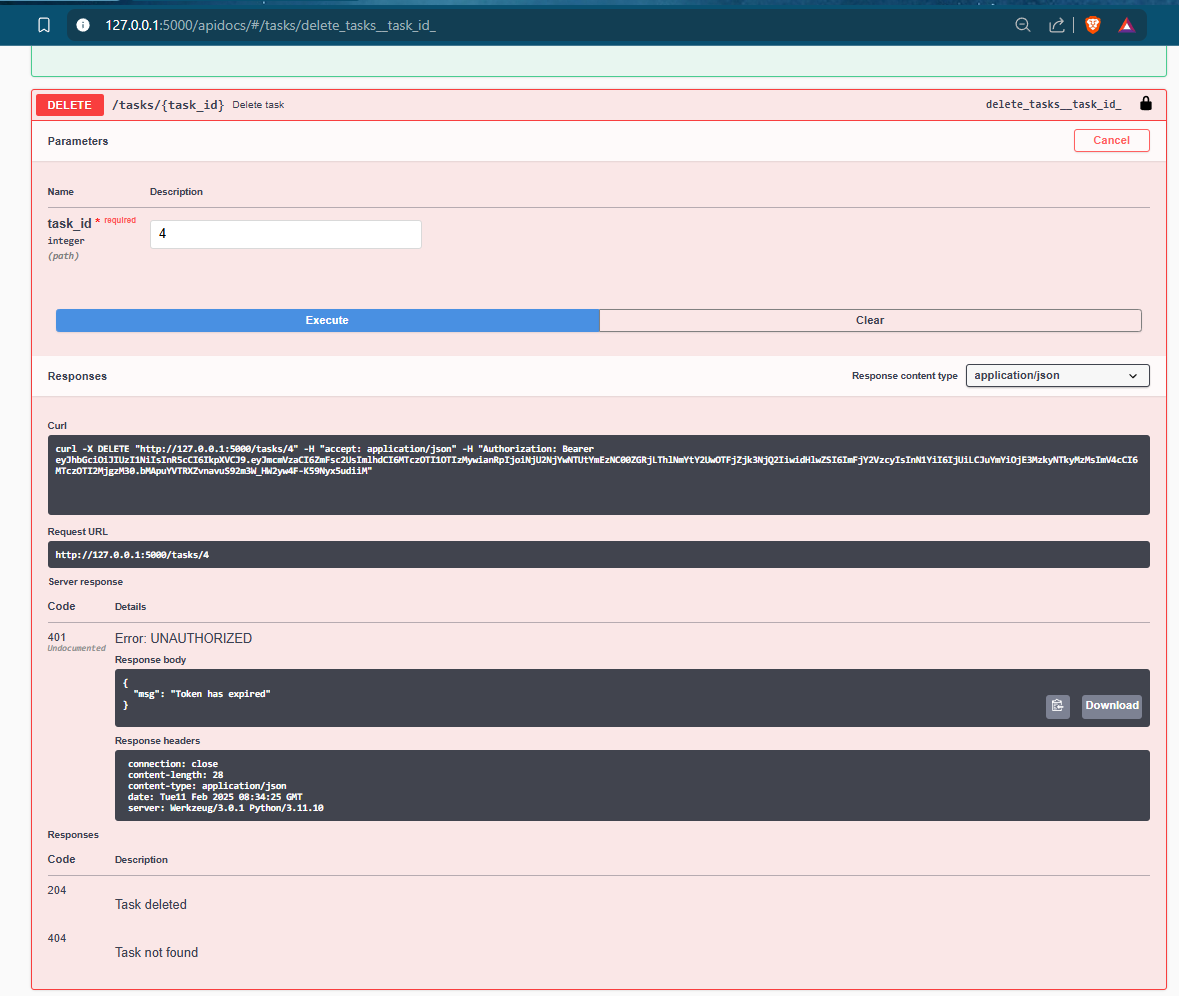




**7.Let's delete the task: click on {Try it out}**

**just mention the task\_id :4,**

**Then click on Execute:**



**12. Future Enhancements**

### **1. Implement OAuth Authentication**

* Support Google, GitHub authentication

### **2. Task Categorization & Reminders**

* Categorize tasks
* Set reminders & push notifications

### **3. Multi-User & Shared Task Lists**

* Allow users to share tasks
* Implement collaborative task management

### **4. Microservices Architecture with Docker & Kubernetes**

* Deploy as a microservice
* Use **Docker & Kubernetes** for scalability

## **13.Conclusion**

This **ToDo List API** provides a secure and efficient way for users to manage tasks. It follows RESTful best practices and includes robust authentication, database handling, security measures, and API documentation. Future enhancements will make it even more feature-rich and scalable for enterprise use.