

从零到一开发指南：像专业开发者一样构建 Todo API

目标读者：正在学习专业开发实践的初学者 **项目：**Todo API (Python + FastAPI + SQLAlchemy) **最后更新：**2026-02-14

目录

- [一、技术栈选择：我们为什么选择这些工具](#)
 - [1.1 Web 框架：FastAPI](#)
 - [1.2 ORM：SQLAlchemy](#)
 - [1.3 数据库：SQLite](#)
 - [1.4 依赖管理：pip + requirements.txt](#)
- [二、环境搭建：专业的第一步](#)
 - [2.1 确认 Python 版本](#)
 - [2.2 创建虚拟环境](#)
 - [2.3 安装项目依赖](#)
- [三、核心代码逐文件解析](#)
 - [3.1 config.py — 配置管理](#)
 - [3.2 database.py — 数据库连接](#)
 - [3.3 models/todo.py — 数据库模型](#)
 - [3.4 schemas/todo.py — 数据验证](#)
 - [3.5 routers/todo.py — API 路由](#)
 - [3.6 main.py — 应用入口](#)
- [四、API 端点详解](#)
- [五、数据流完整走读](#)
- [六、关键概念深入理解](#)
- [七、专业开发者的思维清单](#)
- [八、常见疑问解答](#)

一、技术栈选择：我们为什么选择这些工具

在动手写代码之前，专业开发者做的第一件事是**技术选型**——选择合适的工具。就像做饭前要决定用炒锅还是蒸锅，选择工具决定了你后续的开发体验。

1.1 Web 框架：FastAPI

什么是 Web 框架？

当你在浏览器里输入一个网址、或者手机 App 请求数据时，背后发生的事情是：

浏览器/App —发送请求—> 服务器 —返回数据—> 浏览器/App
(HTTP) (JSON)

Web 框架就是帮你搭建“服务器”这一端的工具。它帮你处理了：

- **网络通信**: 怎么接收 HTTP 请求
- **URL 路由**: 不同的 URL 对应不同的处理函数
- **数据解析**: 怎么解读请求中的数据 (JSON, 表单等)
- **响应构造**: 怎么把结果包装成标准的 HTTP 响应

为什么选 FastAPI 而不是 Flask 或 Django?

特性	FastAPI	Flask	Django
性能	⚡ 非常快 (异步)	一般	一般
自动 API 文档	☑ 自动生成	✗ 需要插件	✗ 需要插件
数据验证	☑ 内置 (Pydantic)	✗ 需要手动	部分内置
学习曲线	中等	低	高
类型提示	☑ 原生支持	可选	可选
适合场景	API 开发	小型项目	全栈 Web 应用

FastAPI 的杀手锏：**你写好代码后，API 文档自动就有了**，而且可以直接在浏览器里测试。

1.2 ORM： SQLAlchemy

什么是 ORM?

ORM = Object-Relational Mapping (对象-关系映射)

它在 Python 对象和数据库表之间做翻译：



为什么不直接写 SQL?

1. **安全性**: ORM 自动防止 SQL 注入攻击 (一种常见的安全漏洞)
2. **可移植性**: 换数据库时 (SQLite → PostgreSQL)，代码基本不用改
3. **可读性**: Python 代码比 SQL 更容易理解和维护
4. **类型安全**: IDE 能自动提示字段名，拼错了立即报红

SQLAlchemy 的地位

SQLAlchemy 在 Python 生态中的地位，相当于 React 在前端的地位——不是唯一选择，但是最主流的选择。大量的企业级项目都在使用它。

1.3 数据库：SQLite

为什么用 SQLite 而不是 MySQL/PostgreSQL？

数据库	安装方式	适合场景
SQLite	不用安装！数据存在一个 <code>.db</code> 文件里	开发、学习、小项目
MySQL	需要安装并启动数据库服务器	中型 Web 应用
PostgreSQL	需要安装并启动数据库服务器	大型企业应用

SQLite 的优势：

- **零配置**：Python 自带 SQLite 支持，不需要安装任何东西
- **文件级**：整个数据库就是一个 `todo.db` 文件
- **足够学习使用**：SQL 语法和其他数据库是一样的

 **专业技巧：**在开发阶段用 SQLite 快速迭代，项目成熟后切换到 PostgreSQL。这就是 ORM 的好处——切换数据库时，只需要改 `DATABASE_URL` 配置，代码不用改。

1.4 依赖管理：pip + requirements.txt

什么是依赖？

你的项目使用了别人写好的代码库（如 FastAPI、SQLAlchemy），这些就是“依赖”。你的项目“依赖”它们才能运行。

依赖管理要解决三个问题：

1.  记录：我的项目用了哪些库?
→ `requirements.txt`
2.  隔离：不同项目的依赖不要互相干扰
→ 虚拟环境 (`.venv/`)
3.  复现：别人拿到代码后能一键安装
→ `pip install -r requirements.txt`

类比：

- `requirements.txt` = 菜谱里的“食材清单”
- `.venv/` = 这道菜专用的厨房（不和其他菜共用锅碗瓢盆）
- `pip install` = 按照清单去超市采购

二、环境搭建：专业的第一步

2.1 确认 Python 版本

```
# 查看系统上安装了哪些 Python 版本  
py --list  
  
# 输出示例:  
# -V:3.13 * Python 3.13 (64-bit)  
# -V:3.10 Python 3.10 (64-bit)
```

💡 为什么版本很重要?

不同 Python 版本支持的语法不同。例如：

- `str | None` 这种写法需要 Python 3.10+
- `match` 语句需要 Python 3.10+

如果你用的语法在目标版本中不支持，代码会报语法错误。专业项目都会在文档中注明最低 Python 版本要求。

2.2 创建虚拟环境

```
# 使用 Python 3.13 创建虚拟环境  
py -3.13 -m venv .venv  
  
# 激活虚拟环境 (Windows PowerShell)  
.venv\Scripts\Activate.ps1  
  
# 激活后，提示符前面会出现 (.venv) 标识  
# (.venv) PS D:\Code\project\todo-api>
```

虚拟环境的本质

.venv/ 目录里实际上就是一个“迷你的 Python 安装”：

```
.venv/  
└── Scripts/  
    ├── python.exe      ← 可执行文件 (python.exe, pip.exe 等)  
    ├── pip.exe          ← 虚拟环境专用的 Python  
    └── Activate.ps1    ← 虚拟环境专用的 pip  
└── Lib/  
    └── site-packages/  ← 虚拟环境安装的第三方库 (隔离的！)  
        ├── fastapi/  
        ├── sqlalchemy/  
        └── ...
```

当虚拟环境激活后，你运行 `python` 和 `pip` 时，用的都是 .venv/ 里的版本，而不是系统全局的版本。这就实现了“隔离”。

关于激活虚拟环境

操作系统	Shell	激活命令
Windows	PowerShell	.venv\Scripts\Activate.ps1
Windows	CMD	.venv\Scripts\activate.bat
macOS/Linux	bash/zsh	source .venv/bin/activate

⚠ 注意：每次打开新的终端窗口，都需要重新激活虚拟环境。如果你发现 `pip install` 的包找不到，很可能是忘记激活了。

2.3 安装项目依赖

```
# 安装核心依赖
pip install fastapi uvicorn[standard] sqlalchemy aiosqlite pydantic-settings

# 将当前安装的所有包及版本号保存到 requirements.txt
pip freeze > requirements.txt
```

每个依赖包的作用

包名	角色	类比
fastapi	Web 框架	饭店的“厨房设备”
uvicorn	ASGI 服务器	“服务员”——把客人的点单送到厨房
sqlalchemy	ORM	“翻译官”——Python 和数据库之间的桥梁
aiosqlite	SQLite 异步驱动	“翻译官的助手”——让翻译更快速高效
pydantic-settings	配置管理	“行政助理”——帮你管理所有配置信息

什么是 uvicorn？为什么 FastAPI 还需要它？

FastAPI 只是一个“框架”——它知道如何处理请求、调用你的函数、返回响应。但它**不知道如何监听网络端口、接收 TCP 连接**。

uvicorn 就是那个负责“监听端口、接收连接”的程序。它是 ASGI 服务器。

```
网络请求 → uvicorn (接收连接) → FastAPI (处理请求) → uvicorn (发送响应) → 客户端
      "门卫 + 服务员"           "厨师"                  "服务员"
```

三、核心代码逐文件解析

3.1 config.py — 配置管理

```

from pydantic_settings import BaseSettings

class Settings(BaseSettings):
    APP_NAME: str = "Todo API"          # 应用名称, 默认值 "Todo API"
    APP_VERSION: str = "0.1.0"           # 版本号
    DEBUG: bool = True                  # 是否启用调试模式
    DATABASE_URL: str = "sqlite+aiosqlite:///./todo.db"  # 数据库连接地址

    class Config:
        env_file = ".env"                # 从 .env 文件读取配置

settings = Settings()                 # 全局唯一的配置实例

```

配置读取的优先级 (从高到低) :

- | | |
|---------------|------------------|
| 1. 系统环境变量 | ← 最高优先级 (生产环境常用) |
| 2. .env 文件中的值 | ← 开发环境常用 |
| 3. 代码中的默认值 | ← 命底值 |

数据库 URL 格式解析

sqlite+aiosqlite:///./todo.db

- 异步驱动 (aiosqlite)
- 数据库类型 (sqlite)
- 三个斜杠表示使用文件路径
- 数据库文件名 (在当前目录下的 todo.db)

如果是 PostgreSQL:

postgresql+asyncpg://username:password@host:port/dbname

应当熟练掌握的知识:

- 理解为什么配置要与代码分离
- 能看懂数据库 URL 的各个部分
- 知道 .env 文件的作用和安全注意事项
- 理解“单例模式”——为什么全局只需要一个 settings 实例

3.2 database.py — 数据库连接

```

from sqlalchemy.ext.asyncio import AsyncSession, async_sessionmaker,
create_async_engine

```

```
# Step 1: 创建引擎 (管理数据库连接的"总管")
engine = create_async_engine(settings.DATABASE_URL, echo=settings.DEBUG)

# Step 2: 创建会话工厂 (每次调用, 产生一个新的数据库"工作台")
async_session = async_sessionmaker(engine, class_=AsyncSession,
expire_on_commit=False)

# Step 3: 依赖注入函数 (自动分配和回收工作台)
async def get_db() -> AsyncSession:
    async with async_session() as session:
        yield session
```

三个核心对象的关系



同步 vs 异步 (sync vs async)

```
# 同步方式: 一次只能做一件事, 必须等前一件做完
def cook_breakfast():
    boil_water()      # 烧水, 等 5 分钟
    fry_egg()         # 煎蛋, 等 3 分钟
    make_toast()      # 烤面包, 等 2 分钟
    # 总共: 10 分钟

# 异步方式: 可以同时做多件事
async def cook_breakfast():
    await boil_water()    # 开始烧水
    await fry_egg()       # 同时煎蛋 (不用等水烧完)
    await make_toast()    # 同时烤面包
    # 总共: 约 5 分钟 (取决于最慢的那个)
```

我们选择异步的原因：当一个请求在等待数据库返回结果时，服务器可以去处理其他请求，而不是傻等着。这大大提高了服务器的并发处理能力。

yield vs **return** 的区别

```
# 使用 return: 函数执行完就结束了
def get_value():
    value = create()
    return value      # 函数结束, 无法执行清理操作

# 使用 yield: 函数可以在"暂停点"恢复执行
def get_value():
    value = create()  # 1. 创建资源
    yield value       # 2. 暂停, 把 value 交出去使用
                      # 3. 使用完毕后, 从这里继续执行清理操作
```

在 `get_db()` 中:

- `yield session` 之前: 创建数据库会话
- `yield session`: 把会话交给路由函数使用
- 函数块结束后, `async with` 自动关闭会话 (清理)

应当熟练掌握的知识:

- Engine、Session 的概念和关系
- 理解 `async/await` 异步编程的基本思想
- 理解 `yield` 的暂停和恢复机制
- 理解依赖注入的好处 (自动资源管理)

3.3 `models/todo.py` — 数据库模型

```
class Base(DeclarativeBase):
    pass

class Todo(Base):
    __tablename__ = "todos"

    id: Mapped[int] = mapped_column(Integer, primary_key=True, autoincrement=True)
    title: Mapped[str] = mapped_column(String(200), nullable=False)
    description: Mapped[str | None] = mapped_column(Text, nullable=True,
default=None)
    is_completed: Mapped[bool] = mapped_column(Boolean, default=False,
nullable=False)
    created_at: Mapped[datetime] = mapped_column(DateTime(timezone=True),
server_default=func.now())
    updated_at: Mapped[datetime] = mapped_column(DateTime(timezone=True),
server_default=func.now(), onupdate=func.now())
```

与关系代数的对应关系

你在课上学过的关系模型:

关系模式: Todos(id, title, description, is_completed, created_at, updated_at)
 主键: {id}
 函数依赖: id → title, description, is_completed, created_at, updated_at

转换到 SQLAlchemy 模型, 对应关系:

关系代数概念	SQLAlchemy 代码	SQL 语句
关系名 Todos	<code>__tablename__ = "todos"</code>	<code>CREATE TABLE todos</code>
属性 id (域: 整数)	<code>id: Mapped[int] = mapped_column(Integer)</code>	<code>id INTEGER</code>
属性 title (域: 字符串)	<code>title: Mapped[str] = mapped_column(String(200))</code>	<code>title VARCHAR(200)</code>
主键约束	<code>primary_key=True</code>	<code>PRIMARY KEY (id)</code>
非空约束	<code>nullable=False</code>	<code>NOT NULL</code>
默认值	<code>default=False</code>	<code>DEFAULT FALSE</code>
外键 (未来会用到)	<code>ForeignKey("users.id")</code>	<code>FOREIGN KEY REFERENCES users(id)</code>

Mapped[str | None] 的含义

```
title: Mapped[str]          # 类型是 str, 不能为 None → 对应 NOT NULL
description: Mapped[str | None] # 类型是 str 或 None → 对应 NULLABLE
```

这是 Python 的**类型提示 (Type Hint)**, 它告诉 IDE 和其他开发者:

- `title` 永远是字符串, 不会是 None
- `description` 可能是字符串, 也可能是 None

server_default vs default 的区别

```
# default: 由 Python/SQLAlchemy 在应用层设置默认值
is_completed: ... = mapped_column(default=False)
# → SQLAlchemy 在创建对象时自动设置 is_completed=False

# server_default: 由数据库服务器设置默认值
created_at: ... = mapped_column(server_default=func.now())
# → 数据库在 INSERT 时自动填入当前时间
# → 好处: 即使不通过你的应用写入数据, 默认值也会生效
```

__repr__ 方法的作用

```
def __repr__(self) -> str:
    return f"<Todo(id={self.id}, title='{self.title}', is_completed={self.is_completed})>"
```

当你在调试时打印一个 Todo 对象：

```
# 没有 __repr__:
print(todo) # <app.models.todo.Todo object at 0x7f8b8b8b8b80> ← 看不出是啥

# 有了 __repr__:
print(todo) # <Todo(id=1, title='买菜', is_completed=False)> ← 一目了然!
```

应当熟练掌握的知识：

- ❑ Python 类与数据库表的映射关系
- ❑ 常用的列类型： Integer, String, Text, Boolean, DateTime
- ❑ 约束条件： primary_key, nullable, default, unique
- ❑ 理解关系代数中的函数依赖与数据库表设计的关系
- ❑ Mapped 类型提示的作用
- ❑ server_default 与 default 的使用场景

3.4 schemas/todo.py — 数据验证

```
from pydantic import BaseModel, Field

class TodoCreate(BaseModel):
    title: str = Field(..., min_length=1, max_length=200)
    description: str | None = Field(default=None, max_length=2000)

class TodoUpdate(BaseModel):
    title: str | None = Field(default=None, min_length=1, max_length=200)
    description: str | None = Field(default=None, max_length=2000)
    is_completed: bool | None = Field(default=None)

class TodoResponse(BaseModel):
    id: int
    title: str
    description: str | None
    is_completed: bool
    created_at: datetime
    updated_at: datetime

    model_config = {"from_attributes": True}
```

为什么数据验证如此重要？

想象你的 API 没有任何验证：

```
# 用户发送了这样的请求:
POST /todos
{"title": ""}          # 空标题 → 数据库里会有一条没有标题的 Todo!
{"title": 123}           # 标题是数字 → 可能导致显示错误
{"title": "a" * 10000}   # 超长标题 → 可能撑爆数据库
{}                      # 没有标题 → 应用直接报错崩溃
```

Pydantic 帮你在数据进入应用之前，就把这些非法数据拦截：

```
用户请求 → Pydantic 验证 → 验证通过 → 进入你的代码
|           ↓
|→ 验证失败 → 自动返回 422 错误（告诉用户哪里不对）
```

Field(...) 中 ... 的含义

```
title: str = Field(...)          # ... 表示"必填" (Ellipsis 对象)
description: str | None = Field(default=None)  # default=None 表示"可选，默认为空"
```

... 是 Python 的一个特殊常量 Ellipsis，在 Pydantic 中约定用来表示“此字段没有默认值，必须提供”。

exclude_unset=True 的精妙之处

这是 TodoUpdate 的关键设计：

```
# 用户发送: {"is_completed": true}
# 只想标记完成，不改其他字段

update_data = todo_in.model_dump(exclude_unset=True)
# 结果: {"is_completed": True}
# 注意: title 和 description 虽然默认值是 None，但它们没被"设置过"
# 所以不会出现在结果中，不会被误更新为 None

# 如果用 model_dump() (不加 exclude_unset) :
update_data = todo_in.model_dump()
# 结果: {"title": None, "description": None, "is_completed": True}
# 危险! title 和 description 会被更新为 None!
```

model_config = {"from_attributes": True} 的作用

```
# SQLAlchemy 的 Todo 对象长这样:
todo = Todo()
todo.id = 1
```

```

todo.title = "买菜"

# Pydantic 默认期望从字典读取数据:
TodoResponse(**{"id": 1, "title": "买菜"}) # ✅ 正常

# 但如果我们想从 SQLAlchemy 对象读取:
TodoResponse.model_validate(todo) # ❌ 没有 from_attributes 会报错

# 加了 from_attributes = True 后:
TodoResponse.model_validate(todo) # ✅ Pydantic 会自动从 todo.id, todo.title 读取

```

应当熟练掌握的知识:

- 理解为什么需要多个 Schema (Create/Update/Response)
- Field() 的常用参数: min_length, max_length, ge(≥), le(≤), examples
- exclude_unset=True 的工作原理和使用场景
- Pydantic 自动验证和错误响应的机制
- model_config 的常用配置

3.5 routers/todo.py — API 路由

五个端点的完整对照表

端点	HTTP 方法	URL	功能	状态码	请求体	返回值
创建	POST	/todos/	创建新 Todo	201	TodoCreate	TodoResponse
列表	GET	/todos/	获取所有 Todo	200	无	List[TodoResponse]
详情	GET	/todos/{id}	获取单个 Todo	200	无	TodoResponse
更新	PATCH	/todos/{id}	更新 Todo	200	TodoUpdate	TodoResponse
删除	DELETE	/todos/{id}	删除 Todo	204	无	无

依赖注入 (Dependency Injection) 详解

```

@router.post("/")
async def create_todo(
    todo_in: TodoCreate,                      # 参数 1: 自动从请求体解析
    db: AsyncSession = Depends(get_db),          # 参数 2: 自动注入数据库会话
) -> Todo:

```

FastAPI 看到函数参数时的推理过程:

参数 todo_in: TodoCreate
→ TodoCreate 是 Pydantic 模型
→ 从请求体 (Body) 中读取 JSON 并验证
→ 如果验证失败, 自动返回 422 错误

参数 db: AsyncSession = Depends(get_db)
→ 有 Depends() 装饰器
→ 调用 get_db() 获取数据库会话
→ 把会话传给 db 参数
→ 函数执行完毕后，自动清理会话

你完全不需要手动管理这些——FastAPI 帮你自动处理。这就是依赖注入的威力。

路径参数 (Path Parameters)

```
@router.get("/{todo_id}")
async def get_todo(todo_id: int):
    # todo_id 自动从 URL 中提取
    # GET /todos/42 → todo_id = 42
```

FastAPI 看到 `{todo_id}` 和函数参数 `todo_id: int`:

1. 从 URL 中提取 `todo_id` 的值
2. 自动转换为 `int` 类型
3. 如果转换失败（比如 `/todos/abc`），自动返回 422 错误

查询参数 (Query Parameters)

```
@router.get("/")
async def list.todos(skip: int = 0, limit: int = 20):
    # skip 和 limit 从 URL 查询字符串中提取
    # GET /todos/?skip=10&limit=5 → skip=10, limit=5
    # GET /todos/ → skip=0, limit=20 (使用默认值)
```

HTTPException — 标准化的错误处理

```
if todo is None:
    raise HTTPException(
        status_code=status.HTTP_404_NOT_FOUND,
        detail=f"Todo with id {todo_id} not found",
    )
```

`raise HTTPException` 就像按下“紧急按钮”：

1. 立即停止当前函数的执行
2. FastAPI 捕获这个异常
3. 自动构造 JSON 错误响应：

```
{
    "detail": "Todo with id 999 not found"
}
```

4. 返回 404 状态码

应当熟练掌握的知识：

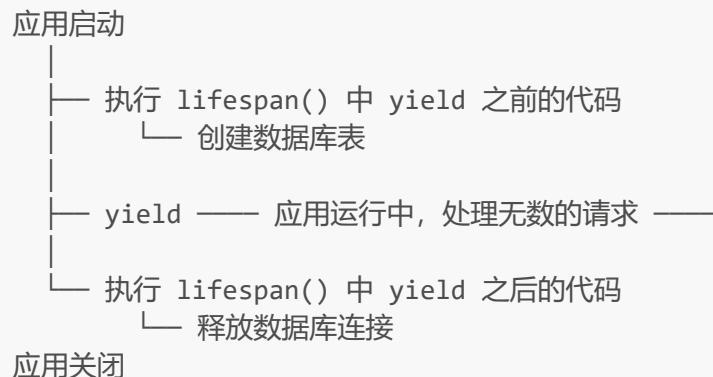
- HTTP 方法的语义 (GET=获取, POST=创建, PATCH=部分更新, DELETE=删除)
- HTTP 状态码的含义 (200, 201, 204, 404, 422)
- FastAPI 参数提取的三种方式：路径参数、查询参数、请求体
- 依赖注入的概念和好处
- 标准的错误处理模式

3.6 main.py — 应用入口

```
@asynccontextmanager
async def lifespan(app: FastAPI):
    # 启动时：创建数据库表
    async with engine.begin() as conn:
        await conn.run_sync(Base.metadata.create_all)
    yield
    # 关闭时：释放连接
    await engine.dispose()

app = FastAPI(title=settings.APP_NAME, ...)
app.include_router(todo_router.router)
```

Lifespan (生命周期) 管理



Base.metadata.create_all 做了什么？

还记得 models/todo.py 中 class Todo(Base) 吗？

Base 类会自动“记忆”所有继承它的模型类：

Base 认识的模型: [Todo]
Base.metadata 包含的表信息: [todos 表的所有列定义]

`create_all` 会检查数据库中是否存在这些表:

- 如果不存在 → 执行 `CREATE TABLE`
- 如果已存在 → 跳过 (不会重复创建)

app.include_router() — 模块化路由注册

```
app.include_router(todo_router.router)
# → 把 todo_router 里所有的路由 (/todos/...) 注册到主应用

# 将来添加更多功能:
# app.include_router(user_router.router)      # /users/...
# app.include_router(auth_router.router)       # /auth/...
```

应当熟练掌握的知识:

- 理解应用的生命周期 (启动 → 运行 → 关闭)
- `asynccontextmanager` 和 `yield` 的协作方式
- 路由注册的模块化组织方式
- 健康检查端点的意义和实现

四、API 端点详解

完整的请求和响应示例

1. 创建 Todo

```
POST /todos/
Content-Type: application/json

{
    "title": "学习 Python",
    "description": "完成 FastAPI 教程的第一章"
}
```

成功响应 (201 Created) :

```
{
    "id": 1,
    "title": "学习 Python",
    "description": "完成 FastAPI 教程的第一章",
```

```
"is_completed": false,  
"created_at": "2026-02-14T19:36:06",  
"updated_at": "2026-02-14T19:36:06"  
}
```

验证失败时 (422 Unprocessable Entity) :

```
{  
    "detail": [  
        {  
            "type": "string_too_short",  
            "loc": ["body", "title"],  
            "msg": "String should have at least 1 character",  
            "input": "",  
            "ctx": {"min_length": 1}  
        }  
    ]  
}
```

2. 获取列表

```
GET /todos/?skip=0&limit=10
```

成功响应 (200 OK) :

```
[  
    {  
        "id": 1,  
        "title": "学习 Python",  
        "description": "完成 FastAPI 教程的第一章",  
        "is_completed": false,  
        "created_at": "2026-02-14T19:36:06",  
        "updated_at": "2026-02-14T19:36:06"  
    }  
]
```

3. 获取单个

```
GET /todos/1
```

成功响应：同上面的单个对象。

不存在时 (404 Not Found) :

```
{  
    "detail": "Todo with id 999 not found"  
}
```

4. 部分更新

```
PATCH /todos/1  
Content-Type: application/json
```

```
{  
    "is_completed": true  
}
```

成功响应 (200 OK) :

```
{  
    "id": 1,  
    "title": "学习 Python",  
    "description": "完成 FastAPI 教程的第一章",  
    "is_completed": true,  
    "created_at": "2026-02-14T19:36:06",  
    "updated_at": "2026-02-14T19:38:00"  
}
```

注意 `updated_at` 自动更新了！

5. 删除

```
DELETE /todos/1
```

成功响应: 204 No Content (没有响应体)

五、数据流完整走读

以 "创建一个 Todo" 为例, 追踪数据从用户请求到数据库存储的全过程:

用户发送 HTTP 请求
|
| POST /todos/
| Body: {"title": "买菜", "description": "去超市"}
|
|

```
[uvicorn] 接收 TCP 连接, 解析 HTTP 协议
|
▼
[FastAPI 路由匹配] POST /todos/ → 匹配到 create_todo 函数
|
▼
[FastAPI 参数解析]
    └─ 发现参数 todo_in: TodoCreate → 从请求体解析 JSON
        └─ Pydantic 验证: title 非空? ≤200字符? 
    └─ 发现参数 db = Depends(get_db) → 调用 get_db()
        └─ 创建数据库会话 session
|
▼
[create_todo 函数执行]


```
todo = Todo(**todo_in.model_dump())
等价于 Todo(title="买菜", description="去超市")

db.add(todo)
告诉 Session: "把这个对象加入待写入队列"

await db.commit()
Session 生成 SQL: INSERT INTO todos (title, description, ...) VALUES (...)

发送给数据库执行

await db.refresh(todo)
Session 生成 SQL: SELECT * FROM todos WHERE id=1
读回数据库生成的 id、created_at、updated_at

return todo
返回 Todo 对象
```


|
▼
[FastAPI 响应序列化]
    response_model=TodoResponse
    将 Todo ORM 对象 → 按 TodoResponse 格式转为 JSON
|
▼
[FastAPI 设置状态码] 201 Created
|
▼
[uvicorn] 发送 HTTP 响应
|
▼
[get_db 清理] async with 自动关闭数据库会话
|
▼
用户收到 JSON 响应 
```

六、关键概念深入理解

6.1 异步编程 (async/await)

为什么我们要用 `async def` 而不是普通的 `def`?

```
# 同步服务器 (一次只能处理一个请求) :  
# 用户 A 请求 → 查询数据库 (等 100ms) → 返回  
# 用户 B 请求 → 等用户 A 完成 → 查询数据库 (100ms) → 返回  
# 用户 C 请求 → 等 A 和 B 完成 → ...  
# 总耗时: 300ms  
  
# 异步服务器 (可以同时处理多个请求) :  
# 用户 A 请求 → 查询数据库 (开始等待)  
# 用户 B 请求 → 查询数据库 (开始等待) ← A 等待时就开始处理 B  
# 用户 C 请求 → 查询数据库 (开始等待) ← 同时处理 C  
# 当数据库返回时, 依次回来处理  
# 总耗时: 约 100ms
```

`async/await` 就是 Python 实现异步编程的方式:

- `async def` 声明 "这个函数可以在等待时让出控制权"
- `await` 表示 "这里要等待一个耗时操作, 在等待期间可以去做别的"

6.2 CRUD — Web 开发的四种基本操作

CRUD 是几乎所有 Web 应用的基础:

```
Create (创建) → 用户注册、发帖、写评论  
Read (读取) → 查看个人资料、浏览列表、搜索  
Update (更新) → 编辑资料、修改文章  
Delete (删除) → 删除帖子、注销账号
```

几乎所有的 Web 应用 (微博、淘宝、知乎...) 本质上都是对某些资源做 CRUD。掌握了 CRUD, 你就掌握了 Web 开发的核心模式。

6.3 HTTP 状态码分类

```
1xx → 信息性 (很少直接使用)  
2xx → 成功  
    200 OK          → 一般性成功  
    201 Created     → 创建成功  
    204 No Content → 成功但无内容返回 (用于删除)  
3xx → 重定向 (URL 跳转)  
4xx → 客户端错误 (用户的问题)  
    400 Bad Request   → 请求格式不对  
    401 Unauthorized  → 未登录  
    403 Forbidden      → 没有权限  
    404 Not Found      → 资源不存在  
    422 Unprocessable  → 数据验证失败
```

5xx → 服务器错误（你的程序出 Bug 了）
500 Internal Error → 服务器内部错误

💡 **专业建议：**返回正确的状态码是 API 设计的基本素养。不要所有情况都返回 200，然后在响应体里用 "success": false 来表示失败。

七、专业开发者的思维清单

开始一个新项目时应该思考的

1. 技术选型

- 选择什么框架？为什么？
- 选择什么数据库？开发阶段和生产阶段用什么？
- 需要哪些第三方库？

2. 项目结构

- 代码怎么组织？
- 哪些东西需要分离？（配置、模型、路由、测试）
- 目录结构是否支持未来的扩展？

3. 数据模型设计

- 需要哪些表？
- 每个表有哪些字段？类型和约束是什么？
- 表之间的关系是什么？（一对多、多对多？）
- 是否包含审计字段？（created_at, updated_at）

4. API 设计

- 有哪些端点？URL 怎么设计？
- 每个端点的输入和输出是什么？
- 错误情况怎么处理？

5. 环境管理

- 配置怎么管理？（环境变量）
- 敏感信息怎么保护？（.env 不入 Git）
- 开发/测试/生产环境怎么区分？

写每一个函数时应该思考的

1. 这个函数的**单一职责**是什么？（只做一件事）
2. **输入**是什么类型？需要验证吗？
3. **输出**是什么类型？
4. **异常情况**怎么处理？（找不到数据、数据格式错误...）
5. **资源管理**：有没有需要清理的资源？（数据库连接、文件句柄）

交付代码前应该检查的

1. 代码能正常运行吗？
 2. 有没有写测试？测试通过了吗？
 3. 错误情况有没有处理？
 4. 注释是否充分？
 5. 有没有敏感信息硬编码在代码里？
 6. `.gitignore` 是否正确配置？
-

八、常见疑问解答

Q1：为什么有了 Model 还需要 Schema？不是重复吗？

不重复。它们的职责不同：

- **Model** 面向数据库：定义表结构、存储约束
- **Schema** 面向 API 用户：定义请求/响应格式、数据验证

而且同一个表对应多个 Schema（创建、更新、响应各一个），但只对应一个 Model。

Q2：什么时候用 `GET`，什么时候用 `POST`？

- **GET**：获取数据，不改变服务器状态（“看一看但不动”）
- **POST**：创建新资源，会改变服务器状态（“添加新东西”）
- **PATCH**：部分修改已有资源（“改一改”）
- **DELETE**：删除资源（“扔掉”）

Q3：`Depends(get_db)` 是怎么工作的？

FastAPI 看到 `Depends(get_db)` 时：

1. 在处理请求前，调用 `get_db()` 函数
2. 把 `yield` 出来的 Session 赋值给 `db` 参数
3. 执行你的路由函数
4. 路由函数结束后，继续执行 `get_db()` 中 `yield` 之后的清理代码

你只需要声明“我需要一个数据库会话”，FastAPI 帮你创建和回收。

Q4：为什么用 `PATCH` 而不是 `PUT`？

- **PUT**：需要发送完整对象（所有字段都必须提供）
- **PATCH**：只发送需要修改的字段

在实际开发中，PATCH 更加灵活和常用。用户通常不想每次都发送所有字段。

Q5：`async with` 是什么？

```
async with async_session() as session:  
    yield session
```

`async with` 是异步版本的上下文管理器，它保证：

- 进入时：自动执行初始化操作（创建 session）
- 退出时：自动执行清理操作（关闭 session）

无论中间是否发生异常，清理操作都会执行。这是 Python 的资源管理最佳实践。

Q6：应用是怎么知道要启动在 8000 端口的？

这是我们启动时指定的：

```
python -m uvicorn app.main:app --reload --port 8000
```

- `app.main` → 找到 `app/main.py` 文件
- `:app` → 找到文件中的 `app` 对象（FastAPI 实例）
- `--reload` → 文件修改时自动重启（开发模式）
- `--port 8000` → 监听 8000 端口

⌚ 下一步学习建议：

1. 在浏览器中打开 `http://127.0.0.1:8000/docs`，亲手测试每个 API 端点
2. 仔细阅读每个源代码文件中的注释
3. 尝试修改代码（比如给 Todo 加一个 `priority` 字段），看看需要改哪些地方
4. 回顾本文中的“思维清单”，尝试不看参考回答每个问题

学习编程最好的方式就是：**读代码 → 改代码 → 看效果 → 理解原理**