🚀 AI Configuration

🔑 OpenAI API Key

••

✅ API Key configured

AI Model

gpt-4o

Selected Model: gpt-4o

🌐 Custom API Base URL (Optional)

http://localhost:11434/v1 (optional)

⚙️ Advanced Settings

📚 About Multi-Agent Framework

Multi-Agentic System powered by AutoGen with GPT-4o

This framework orchestrates 7 specialized AI agents that collaborate to transform natural language requirements into production-ready code with full documentation, tests, and deployment configuration.

🔄 Agent Pipeline

1. 📋 Requirement Analyst - Structure requirements
2. 💻 Senior Developer - Generate code
3. 🔍 Code Reviewer - Review & iterate (AutoGen loop)
4. 📖 Tech Writer - Create documentation
5. 🧪 QA Engineer - Generate tests
6. 🚀 DevOps - Deployment config
7. 🎨 UI Designer - Streamlit interface

🚀 AutoGen Multi-Agent Code Generator

Transform Ideas into Production-Ready Code with AI Agent Collaboration

AI Agents	Framework	Model	Version
7	Multi-Agent	GPT-4o	2026

📝 Enter Your Requirements

Describe what you want to build:

Create a Fast API REST API for a Todo List Manager with the following features:

1. CRUD Operations:
 - Create new Todo items with title, description, priority (low/medium/high), and due date

💡 Quick Start Examples

🚀 Generate Code with AI Agents

📊 AutoGen Pipeline Results

Generated Artifacts from Multi-Agent Collaboration

📈 Execution Metrics

	Review Iterations	Iteration Limit	Run ID
✅ SUCCESS	2	Within Limit	37e66994

📋 Requirements Analysis | 💻 Python Code | 🔍 Code Review | 📖 Documentation | 🧪 Test Suite | 🚀 Deployment

📖 Technical Documentation

Generated by Tech Writer Agent

Todo List Manager API

Overview

The Todo List Manager API is a FastAPI-based application that provides a RESTful interface for managing todo items. It supports CRUD operations, allowing users to create, read, update, and delete todo items. The API also includes features such as input validation, rate limiting, CORS support, and logging. It is designed for easy integration with frontend applications and supports filtering of todos by status and priority.

Installation

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

Architecture

The application is structured as follows:

- **FastAPI**: The main framework used to create the API endpoints.
- **SQLAlchemy**: ORM used for database interactions.
- **Pydantic**: Used for data validation and serialization.
- **Rate Limiting**: Implemented using FastAPILimiter to control request rates.
- **CORS Middleware**: Configured to allow cross-origin requests.
- **Logging**: Configured to log important events and errors.

API Reference

Class: TodoModel

Purpose: Represents the database model for a todo item.

Attributes:

- `id` (Integer): Primary key.
- `title` (String): Title of the todo.
- `description` (String): Description of the todo.
- `status` (StatusEnum): Current status of the todo.
- `priority` (PriorityEnum): Priority level of the todo.
- `due_date` (DateTime): Due date for the todo.
- `created_at` (DateTime): Timestamp of creation.
- `updated_at` (DateTime): Timestamp of last update.

Class: TodoBase

Purpose: Base Pydantic model for todo items.

Method: validate_due_date

Parameters:

- `v` (datetime): Due date to validate.

Returns: datetime - Validated due date.

Raises:

- `ValueError` : If the due date is in the past.

Example:

```
todo = TodoBase(title="Sample", description="Sample description", priority=PriorityEnum.medium, due_date=datetime(2023, 12, 31))
```

Endpoints

POST /todos

Purpose: Create a new todo item.

Parameters:

- `todo` (TodoCreate): Todo data.

Returns: TodoResponse - Created todo item.

Raises:

- `HTTPException` : If there is a database error.

Example:

```
response = requests.post("/todos", json={"title": "New Task", "description": "Task description", "priority": "medium", "due_date": "2023-12-31T00:00:00"})
```

GET /todos

Purpose: Retrieve all todos with optional filtering.

Parameters:

- `status` (Optional[StatusEnum]): Filter by status.
- `priority` (Optional[PriorityEnum]): Filter by priority.

Returns: List[TodoResponse] - List of todos.

Raises:

- `HTTPException` : If there is a database error.

Example:

```
response = requests.get("/todos?status=pending&priority=high")
```

GET /todos/{id}

Purpose: Retrieve a specific todo item by ID.

Parameters:

- `id` (int): Todo ID.

Returns: TodoResponse - Todo item.

Raises:

- `HTTPException` : If the todo is not found.

Example:

```
response = requests.get("/todos/1")
```

PUT /todos/{id}

Purpose: Update a specific todo item by ID.

Parameters:

- `id` (int): Todo ID.
- `todo` (TodoUpdate): Updated todo data.

Returns: TodoResponse - Updated todo item.

Raises:

- `HTTPException` : If the todo is not found or there is a database error.

Example:

```
response = requests.put("/todos/1", json={"status": "completed"})
```

DELETE /todos/{id}

Purpose: Delete a specific todo item by ID if it is completed.

Parameters:

- `id` (int): Todo ID.

Returns: None

Raises:

- `HTTPException` : If the todo is not found or not completed.

Example:

```
response = requests.delete("/todos/1")
```

GET /todos/stats

Purpose: Retrieve statistics including total, completed, and pending todos.

Returns: dict - Statistics data.

Raises:

- `HTTPException` : If there is a database error.

Example:

```
response = requests.get("/todos/stats")
```

Usage Examples

1. **Creating a Todo:**

   ```
   response = requests.post("/todos", json={
       "title": "Buy groceries",
       "description": "Milk, Bread, Cheese",
       "priority": "medium",
       "due_date": "2023-12-31T00:00:00"
   })
   ```

2. **Listing Todos with Filters:**

   ```
   response = requests.get("/todos?status=pending&priority=high")
   ```

3. **Updating a Todo:**

   ```
   response = requests.put("/todos/1", json={"status": "completed"})
   ```

4. **Deleting a Completed Todo:**

   ```
   response = requests.delete("/todos/1")
   ```

5. **Getting Todo Statistics:**

   ```
   response = requests.get("/todos/stats")
   ```

Configuration

- **Environment Variables:**
  - `DATABASE_URL` : URL for the database connection (e.g., `sqlite:///./test.db`).
- **Config Files:**
  - Use `.env` file to store environment variables for local development.

Error Handling

- **Common Errors:**
  - `HTTP_404_NOT_FOUND` : Raised when a todo item is not found.
  - `HTTP_400_BAD_REQUEST` : Raised when trying to delete a non-completed todo.
  - `HTTP_500_INTERNAL_SERVER_ERROR` : Raised for database errors.

Best Practices

- Use environment variables for configuration to enhance security.
- Validate all inputs using Pydantic models to prevent invalid data.
- Implement proper error handling to provide meaningful error messages.

- Use rate limiting to prevent abuse of the API.

Troubleshooting

- **Database Connection Issues:**
  - Ensure the `DATABASE_URL` is correctly set and accessible.
  - Check the database server status if using a remote database.
- **Rate Limiting Errors:**
  - If receiving a `429 Too Many Requests` error, reduce the request rate or increase the rate limit settings.
- **CORS Issues:**
  - Ensure the frontend origin is allowed in the CORS configuration.

This documentation provides a comprehensive guide to understanding, installing, and using the Todo List Manager API. It includes detailed API references, usage examples, and configuration instructions to facilitate easy integration and deployment.

Download Documentation (Markdown)

Download All Artifacts (ZIP)

Multi-Agentic Framework © 2026
Powered by AutoGen with GPT-4o • Version 2026.1.0