To Do List Project

(Phase 3 – WEB API)

PREPARED FOR

Software Engineering Course AUT

Date

Nov 2025

ToDoList – (WEB API)

مقدمه

در فازهای قبلی پروژه‌ی ToDoList، ابتدا ساختار برنامه با استفاده از **OOP** و **In-Memory Storage** طراحی شد (فاز ۱)، و سپس داده‌ها با استفاده از **پایگاه داده رابطه‌ای (PostgreSQL + SQLAlchemy)** پایدار شدند (فاز ۲).

اکنون در فاز سوم، قرار است این سیستم به یک **وب‌سرویس کامل (Web API)** تبدیل شود که از طریق اینترنت یا شبکه، بتوان به داده‌ها و عملیات سیستم دسترسی داشت.

به طور خلاصه در این مرحله از فریم‌ورک **FastAPI** استفاده می‌کنیم تا:

- منطق دامنه‌ی قبلی (Domain Logic) را بدون تغییر، در قالب Endpointهای RESTful ارائه کنیم،
- ساختار پروژه همچنان لایه‌ای (Layered Architecture) باقی بماند،
- اعتبارسنجی داده‌ها (Validation) و مستندسازی خودکار (Auto Documentation) به شکل حرفه‌ای انجام شود.
- و در نهایت API آماده‌ی اتصال به رابط‌های کاربری (Frontend یا Mobile App) گردد.

در ادامه توضیحات و منابعی برای درک پیش نیاز های اجرای این بخش پروژه ارائه می شود.

نکته: در ادامه درس با این موضوعات بیشتر و مفصل‌تر آشنا خواهید شد، این بخش به عنوان مقدمه برای آشنایی با ادبیات توسعه محصول آورده شده است.

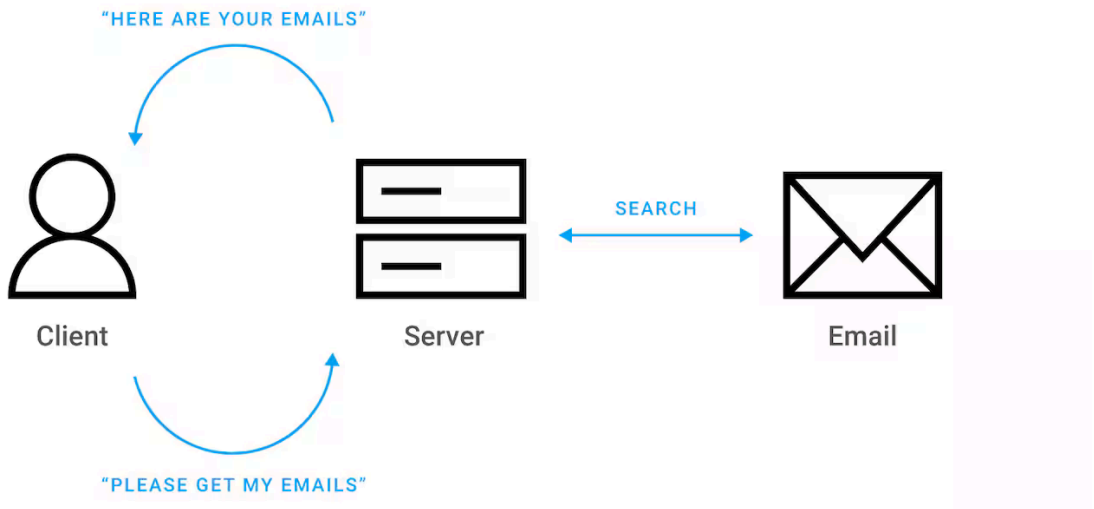
مقدمات و مفاهیم پایه:

پیش از شروع پیاده‌سازی API لازم است با مفاهیم پایه‌ی معماری وب و پروتکل HTTP آشنا شوید.

- **Client–Server**

در معماری Client–Server، کلاینت (مثلاً مرورگر یا اپ موبایل) درخواست (Request) می‌فرستد و سرور پاسخ (Response) برمی‌گرداند. این تفکیک باعث مقیاس‌پذیری و توسعه‌پذیری بالاتر سیستم می‌شود. به طور کلی نرم‌افزار تولید شده دو بخش دارد که یک بخش روی سرور اجرا می‌شود و مسئولیت اجرای منطق اصلی، محاسبات، ذخیره سازی و ... را دارد. بخش دوم که مربوط به کلاینت است وظیفه نمایش درست اطلاعات به کاربر را دارد و با سرور در ارتباط است تا از طریق درخواست-پاسخ اطلاعات مورد نیاز را ارسال و دریافت کند. در سمت کلاینت معمولا business logic اجرا نمی‌شود. گاهی این امر به این دلیل است که کلاینت یا نمی‌تواند همه داده های مورد نیاز را داشته باشد یا توانایی انجام عملیات ها را ندارد یا میخواهیم کار را به دلایل تجاری-امنیتی، در محلی خارج از دسترس کاربر انجام دهیم.

مدل کلی:کلاینت – سرور



سرور (Server): نرم‌افزاری که سرویس یا منبعی را فراهم می‌کند (مثلاً داده‌ها یا منطق تجاری). معمولاً روی یک ماشین (فیزیکی یا ابری) قدرتمند اجرا می‌شود و برای دریافت و پاسخ به درخواست‌ها منتظر می‌ماند.

کلاینت (Client): هر نرم‌افزاری که به سرور درخواست سرویس و اطلاعات می‌فرستد (مرورگر، اپ موبایل، اسکریپت، ابزار تست مانند curl/postman).

ارتباط: کلاینت درخواست (Request) می‌فرستد؛ سرور آن را پردازش و پاسخ (Response) برمی‌گرداند. معمولاً ارتباط روی پروتکل‌های شبکه‌ای مانند TCP/IP برقرار می‌شود و در سطح اپلیکیشن معمولاً HTTP/HTTPS استفاده می‌شود. سرور به خودی خود در امکان ارسال پاسخ بدون وجود درخواست را ندارد مگر در مدل های ارتباطی خاص که توضیح داده می‌شود.

- **Request / Response**

هر ارتباط HTTP شامل یک درخواست از سوی کلاینت و یک پاسخ از سوی سرور است.
درخواست معمولاً شامل اطلاعاتی مانند روش (Method)، مسیر (Path)، هدرها (Headers) و بدنه (Body) است. ارتباط حتما باید از طریق کلاینت شروع شود و سرور پاسخ دهد و برعکس این کار ممکن نیست.

- **پروتکل HTTP**

HTTP یا HyperText Transfer Protocol استاندارد اصلی ارتباط بین کلاینت و سرور در وب است. این پروتکل بر پایه‌ی متدها و کدهای وضعیت (Status Codes) تعریف می‌شود. این پروتکل روی لایه TCP/IP تعریف شده که یک پروتکل اساسی در سطح شبکه است و صورت کلی یک درخواست و پاسخ در آن به شکل زیر است.

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 123

<html>
  <body>
    <h1>Example Response</h1>
  </body>
</html>
```

```
GET /path/resource HTTP/1.1
Host: example.com
User-Agent: SomeClient/1.0
Accept: */*
```

Hypertext Transfer Protocol - HTTP

در ادامهٔ همین بحث، **HTTPS** هم یکی از مفاهیم پایه‌ای مهم در ارتباط با API و وب‌سرویس‌هاست.

HTTPS چیست و چرا مهم است؟

HTTPS نسخه‌ی امن‌تر **HTTP** است.

در HTTP داده‌ها به‌صورت عادی (Plain Text) بین کلاینت و سرور ردوبدل می‌شوند؛ یعنی اگر کسی وسط راه شبکه را شنود کند، می‌تواند محتوای درخواست و پاسخ را بخواند—مثلاً:

- نام کاربری و رمز عبور
- توکن
- داده‌های حساس
- مسیری که درخواست می‌رود

برای رفع این مشکل، HTTPS معرفی شد.

HTTPS چطور کار می‌کند؟

HTTPS از **TLS/SSL** استفاده می‌کند به طوری که هنگام جابجایی داده‌ها شرایط زیر محقق شود:

رمزنگاری (Encryption)

همهٔ داده‌ها قبل از ارسال رمز می‌شوند. یعنی اگر کسی بسته‌ها را بدزدد، نمی‌تواند محتوایش را بفهمد.

احراز هویت سرور (Authentication)

مرورگر یا برنامهٔ کلاینت مطمئن می‌شود که دارد با «سرور واقعی» حرف می‌زند، نه یک هکر وسط مسیر.

این کار با **SSL Certificate** انجام می‌شود (مثل گواهی Let’s Encrypt).

یکپارچگی داده (Integrity)

مطمئن می‌شویم داده‌ها در مسیر تغییر نکرده‌اند. اگر کسی بخواهد بسته‌ها را دست‌کاری کند، درخواست معتبر نخواهد بود.

HTTPS چه تأثیری روی API دارد؟

- توکن‌ها (JWT، Bearer Token) امن منتقل می‌شوند
- اطلاعات کاربران قابل شنود نیست
- کوکی‌ها با فلگ‌های امن (HttpOnly و Secure) قابل استفاده هستند
- مرورگر اجازهٔ ارسال برخی داده‌های حساس را فقط روی HTTPS می‌دهد
- خیلی از سرویس‌ها (Google، Stripe، OAuth) فقط روی HTTPS کار می‌کنند

مثال ساده

:HTTP

```
http://example.com/users?id=5
```

همه‌چیز قابل شنود است.

```
https://example.com/users?id=5
```

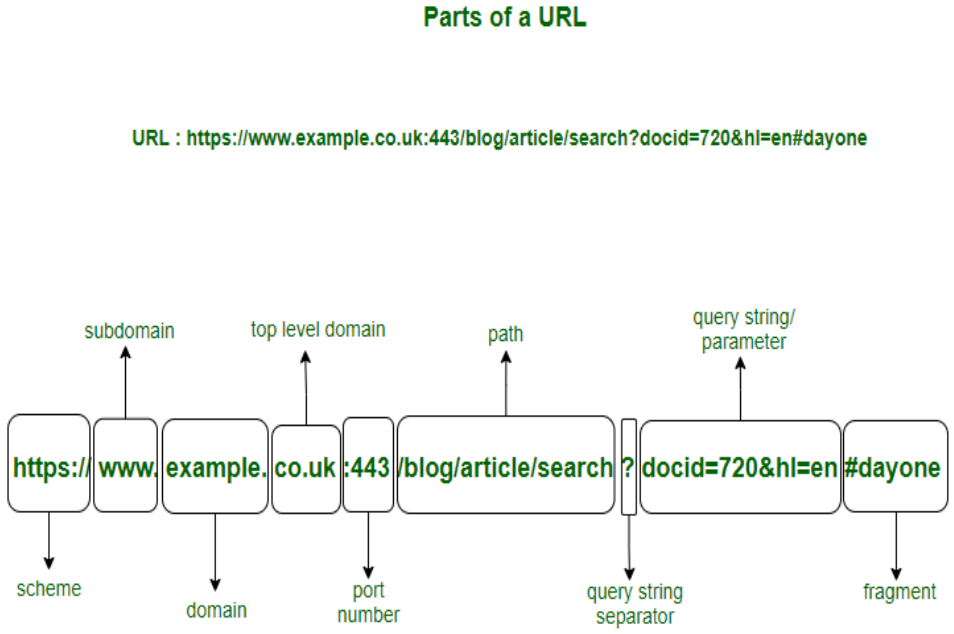
حتی اگر کسی ترافیک را ببیند، فقط بسته‌های رمزگذاری‌شده را دریافت می‌کند، نه محتوای واقعی.

چرا همه باید از HTTPS استفاده کنند؟

- امنیت اطلاعات
- اعتماد بیشتر
- الزام مرورگرها
- بهتر شدن SEO
- جلوگیری از حملات MITM (man-in-the-middle)

امروزه HTTPS به‌قدری استاندارد شده که تقریباً **بدون آن هیچ API جدی‌ای قابل قبول نیست.**

- **URL و URI**



URI و **URL** هر دو ابزار شناسایی منابع روی اینترنت هستند، اما کاربرد و محدوده‌شان کمی با هم متفاوت است. توضیح ساده و قابل‌فهمش اینه:

URI (Uniform Resource Identifier)

URI یک *شناسه* است؛ یعنی رشته‌ای که یک «منبع» را روی اینترنت **نام‌گذاری یا شناسایی** می‌کند. مسیر دسترسی به منبع را مشخص میکند.

URI می‌تواند فقط نام باشد، یا هم نام و هم آدرس.

مثال‌های URI:

- mailto:info@example.com
- ftp://example.com/file.txt
- urn:isbn:0451450523 → یک نام استاندارد بدون آدرس
- https://example.com/users/5

در واقع URL هم یک نوع URI است.

URL (Uniform Resource Locator)

URL زیرمجموعه‌ی URI است که علاوه بر شناسایی، **مسیر دسترسی** به منبع را هم مشخص می‌کند؛ یعنی:

- پروتکل (http, https, ftp…)
- دامنه
- مسیر
- پورت
- کوئری‌ها و …

به زبان ساده: **URL = URI که می‌گوید "چطور و از کجا به منبع برسیم".**

مثال :

```
mailto:someone@example.com
tel:+49123456789
ftp://ftp.example.com/files/
urn:isbn:9780131103627
```

URI

```
https://example.com
https://example.com/articles/123?sort=asc
http://192.168.1.1/login
ftp://ftp.example.org/pub/file.txt
```

URL

در واقع در درخواست های HTTP با استفاده از URL به نرم‌افزاری مثل مرورگر میگوییم با پروتکل HTTP کدام منبع را از سرور درخواست کند.

یک مقایسه‌ی کاملاً ساده

- **URI** = "نام یا شناسه‌ی یک چیز"
- **URL** = "آدرس کامل برای رسیدن به آن چیز"

هر URL یک URI است،

اما هر URI الزاماً URL نیست.

What is URL (Uniform Resource Locator)

URI Full Form

مثال واقعی برای فهم بهتر

فرض کن یک فایل داری به اسم:

urn:myfile:report2025

این فقط **URI** است — یک نام که چیزی را معرفی می‌کند، اما نمی‌گوید کجاست.

اما این:

https://storage.mysite.com/reports/2025.pdf

URL است، چون مکان و روش دسترسی را هم مشخص می‌کند.

- **HTTP Methods**
  - **GET** — واکشی منبع (بدون تغییر در سرور). باید ایمن و ایدمپوتنت باشد.
  - **POST** — ایجاد منبع / عملیات که معمولاً تغییر وضعیت سرور دارد.
  - **PUT** — جایگزینی کامل یک منبع (ایدمپوتنت).
  - **PATCH** — به‌روزرسانی جزئی یک منبع.
  - **DELETE** — حذف منبع.

سناریوی ما: تصور کنید در حال ساخت یک API برای مدیریت «پروفایل کاربران» در یک وب‌سایت هستید. هر کاربر یک «منبع» (Resource) محسوب می‌شود.

GET: فقط خواندن

- کاربرد اصلی: واکشی و خواندن اطلاعات.
- مثال: می‌خواهید پروفایل کاربری با شناسه (ID 123) را *ببینید*.
- چطور استفاده می‌شود؟ GET /users/123
- نکته کلیدی: GET نباید هیچ تغییری در سرور ایجاد کند (مثلاً نباید چیزی را حذف یا ویرایش کند). به همین دلیل به آن «ایمن» (Safe) می‌گویند. اگر ۱۰۰ بار هم آن را صدا بزنید، اتفاقی نمی‌افتد و فقط اطلاعات را می‌خوانید.

POST: ایجاد کردن موجودیت جدید

- کاربرد اصلی: ایجاد یک منبع جدید.
- مثال: یک کاربر جدید در سایت شما ثبت‌نام می‌کند. شما اطلاعات او (نام، ایمیل و...) را می‌گیرید و برای سرور می‌فرستید تا یک پروفایل جدید برای او بسازد.
- چطور استفاده می‌شود؟ POST /users (اطلاعات کاربر جدید در Body درخواست ارسال می‌شود).
- نکته کلیدی: POST ایدمپوتنت (Idempotent) نیست. یعنی اگر شما یک درخواست POST را ۵ بار تکرار کنید، (احتمالاً) ۵ کاربر جدید در سیستم ایجاد خواهید کرد.

استفاده غیرمعمول از POST: گاهی اوقات برای عملیاتی که در قالب‌های دیگر نمی‌گنجند (مثل "ارسال ایمیل" یا "محاسبه یک مقدار") هم از POST استفاده می‌شود، چون این عملیات وضعیت سرور را تغییر می‌دهند (یا حداقل ایدمپوتنت نیستند).

PUT: جایگزینی کامل

- کاربرد اصلی: جایگزینی کامل یک منبع موجود.
- مثال: کاربر 123 می‌خواهد *کل* اطلاعات پروفایل خود را ویرایش کند. شما باید *تمام* اطلاعات او را (حتی آن‌هایی که تغییر نکرده‌اند) در Body درخواست بفرستید.
- چطور استفاده می‌شود؟ PUT /users/123
- نکته کلیدی: PUT ایدمپوتنت (Idempotent) است. یعنی اگر شما ۵ بار هم اطلاعات *یکسان* را برای کاربر 123 PUT کنید، نتیجه نهایی با ۱ بار PUT کردن فرقی ندارد.
- خطر: اگر شما فقط «نام» کاربر را در درخواست PUT بفرستید و «شماره تلفن» را نفرستید، سرور فرض می‌کند که می‌خواهید شماره تلفن حذف شود (یا null شود). چون PUT یعنی: «کل این منبع را با این چیزی که من می‌فرستم جایگزین کن.»

PATCH: به‌روزرسانی جزئی (وصله زدن)

- کاربرد اصلی: به‌روزرسانی جزئی یا تغییر بخشی از یک منبع.
- مثال: کاربر 123 می‌خواهد *فقط* شماره تلفن خود را تغییر دهد و به نام و ایمیل خود دست نمی‌زند.
- چطور استفاده می‌شود؟ PATCH /users/123 (و در Body فقط شماره تلفن جدید را می‌فرستید، مثلاً: {"...phone": "0912"})
- نکته کلیدی: این متود بسیار بهینه‌تر از PUT است وقتی فقط قصد تغییر یک یا دو فیلد را دارید. سرور فقط فیلدهایی که شما فرستاده‌اید را به‌روز می‌کند و به بقیه کاری ندارد.

DELETE: حذف کردن

- کاربرد اصلی: حذف یک منبع مشخص.
- مثال: کاربر 123 می‌خواهد حساب کاربری خود را *پاک کند*.
- چطور استفاده می‌شود؟ DELETE /users/123

- نکته کلیدی: DELETE هم ایدمپوتنت است. اگر ۵ بار درخواست حذف کاربر 123 را بفرستید، بار اول حذف می‌شود و در ۴ بار بعدی سرور احتمالاً خطای 404 (Not Found) برمی‌گرداند، اما *وضعیت نهایی* سیستم (یعنی نبودن کاربر 123) یکسان است.

جمع‌بندی و رفع ابهام اصلی: POST در مقابل PUT در مقابل PATCH

این سه اغلب با هم قاطی می‌شوند. اینطور به خاطر بسپارید:

1. آیا می‌خواهم چیز *جدیدی* بسازم؟
   - بله استفاده از POST (مثلاً POST /users برای ساخت کاربر جدید).
2. آیا می‌خواهم چیزی که *از قبل وجود دارد* را تغییر دهم؟ (مثلاً کاربر 123)
   - آیا می‌خواهم *کل* اطلاعاتش را جایگزین کنم؟ (مثلاً یک فرم ویرایش کامل پروفایل را پُر کرده):
     - بله استفاده از PUT (مثلاً PUT /users/123 با ارسال *کل* اطلاعات پروفایل).
   - آیا می‌خواهم *فقط یک بخش کوچک* (مثلاً فقط آواتار یا فقط رمز عبور) را تغییر دهم؟
     - بله استفاده از PATCH (مثلاً PATCH /users/123 با ارسال *فقط* فیلد رمز عبور).

📖 جدول تقلب (Cheat Sheet)

ایدمپوتنت؟ (تکرارپذیر)	مثال URL	چه زمانی استفاده شود؟	متود
بله	users/123/	برای خواندن اطلاعات	GET
خیر (چندین منبع ایجاد می‌کند)	users/	برای ایجاد منبع جدید	POST

PUT	برای جایگزینی کامل منبع	users/123/	بله (نتیجه نهایی یکسان است)
PATCH	برای ویرایش جزئی منبع	users/123/	خیر (معمولاً اینطور در نظر گرفته می‌شود)
DELETE	برای حذف منبع	users/123/	✅ بله (نتیجه نهایی یکسان است)

- **Header, Body, Query Params, Path Params**

در یک درخواست HTTP (مثل وقتی مرورگر یا اپ با سرور حرف می‌زند)، اطلاعات می‌تواند در بخش‌های مختلفی قرار بگیرد. چهار بخش مهم و رایج عبارتند از: **Header**, **Body**, **Query Params**, **Path Params**.

در ادامه هر کدام را ساده، با مثال و با جزئیات کافی توضیح می‌دهم.

---

1) Header — اطلاعات جانبی و تنظیمات درخواست

هدرها مثل «برگه‌های اطلاعاتی» هستند که همراه درخواست ارسال می‌شوند.

توی هدرها معمولاً **اطلاعاتی که داده اصلی نیستند** قرار می‌گیرد؛ مثلا:

- نوع محتوایی که می‌فرستیم: Content-Type: application/json
- توکن احراز هویت: Authorization: Bearer <token>
- زبان مورد ترجیح

- نوع مرورگر
- اندازه محتوا و ...

ویژگی مهم:

Header هیچ داده اصلی مثل "نام کاربر" یا "قیمت محصول" را حمل نمی‌کند؛ فقط **اطلاعات فرعی مخصوص پردازش درخواست** است.

```
GET /users HTTP/1.1
Host: example.com
Authorization: Bearer 123abc
Accept: application/json
```

2) Body – محتوای اصلی درخواست

Body جایی است که داده اصلی درخواست داخل آن قرار می‌گیرد؛ معمولاً در درخواست‌هایی مثل POST و PUT.

وقتی می‌خواهی یک کاربر بسازی، متن یک پیام بفرستی، یا اطلاعاتی آپلود کنی، این داده‌ها در Body قرار می‌گیرند.

مثال: ایجاد کاربر جدید

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

{
  "name": "Ali",
  "email": "ali@example.com"
}
```

نکته:

در درخواست GET معمولاً Body نداریم.

---

3) Query Params – پارامترهای اختیاری برای فیلتر، جستجو، صفحه‌بندی

Query Params بعد از علامت ? در URL قرار می‌گیرند.

این‌ها برای پرس‌وجو کردن، فیلتر کردن، مرتب‌سازی و صفحه‌بندی استفاده می‌شوند.

ویژگی‌ها:

- **اختیاری‌اند**
- **برای تغییر رفتار یک endpoint بدون تغییر مسیر**
- **خواندن و ساختنشان آسان است**

مثال: پیدا کردن کاربران صفحه ۲، مرتب‌شده بر اساس نام

```
GET /users?page=2&sort=name
```

مثال: جستجو

```
GET /products?search=phone
```

4) Path Params – بخش‌های متغیر در مسیر که یک منبع را مشخص می‌کنند

Path Params داخل مسیر قرار می‌گیرند و معمولاً بخشی از شناسه‌ی یک منبع هستند.

یعنی برای دسترسی به یک چیز مشخص استفاده می‌شوند:

- کاربر شماره ۱۲
- سفارش شماره ۵۵
- مقاله با slug خاص

ویژگی‌ها:

- اجباری‌اند
- نشان‌دهنده‌ی یک «منبع خاص» هستند
- بخشی از مسیر URL هستند نه بعد از ?

مثال: گرفتن اطلاعات کاربر شماره 12

```
GET /users/12
```

اینجا 12 یک Path Param است.

مثال: گرفتن سفارش شماره 55

```
GET /orders/55
```

مقایسه‌ی ساده و قابل‌فهم

- **Header:** اطلاعات جانبی و کنترلی → مثل کارت شناسایی
- **Body:** داده اصلی → مثل محتوای پیام
- **Query Params:** تنظیمات و فیلترها → مثل گزینه‌های جستجو
- **Path Params:** شناسه‌ی دقیق یک منبع → مثل شماره پرونده

- **Statelessness**

در سیستم REST، هر درخواست مستقل از دیگری است و وضعیت طولانی‌مدت در سمت سرور نگه‌داری نمی‌شود. یعنی هر درخواست باید تمام اطلاعات و جزئیات مورد نیاز برای ارضای درخواست در خود داشته باشد و نمی‌توان به درخواست های قبلی اتکا کرد. این خاصیت از خود پروتکل http منجر میشود و در سبک REST تاکید بسیاری بر آن می‌شود. برای درک بهتر به این لینک مراجعه کنید.

- **RESTful API**

REST که مخفف Representational State Transfer است، سبکی از طراحی API است که از منابع (Resources) و عملیات استاندارد HTTP استفاده می‌کند.

- **Resource و Endpoint**

Resource در یک API به موجودیتی گفته می‌شود که سرور آن را مدیریت می‌کند. این موجودیت می‌تواند واقعی یا مفهومی باشد؛ مثل *User، Product، Order، Comment*. هر Resource معمولاً یک ساختار داده دارد و

مجموعه‌ای از عملیات روی آن قابل انجام است. مثلاً Resource مربوط به «کاربر» می‌تواند شامل فیلدهایی مثل id, name, email باشد و سرور مسئول ذخیره، ویرایش و بازیابی این داده‌هاست.

Endpoint مسیر مشخصی در سرور است که از طریق آن می‌توان به یک Resource دسترسی پیدا کرد یا عملیاتی روی آن انجام داد. Endpoint یک URL است که معمولاً همراه با یک متد HTTP (مثل GET یا POST) معنا پیدا می‌کند. کلاینت از طریق endpoint درخواست می‌فرستد و سرور پاسخ می‌دهد.

مثال ساده:

Resource = «کاربر»

Endpointهای مربوط به آن:

- **GET /users** → گرفتن لیست کاربران
- **GET /users/10** → گرفتن اطلاعات کاربر با شناسه ۱۰
- **POST /users** → ساخت کاربر جدید
- **PUT /users/10** → ویرایش کاربر شماره ۱۰
- **DELETE /users/10** → حذف کاربر

در اینجا **Resource همان "users"** است که یک موجودیت داده‌ای است،

و **Endpointها مسیرهایی هستند** که اجازه می‌دهند روی این Resource عملیات انجام دهیم.

یک مثال دیگر:

Resource = «سفارش»

Endpointها:

- **GET /orders**
- **POST /orders**
- **GET /orders/45**

- **PATCH /orders/45/status**

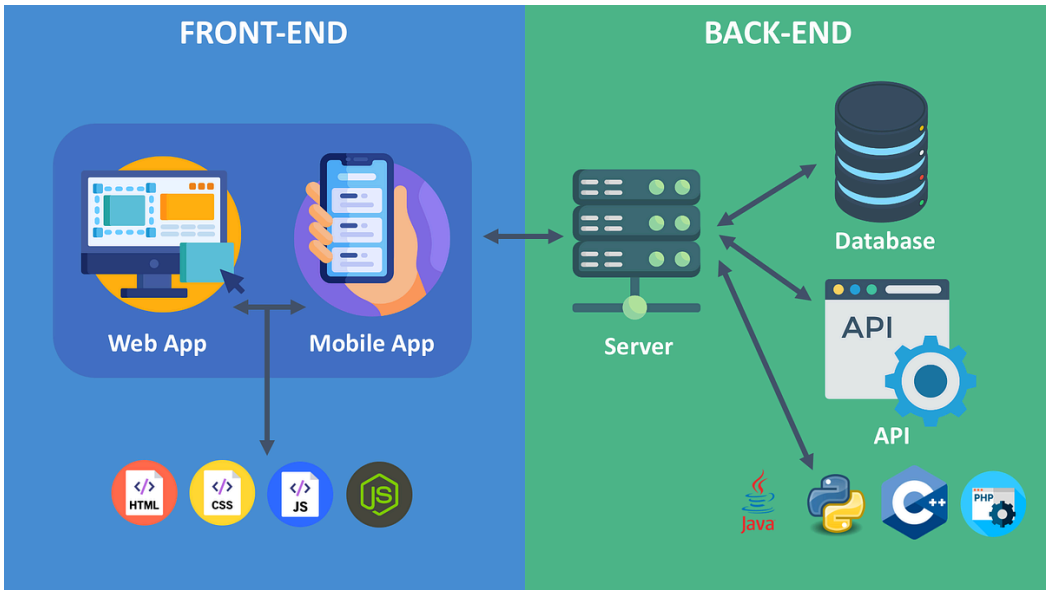
Resource «سفارش» فقط یک مفهوم داده‌ای است، اما endpointها روش‌های مختلف دسترسی و تغییر آن را مشخص می‌کنند.

درنتیجه:

Resource = داده و مفهوم پشت API

Endpoint = مسیر عملیاتی برای دسترسی به آن داده‌ها یا تغییرشان.

- **Frontend و Backend**



Backend

بخش پشت‌صحنه‌ی یک نرم‌افزار یا وب‌اپلیکیشن است که منطق، داده، و ارتباط با پایگاه داده را مدیریت می‌کند. کاربر مستقیماً آن را نمی‌بیند.

وظایف اصلی:

- پردازش درخواست‌های کاربر (از طریق API یا فرم‌ها)
- ارتباط با پایگاه داده (ذخیره، خواندن، و ویرایش داده)
- پیاده‌سازی منطق تجاری (Business Logic)
- مدیریت امنیت، احراز هویت، و مجوزها
- ارسال پاسخ به Frontend

Frontend

بخش قابل‌مشاهده و تعاملی برای کاربر است – یعنی ظاهر سایت یا اپ. معمولاً با **HTML، CSS،** و **JavaScript** ساخته می‌شود.

تفاوت Backend و Frontend:

Frontend بخشی از نرم‌افزار است که کاربر آن را می‌بیند و با آن تعامل دارد. ظاهر سایت، دکمه‌ها، فرم‌ها، و تمام عناصر گرافیکی در این بخش ساخته می‌شوند. تکنولوژی‌های رایج آن شامل **HTML، CSS** و **JavaScript** و فریم‌ورک‌هایی مانند **Vue، React** یا **Blazor** هستند. هدف اصلی فرانت‌اند ایجاد تجربه کاربری زیبا، روان و قابل‌فهم است.

در مقابل، **Backend** بخش پنهان سیستم است که پشت صحنه کار می‌کند. این بخش مسئول دریافت و پردازش درخواست‌ها، مدیریت پایگاه داده، اجرای منطق تجاری (Business Logic)، و ارسال پاسخ مناسب به فرانت‌اند است. معمولاً با زبان‌هایی مانند **Java، Go، Python، C#** یا **Node.js** توسعه داده می‌شود و روی سرور اجرا می‌گردد.

به طور خلاصه، **Frontend** با ظاهر و تعامل کاربر سر و کار دارد، در حالی که **Backend** زیرساخت و منطق پشت آن را فراهم می‌کند. فرانت‌اند چیزی است که کاربر مستقیماً تجربه می‌کند، اما بک‌اند همان بخشی است که باعث می‌شود همه چیز درست و امن کار کند.

Frontend = چیزی که کاربر می‌بیند

Backend = چیزی که کاربر نمی‌بیند ولی همه‌چیز را ممکن می‌کند

- ## چند نوع از پروتکل ارتباطی

REST

مجموعه‌ای از اصول معماری برای طراحی وب‌سرویس‌ها است

- **منابع (Resources):** هر چیز قابل‌نام‌گذاری (کاربر، سفارش) یک منبع است.
- **شناسه منابع (URIs):** هر منبع باید با یک URI یکتا قابل دسترسی باشد.
- **نمایش (Representation):** منبع می‌تواند اشکال مختلف (JSON, XML) داشته باشد؛ معمولاً JSON استفاده می‌کنیم.
- **روش‌ها (HTTP verbs):** عملیات روی منابع باید با متدهای HTTP هم‌خوانی داشته باشد.
- **بدون حالت بودن سرویس (Statelessness):** هر درخواست باید شامل تمام اطلاعات لازم باشد (سرور نباید state از کلاینت بین درخواست‌ها نگه دارد؛ مگر در موارد auth token/session که client آن را ارسال کند).
- **کَشینگ:** پاسخ‌ها باید امکان کش شدن را داشته باشند تا عملکرد بهتر شود.

یادآوری: REST یک قرارداد سخت‌گیرانه نیست؛ بلکه مجموعه‌ای از بهترین تمرین‌هاست. هدف خوانایی، پیش‌بینی‌پذیری و سازگاری است. در این بخش ما از REST استفاده خواهیم کرد و پروتکل‌های بعدی محض آشنایی معرفی می‌شوند.

GraphQL: زبان پرس‌وجوی انعطاف‌پذیر.

GraphQL زبان query برای APIها است که به کلاینت اجازه می‌دهد دقیقاً داده‌ی موردنیازش را از سرور درخواست کند.

مفاهیم اصلی:

- **Schema (طرح‌واره):** ساختار و نوع داده‌هایی را که سرور ارائه می‌دهد تعریف می‌کند (نوع‌ها، فیلدها، روابط).
- **Query:** درخواست خواندن داده است؛ کلاینت مشخص می‌کند دقیقاً چه فیلدهایی از چه نوعی را می‌خواهد.
- **Mutation:** برای عملیات تغییر داده (ایجاد، ویرایش، حذف) استفاده می‌شود.
- **Subscription:** برای دریافت داده‌های بلادرنگ از سرور.
- **Resolver:** تابعی در سرور که مشخص می‌کند هر فیلد چگونه باید مقداردهی شود.

ویژگی‌ها:

- **انعطاف‌پذیری:** کلاینت فقط همان داده‌ای را می‌گیرد که نیاز دارد (نه بیشتر، نه کمتر).
- **تک‌نقطه‌ای بودن:** معمولاً کل API از طریق یک endpoint (مثلاً /graphql) در دسترس است.
- **عدم وابستگی به HTTP methods:** برخلاف REST، همه‌ی عملیات (خواندن، نوشتن) از طریق POST انجام می‌شود.
- **ساختار پاسخ:** پاسخ دقیقاً ساختاری مشابه query دارد.
- **بدون حالت بودن:** مانند GraphQL ،REST هم معمولاً stateless است؛ هر درخواست مستقل و خودکفاست.
- **کَشینگ:** به‌صورت ذاتی ندارد و معمولاً توسط لایه‌های بالاتر (کلاینت یا gateway) انجام می‌شود.

🔹 **یادآوری:** GraphQL جایگزین REST نیست، بلکه رویکردی متفاوت برای تعامل داده بین کلاینت و سرور است؛ هدف آن کاهش over-fetching و under-fetching و افزایش کارایی ارتباط است.

gRPC: ارتباط سریع‌تر بر پایه‌ی باینری و پروتکل‌های RPC.

gRPC فریم‌ورکی برای ارتباط بین سرویس‌ها است که توسط Google توسعه یافته و بر پایه‌ی پروتکل **HTTP/2** و قالب داده‌ی **Protocol Buffers (Protobuf)** کار می‌کند.

مفاهیم اصلی:

- **Service (سرویس):** مجموعه‌ای از متدها که در فایل .proto تعریف می‌شوند.
- **Remote Procedure Call:** هر متد معادل یک تابع از راه دور است که کلاینت می‌تواند آن را فراخوانی کند.
- **Messages:** ساختار داده‌ها در قالب Protobuf تعریف می‌شود (سریع‌تر و فشرده‌تر از JSON).
- **Stub:** کدی که از فایل proto تولید می‌شود و ارتباط بین کلاینت و سرور را تسهیل می‌کند.

انواع ارتباط:

- **Unary RPC:** یک درخواست → یک پاسخ.
- **Server Streaming:** یک درخواست → چند پاسخ متوالی.
- **Client Streaming:** چند درخواست → یک پاسخ نهایی.
- **Bidirectional Streaming:** چند درخواست و چند پاسخ به‌صورت هم‌زمان و دوطرفه.

ویژگی‌ها:

- **کارایی بالا:** استفاده از HTTP/2 و Protobuf باعث سرعت زیاد و حجم داده کم می‌شود.
- **قراردادمحور:** ارتباط کاملاً بر اساس فایل proto تعریف می‌شود؛ هم سرور و هم کلاینت از آن تبعیت می‌کنند.
- **پشتیبانی از چند زبان:** برای اکثر زبان‌ها (C#, Go, Python, Java و...) کد تولید می‌شود.
- **استفاده معمول در ارتباط بین سرویس‌ها (microservices):** مخصوصاً در محیط‌های توزیع‌شده.
- **بدون حالت بودن (Statelessness):** معمولاً هر فراخوانی RPC مستقل است، مگر در Streamها که ارتباط پایدارتر است.
- **کَشینگ:** به‌صورت ذاتی ندارد؛ باید در سطح اپ یا gateway انجام شود.

🔹 **یادآوری:** gRPC بیشتر برای ارتباط بین سرویس‌ها (service-to-service communication) طراحی شده، نه برای تعامل مستقیم با مرورگر. هدف آن سرعت، فشردگی و قابلیت اطمینان در سیستم‌های توزیع‌شده است.

مستندات خودکار

در بسیاری از APIها یک مسیر مخصوص مثل **/doc** یا **/swagger** وجود دارد که برای نمایش مستندات API استفاده می‌شود. این بخش معمولاً به برنامه‌نویس‌ها کمک می‌کند بفهمند API چگونه کار می‌کند و چطور باید درخواست ارسال کنند.

Swagger و OpenAPI

Swagger مجموعه‌ای از ابزارهاست که برای ساخت، نمایش و تست مستندات API استفاده می‌شود.

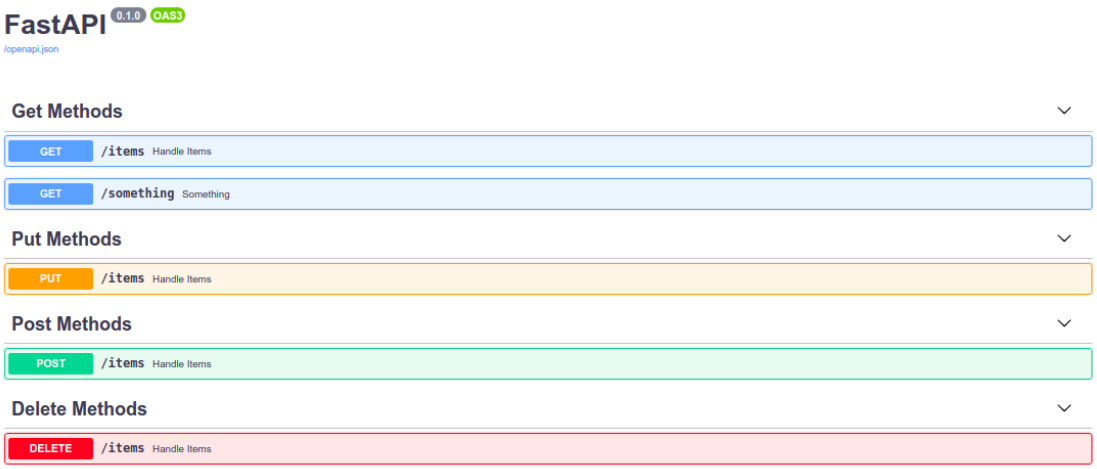
استاندارد اصلی‌ای که Swagger با آن کار می‌کند نامش **OpenAPI** است.

OpenAPI یک فرمت استاندارد است که مشخص می‌کند یک API چه مسیرهایی دارد، چه متدهایی را قبول می‌کند، چه ورودی و خروجی‌هایی دارد و چه خطاهایی برمی‌گرداند.

/swagger یا /doc چیست؟

در برنامه‌های وب (مثلاً در Node.js، .NET یا Java)، معمولاً یک صفحه خودکار ساخته می‌شود که مستندات API را نشان می‌دهد. این صفحه معمولاً روی آدرس‌هایی مثل:

- swagger/



- swagger/index.html/
- doc/
- api-docs/

در دسترس است.

این صفحه چه امکاناتی دارد؟

- لیست تمام endpointهای API را نمایش می‌دهد
- برای هر endpoint نوع درخواست (GET/POST/PUT/DELETE) را نشان می‌دهد
- مثال نمونه‌ی Path Params، Query Params، Body را نشان می‌دهد
- ساختار دقیق درخواست‌ها و پاسخ‌ها را توضیح می‌دهد
- می‌توانی درخواست واقعی بفرستی و نتیجه را همان‌جا ببینی (Try it out)

چرا مهم است؟

- **درک سریع API** بدون نیاز به خواندن کد
- **هماهنگی بهتر بین تیم‌ها** (فرانت‌اند، بک‌اند، موبایل...)
- **تست سریع endpointها** بدون نیاز به Postman
- **کم شدن خطاهای ارتباطی** چون همه می‌دانند ساختار پاسخ‌ها چیست
- **آپدیت خودکار** وقتی کد API تغییر کند مستندات هم به‌روز می‌شود

خلاصهٔ ساده:

- Swagger = ابزار + UI برای مستندات API
- OpenAPI = استاندارد توصیف API
- /swagger/ یا /doc/ = صفحه‌ای برای مشاهده و تست API

در FastAPI همه‌چیز به‌صورت خودکار وارد Swagger می‌شود، چون FastAPI از روی type hintها، پیدانتیک مدل‌ها و تعاریف روت‌ها مستندات OpenAPI را تولید می‌کند. یعنی هرچه در کد تعریف می‌کنید، همان لحظه در /docs (Swagger UI) و /redoc نمایش داده می‌شود.

در ادامه دقیق توضیح می‌دهیم هر بخش چطور وارد Swagger می‌شود

1. Path و Method چطور وارد Swagger می‌شوند؟

هر روتی که تعریف می‌کنیم (با @app.post، @app.get() و...) خودش به‌عنوان یک endpoint وارد Swagger می‌شود.

مثال:

```python
@app.get("/users")
def get_users():
    return [{"id": 1, "name": "Ali"}]
```

در Swagger بخش GET /users ظاهر می‌شود.

---

2. Path Params چطور شناخته می‌شوند؟

اگر در مسیر متغیر قرار بدهی (مثل /FastAPI، users/{id}) خودش تشخیص می‌دهد این یک Path Param است.

```python
@app.get("/users/{user_id}")
def get_user(user_id: int):
    return {"id": user_id}
```

Swagger نمایش می‌دهد:

- user_id (type: integer, required)

---

3. Query Params چطور مستند می‌شوند؟

هر آرگومانی که **در مسیر نیست** و یک مقدار پیش‌فرض دارد → Query Param می‌شود.

```python
@app.get("/search")
def search(q: str = "", limit: int = 10):
    return {"q": q, "limit": limit}
```

---

4. Body چطور مستند می‌شود؟

وقتی از **Pydantic model** استفاده می‌کنی، ساختار Body به‌طور کامل داخل Swagger نمایش داده می‌شود.

```python
from pydantic import BaseModel

class UserCreate(BaseModel):
    name: str
    age: int
    email: str

@app.post("/users")
def create_user(user: UserCreate):
    return user
```

Swagger نمایش می‌دهد:

→ Body schema

```json
{
  "name": "string",
  "age": 0,
  "email": "string"
}
```

5. Headerها چطور وارد Swagger می‌شوند؟

وقتی از Header() استفاده می‌کنی، FastAPI آن را در Swagger در بخش Header params نشان می‌دهد.

```
from fastapi import Header

@app.get("/items")
def read_items(token: str = Header(...)):
    return {"token": token}
```

Swagger می‌فهمد که token در Header است.

---

6. Response Model چطور در Swagger دیده می‌شود؟

اگر response_model تعریف کنی، Swagger ساختار پاسخ را دقیق نمایش می‌دهد.

```python
class User(BaseModel):
    id: int
    name: str

@app.get("/users/{id}", response_model=User)
def get_user(id: int):
    return User(id=id, name="Ali")
```

Swagger نمایش می‌دهد که API چه خروجی‌ای دارد.

اطلاعات و مثال های ارائه شده تنها بخشی از امکانات و جزئیات هستند. حتما مستندات اصلی را هم مطالعه کنید.

---

7. Status Codes

Status Codeها عددهایی هستند که سرور در پاسخ به درخواست برمی‌گرداند تا بگوید چه اتفاقی افتاده است. این کدها باعث می‌شوند کلاینت بدون نگاه‌کردن به متن پاسخ بفهمد نتیجهٔ درخواست موفق بوده، خطا داشته یا نیاز به اقدام دیگری هست.

به‌صورت خلاصه:

۱xx – اطلاعاتی (Informational)

سرور می‌گوید "درخواست را دریافت کردم، ادامه بده".

در APIها تقریباً استفاده نمی‌شود.

۲xx – موفقیت (Success)

یعنی درخواست درست بوده و انجام شده.

- **200 OK** → عملیات موفق
- **201 Created** → منبع جدید ساخته شد
- **204 No Content** → عملیات موفق بدون بدنه پاسخ

3xx – هدایت (Redirect)

یعنی منبع جابه‌جا شده یا باید به آدرس دیگری بروید.

در APIها کم‌استفاده است.

4xx – خطای کلاینت (Client Error)

مشکل در ورودی یا دسترسی کلاینت است.

- **400 Bad Request** → ورودی اشتباه
- **401 Unauthorized** → احراز هویت لازم است
- **403 Forbidden** → دسترسی کافی نیست
- **404 Not Found** → منبع وجود ندارد

- **Conflict 409** → تداخل داده (مثل ایمیل تکراری)

5xx – خطای سرور (Server Error)

سرور نتوانسته درخواست را پردازش کند.

- **Internal Server Error 500** → خطای داخلی
- **Service Unavailable 503** → سرویس موقتاً در دسترس نیست

خلاصهٔ خیلی کوتاه:

- **۲xx** = کار انجام شد
- **۴xx** = مشکل از کلاینت
- **۵xx** = مشکل از سرور

وقتی status_code تعیین می‌کنیم، Swagger آن را هم نشان می‌دهد.

```
@app.post("/users", status_code=201)
```

برای آشنایی کامل با status code ها به این ویدیو مراجعه کنید:

[HTTP Status Code Explained]

---

8. Descriptions و Examples

```python
class UserCreate(BaseModel):
    name: str
    age: int

    class Config:
        json_schema_extra = {
            "example": {
                "name": "Ali",
                "age": 25
            }
        }
```

Swagger یک example زیبا نشان می‌دهد.

---

9. توضیح متدها و روت‌ها با docstring

حتی docstring هم در Swagger نمایش داده می‌شود:

```python
@app.get("/users")
def get_users():
    """
این endpoint لیست کاربران را برمی‌گرداند.
    """
    return []
```

جمع‌بندی خیلی کوتاه

در FastAPI:

- مسیرها → از route decorator
- Path params → از {variable} در URL
- Query params → از آرگومان‌های فانکشن
- Body → از Pydantic models
- Header → با Header()
- Response model → با response_model
- Example / description → از docstring یا extra

همه‌ی این‌ها بدون هیچ کار اضافی در /docs (Swagger UI) ساخته می‌شود.

برای آشنایی کامل با صفحه /doc در fastApi لطفا به مرجع فریم ورک مراجعه کنید: FastApi

---

۴. درخواست‌ها و پاسخ‌ها (Requests & Responses)

Path Parameters

```
@app.get("/api/projects/{project_id}")
def get_project(project_id: int):
    ...
```

Query Parameters

```
@app.get("/api/tasks")
def list_tasks(skip: int = 0, limit: int = 10):
    ...
```

Request Body

ورودی‌ها در قالب JSON با **Pydantic Models** اعتبارسنجی می‌شوند.

Response Model

برای خروجی، از response_model استفاده می‌شود تا داده‌ها و ساختار پاسخ کنترل شوند.

۵. اعتبارسنجی داده‌ها (Validation)

در FastAPI، اعتبارسنجی ورودی‌ها (**Validation**) و تعریف ساختار داده‌ها با **Pydantic Models** انجام می‌شود. هدف این بخش این است که مطمئن شویم داده‌ای که کلاینت ارسال می‌کند **درست، کامل و در قالب مورد انتظار** است. بدون Validation، API می‌تواند داده‌های ناقص یا اشتباه دریافت کند و رفتارهای غیرقابل‌پیش‌بینی به وجود بیاید.

Validation یعنی بررسی صحت داده‌های ورودی قبل از اینکه وارد منطق برنامه یا دیتابیس شوند.

به عنوان مثال، اگر فیلد *email* ارسال شود اما مقدارش 1234 باشد، یا *age* عدد منفی باشد، API باید جلوی آن را بگیرد و خطای مناسب بدهد.

FastAPI این کار را به‌صورت **خودکار** انجام می‌دهد، چون تمام داده‌ها ابتدا از فیلتر Pydantic عبور می‌کنند.

این ویدئو میتواند برای درک این بخش مفید باشد.

Pydantic Model چیست؟

Pydantic یک کتابخانهٔ قدرتمند برای:

- تعریف ساختار داده‌ها (Data Models)
- تعیین نوع هر فیلد
- تعیین محدودیت‌ها (Constraints)
- تولید شِمای OpenAPI (مثل Swagger)
- تبدیل خودکار داده‌ها به انواع درست
- اعتبارسنجی ورودی‌ها و خروجی‌ها

است.

در FastAPI، هر درخواست بدنه‌دار (Body) معمولاً با یک Pydantic Model تعریف می‌شود:

```
from pydantic import BaseModel

class UserCreate(BaseModel):
    username: str
    email: str
    age: int
```

وقتی کلاینت در بدنه درخواست JSON ارسال می‌کند، FastAPI این داده‌های خام را **می‌گیرد، بررسی می‌کند، تبدیل می‌کند** و سپس در قالب یک شیء معتبر Python به تابع شما می‌دهد.

اعتبارسنجی خودکار چگونه کار می‌کند؟

وقتی مدل را تعریف می‌کنی:

- username باید رشته باشد
- email باید رشته باشد
- age باید عدد باشد

اگر ورودی با این قوانین ناسازگار باشد، FastAPI به‌صورت خودکار یک پاسخ خطای **Unprocessable 422** برمی‌گرداند و دقیق توضیح می‌دهد چه فیلدی اشتباه است.

مثال خطای خودکار:

```
{
  "detail": [
    {
      "loc": ["body", "age"],
      "msg": "value is not a valid integer",
      "type": "type_error.integer"
    }
  ]
}
```

افزودن محدودیت‌ها (Constraints) با Field

Pydantic اجازه می‌دهد محدودیت‌های دقیق‌تری تعریف کنید:

```
from pydantic import BaseModel, Field

class UserCreate(BaseModel):
    username: str = Field(min_length=3, max_length=20)
    email: str
    age: int = Field(gt=0, lt=120)
```

- username بین ۳ تا ۲۰ کاراکتر
- age باید بزرگ‌تر از ۰ و کوچک‌تر از ۱۲۰ باشد

تمام این قوانین **در Swagger هم خودکار نمایش داده می‌شوند**.

اعتبارسنجی های پیشرفته

Pydantic از موارد زیر هم پشتیبانی می‌کند:

- **regex** (مثلاً شماره تلفن)
- **List, Dict, Nested Models**
- **custom validators** با validator@
- **model inheritance**

مثال custom validator:

```python
from pydantic import BaseModel, validator

class User(BaseModel):
    password: str

    @validator("password")
    def check_password(cls, value):
        if len(value) < 8:
            raise ValueError("password must be at least 8 characters")
        return value
```

چرا Pydantic این‌قدر مهم است؟

- امنیت (ورودی‌های اشتباه رد می‌شوند)
- جلوگیری از آسیب‌پذیری‌ها
- API قابل‌پیش‌بینی‌تر و تمیزتر
- مستندسازی خودکار
- تبدیل خودکار داده‌ها (مثلاً رشته به int یا datetime)
- کنترل کامل بر ورودی و خروجی

Pydantic باعث می‌شود FastAPI حتی با کمترین کد، **کاملاً پایدار و قابل‌اعتماد** باشد.

خطاهای خودکار

FastAPI به‌صورت خودکار در صورت نقض اعتبارسنجی، پاسخ ۴۲۲ Unprocessable Entity تولید می‌کند.

۶. RESTful (Best Practices)

در طراحی یک **RESTful API** هدف این است که API قابل‌پیش‌بینی، خوانا، سازگار و قابل‌نگه‌داری باشد. برای رسیدن به این هدف، یک‌سری **Best Practice**های بسیار مهم وجود دارند که رعایتشان کیفیت API را چند برابر می‌کند

- استفاده از اسم جمع در مسیرها (/tasks/ ,projects).
- عدم استفاده از افعال در URL.
- تفکیک لایه‌ها:
  - Controller (Routes)
  - Service (Business Logic)
  - Repository (Data Access)

پاسخ یکنواخت با ساختار زیر:

1. از نام منابع (Resource Names) استفاده کنید، نه افعال

در REST، مسیرها باید با **اسم** موجودیت‌ها مشخص شوند، نه با عملیات.

❌ بد:

```
/getUsers
/createUser
/updateUserName
/deleteUser
```

✔️ درست:

```
GET /users
POST /users
PUT /users/{id}
DELETE /users/{id}
```

در REST، *متد HTTP خودش عمل را مشخص می‌کند*، بنابراین نیازی به فعل داخل URL نیست.

---

2. از اسم جمع برای منابع استفاده کنید

وقتی درباره مجموعه‌ای از چیزها صحبت می‌کنید، نام‌ها باید جمع باشند.

مثال‌های صحیح:

```
/users
/projects
/orders
/comments
```

این کار API را قابل پیش‌بینی می‌کند.

---

3. ساختار URL باید سلسله‌مراتبی و معنادار باشد

اگر منابع بین هم رابطه دارند، این رابطه باید داخل URL دیده شود.

مثال درست:

```
GET /projects/{project_id}/tasks
```

مثال غلط:

```
/getTasksForProject?id=3
```

---

4. از HTTP Status Codes درست استفاده کنید

استفاده از کد صحیح، API را قابل فهم می‌کند:

- 200 OK → درخواست موفق
- 201 Created → منبع جدید ساخته شد
- 400 Bad Request → مشکل در ورودی
- 401 Unauthorized → نیاز به توکن
- 403 Forbidden → توکن کافی نیست
- 404 Not Found → منبع وجود ندارد
- 409 Conflict → تداخل داده
- 500 Internal Server Error → خطای سرور

---

5. از Query Params فقط برای فیلتر، جستجو و صفحه‌بندی استفاده کنید

Query params نباید بخش اصلی ساختار باشند.

✔️ درست:

```
GET /products?category=phone&sort=price&limit=10
```

❌ غلط:

```
/getProducts?page=2
/updateTaskStatus?id=5&status=done
```

6. پاسخ‌ها باید یک ساختار ثابت و قابل پیش‌بینی داشته باشند

یک API خوب باید همیشه پاسخش ساختاری مشابه داشته باشد.

مثال پیشنهادی:

```
{
  "status": "success",
  "data": { ... },
  "message": "Operation successful"
}
```

و در خطا:

```
{
  "status": "error",
  "message": "Project not found",
  "code": 404
}
```

این الگو باعث می‌شود کلاینت نیاز به حدس‌زدن نداشته باشد.

---

7. از نسخه‌بندی (Versioning) استفاده کنید

وقتی قصد دارید تغییری به API اضافه کنید که ممکن است رفتار کلاینت‌های قبلی را بشکند، نسخهٔ جدید بسازید. از انجا که ورژنینگ در این تسک اهمیتی ندارد توضیح داده نمی‌شود اما مطالعه آن جهت آشنایی حداقلی پیشنهاد میشود.

مثال:

```
/api/v1/users
/api/v2/users
```

---

8. منابع را مطابق مدل Domain طراحی کنید

مسیرها باید بر اساس *موجودیت‌های واقعی سیستم* ساخته شوند، نه براساس عملیات فنی یا جداول دیتابیس.

مثال صحیح (بر اساس Domain):

```
/projects/{id}/tasks
```

مثال غلط:

```
/tbl_project_tasks/get_list
```

---

9. از HTTP Methods به شکل درست استفاده کنید

هر متد باید معنای واقعی خودش را حفظ کند:

- GET → فقط خواندن
- POST → ساخت
- PUT → جایگزینی کامل
- PATCH → اصلاح بخشی
- DELETE → حذف

مثال:

```
PATCH /users/5  (فقط تغییر name)
PUT /users/5    (جایگزینی کامل کاربر)
```

---

10. از اسامی شفاف و کوتاه استفاده کنید

URI را ساده، کوتاه و قابل حدس نگه دارید.

❌ بد:

```
/getAllActiveUsersFromDatabase
```

✔️ خوب:

```
/users?status=active
```

11. فیلترهای پیچیده را با Query Params ساختارمند کنید

مثال:

```
GET /orders?status=paid&from=2025-01-01&to=2025-02-01
```

---

12. عملیات خاص (Non-CRUD) را با Sub-resource یا Action Name محدود بنویسید

گاهی عملیاتی هست که CRUD نیست.

مثال استاندارد:

```
POST /projects/{id}/archive
POST /tasks/{id}/close
```

نه اینکه:

```
/archiveProject
/updateTaskToClosed
```

خلاصه

یک API زمانی RESTful و باکیفیت است که:

- مسیرها اسم باشند، نه فعل

- رابطه‌ها در URL دیده شود
- از متدهای HTTP درست استفاده شود
- پاسخ‌ها یک شکل و قابل پیش‌بینی باشند
- کد وضعیت‌ها صحیح استفاده شوند
- نسخه‌بندی رعایت شود
- query params فقط برای فیلتر و جستجو باشند

Synchronous and Asynchronous

در فریمورک‌هایی مثل FastAPI و Django، دو نوع مدل اجرای تابع وجود دارد: **Sync** (همگام) و **Async** (غیرهمگام). هرکدام رفتار متفاوتی در اجرای درخواست‌ها دارند و دانستن تفاوتشان برای طراحی یک API سریع و مقیاس‌پذیر ضروری است.

Sync (Synchronous)

در حالت Sync، کد **به‌صورت ترتیبی** اجرا می‌شود.

یعنی تا وقتی یک عملیات تمام نشود، عملیات بعدی شروع نمی‌شود.

وقتی یک درخواست وارد شود:

1. تابع Sync شروع به کار می‌کند.
2. اگر نیاز به عملیات کند (مثل I/O، دیتابیس، شبکه) باشد، **Thread** تا پایان آن بلاک می‌شود.
3. بعد از اتمام کار، پاسخ داده می‌شود.

مزایا

- ساده‌تر برای نوشتن و دیباگ.
- مناسب برای عملیات CPU-heavy یا منطق‌های سریع بدون I/O.

معایب

- هنگام عملیات I/O (دیتابیس، فایل، شبکه) **Thread قفل می‌شود**.
- برای API پر ترافیک استفاده از Sync می‌تواند باعث کندی شدید شود.

نمونه در FastAPI

```
def get_user():
    return {"msg": "sync response"}
```

Async (Asynchronous)

در حالت Async، کد **غیرهمگام** اجرا می‌شود. یعنی وقتی به عملیات کند می‌رسد، Thread آزاد می‌شود تا کارهای دیگر انجام دهد. در یک درخواست:

1. تابع Async شروع می‌شود.
2. اگر به صرف زمان برای I/O برسد (مثل درخواست دیتابیس):
    - تابع **await** می‌شود.
    - Thread آزاد می‌شود برای هندل کردن درخواست‌های دیگر.
3. وقتی I/O تمام شد، ادامه‌ی تابع اجرا می‌شود.

مزایا

- بسیار سریع‌تر برای APIهایی با درخواست زیاد.
- Thread بلاک نمی‌شود → مقیاس‌پذیری بالا.
- برای سیستم‌های real-time یا microservices عالی است.

معایب

- پیچیدگی بیشتر در نوشتن و دیباگ.
- اگر کتابخانه شما async-safe نباشد، باید از sync استفاده کنید.

نمونه در FastAPI

```
async def get_user():
    return {"msg": "async response"}
```

چه زمانی از Sync استفاده کنیم؟

- عملیات CPU-heavy مثل پردازش تصویر یا رمزنگاری
- کارهای کوچک و سریع بدون I/O (یا بسیار کم)
- وقتی کتابخانه‌ای که استفاده می‌کنید async نیست (مثل برخی ORMهای قدیمی)

چه زمانی از Async استفاده کنیم؟

- APIهایی با تعداد درخواست زیاد
- سیستم‌هایی که زیاد با دیتابیس/شبکه کار می‌کنند
- میکروسرویس‌ها
- زمانی که می‌خواهید performance بالا داشته باشید

چرا FastAPI با Async محبوب است؟

FastAPI روی **asyncio** نوشته شده و کاملاً از async پشتیبانی می‌کند. اگر endpointهای شما async باشند:

- concurrency بسیار بالا
- مدیریت بهتر منابع

- پاسخ‌دهی سریع‌تر تحت بار بالا

تسک فاز ۳

در فاز سوم پروژهٔ ToDoList قرار است **رابط CLI کنار گذاشته (Deprecate)** شود و سیستم به‌سمت یک **رابط وب‌سرویس بر پایهٔ FastAPI** منتقل شود و تمام قابلیت های قبلی از این طریق قابل انجام باشند. این موضوع فقط «حذف کردن CLI» نیست؛ بلکه شامل یک فرآیند **مدیریت تغییرات** است که باید به‌صورت اصولی، مرحله‌به‌مرحله و قابل‌پیگیری انجام شود. در ادامه توضیح کامل و مناسب برای درس *مهندسی نرم‌افزار* را می‌نویسم.

Deprecate کردن یعنی اعلام رسمی اینکه یک قابلیت، ابزار یا بخش از سیستم **دیگر نباید استفاده شود** و در آینده **به‌طور کامل حذف خواهد شد**.

اما نکتهٔ مهم اینجاست:

Deprecation = حذف فوری نیست.
Deprecation = «اعلان پایان عمر» یک قابلیت است.

وقتی یک بخش deprecate می‌شود:

- هنوز *فعلاً* وجود دارد
- هنوز *فعلاً* کار می‌کند
- اما توصیه می‌شود **از روش یا سیستم جدید استفاده شود**
- و در نسخهٔ بعدی یا فاز بعدی **به‌طور کامل حذف می‌شود**

این کار برای جلوگیری از شکستن سیستم، مدیریت درست کاربران، و انتقال تدریجی انجام می‌شود.

چرا باید CLI را deprecate کنیم؟

در فازهای اولیه، CLI برای سرعت و سادگی مناسب بود، اما مشکلاتی دارد:

- رابط کاربری سخت و محدود
- عدم امکان اتصال آسان به برنامه‌های دیگر
- سختی توسعه و نگه‌داری
- نبود استاندارد برای تعامل‌های پیچیده
- نبود مستندات خودکار برای ورودی/خروجی

برای فاز سوم، هدف ساخت **یک API استاندارد، قابل‌گسترش و مستقل از UI** است.

این کار **معماری را لایه‌لایه، استاندارد و قابل استفاده در وب یا موبایل** می‌کند.

چگونه باید Deprecation انجام شود؟ (روش اصولی)

در مهندسی نرم‌افزار، deprecate کردن یک قابلیت معمولاً در سه مرحله انجام می‌شود و در این فاز پروژه فقط مرحله اول و دوم مدنظر است:

1. مرحله اعلام (Deprecation Notice)

در این مرحله اعلام می‌کنید:

- CLI دیگر Feature اصلی نیست
- استفاده از آن توصیه نمی‌شود
- قابلیت‌های جدید فقط در API اضافه می‌شوند

این اعلان باید:

- در README و مستندات اضافه شود
- در کلاس‌های CLI (در docstring) قید شود
- در صورت امکان یک پیام هشدار در اجرای CLI چاپ شود

مثال پیام هشدار CLI:

```
WARNING: CLI interface is deprecated and will be removed in the next release.
Please use the FastAPI HTTP interface instead.
```

---

2. مرحله انتقال (Migration Phase)

در این مرحله، CLI هنوز وجود دارد ولی:

- توسعهٔ جدید روی CLI انجام نمی‌شود
- قابلیت‌های جدید فقط به API منتقل می‌شوند
- لایه‌های اصلی نرم‌افزار (Service / Repository) مشترک می‌شوند
- رفتار CLI از طریق API قابل جایگزینی می‌شود

در این فاز باید:

- Routerها در FastAPI طراحی شوند
- مدل‌ها به Pydantic منتقل شوند
- Validation استاندارد شود
- منطق CLI به Service Layer منتقل شود تا API بتواند آن را استفاده کند

این مرحله **مهم‌ترین بخش پروژه است** چون migration واقعی اینجاست.

---

3. مرحله حذف نهایی (Removal Phase)

وقتی تمام قابلیت‌های CLI در API قابل انجام شد:

- CLI به‌طور کامل حذف می‌شود
- فولدرها و فایل‌های مربوطه پاک می‌شوند
- مسیر اجرای پروژه فقط از طریق FastAPI خواهد بود

در این مرحله، نسخهٔ جدید پروژه منتشر می‌شود که CLI دیگر در آن نیست.

توسعه TodoList Api

در ادامه تمام قابلیت های todolist را با استفاده از FastApi قابل دسترس کنید. در خلال این مستند لینک هایی برای اشنایی با اکثر ابزار مورد نیاز این بخش ارائه شد در نهایت برای پیاده سازی میتوانید از این منابع به عنوان مرجع استفاده کنید.

- 🎥 Introduction to FastAPI — YouTube
- 🎥 Building REST API with FastAPI — YouTube
- 🎥 FastAPI Full Crash Course
- 🎥 FastAPI with SQLAlchemy — YouTube
- 🎥 Pydantic data validation
- 📚 FastAPI Documentation

- 📹Python FastAPI Tutorial: Build a REST API in 15 Minutes
- 📹FastAPI Crash Course 2022 | REST API with the most popular Python framework
- 📹FastAPI and Pydantic - Model Classes and Nested Models
- 📹FastAPI + Pydantic Tutorial – Validate Your Data Like a Pro
- 📹Master Pydantic Models in FastAPI | Building APIs using FastAPI

دیدن همه این ویدئوها الزامی نیست و صرفا برای صرفه جویی در وقت شما برای پیدا کردن منابع و توضیحات کافی هستند. اما پیشنهاد می شود آموزش کوتاه FastApi و مستندات Pydantic را مطالعه کنید و تعدادی از فیلم ها را ببینید.

برای ساختار پروژه از ساختار زیر استفاده کنید

```
i want to add this text as a code block to a document and i want it to have text coloring such that its easier to r
.
├── app
│   ├── api     #API Layer
│   │   │   ├── controllers         #FastApi controllers
│   │   │   │   ├── users_controller.py
│   │   │   │   └── __init__.py
│   │   │   ├── routers.py
│   │   │   ├── controller_schemas              #pydantic models for request and response data validation in api
│   │   │   │   ├── requests
│   │   │   │   │   ├── users_request_schema.py
│   │   │   │   ├── responses
│   │   │   │   │   ├── users_response_schema.py
│   │   │   └── __init__.py
│   │   └── __init__.py
│   │
│   ├── services              # Business Logic Layer
│   │   ├── user_service.py
│   │   ├── auth_service.py
│   │   ├── item_service.py
│   │   └── __init__.py
.
.
.
|
|
├── main.py                   # app initialization
├── pyproject.toml
├── README.md
└── .env
```

از شما خواسته میشود:

1. **از رده خارج کردن CLI** را طبق توضیحات انجام دهید
2. **طراحی مسیرهای API** را مطابق اصول RESTful و استانداردهای نام‌گذاری، نسخه‌بندی و ساختاردهی انجام دهید.
3. **کنترلرها را به لایهٔ سرویس و منطق تجاری متصل کنید** تا جداسازی مسئولیت‌ها (Controller → Service → Repository) کاملاً رعایت شود.
4. **کنترلرها را به‌گونه‌ای پیاده‌سازی کنید** که مستندات خودکار (Swagger FastAPI و OpenAPI Docs) شفاف، دقیق و قابل‌استفاده باشند؛ شامل تعریف صحیح مدل‌ها، متدها، توضیحات و مثال‌ها.
5. **استفاده درست از توابع sync و async.**
6. **اعتبارسنجی ورودی و خروجی** را با استفاده از Pydantic Models به‌صورت کامل و دقیق اعمال کنید تا ساختار داده‌ها، انواع (types) و محدودیت‌ها (constraints) کاملاً کنترل شوند.

رعایت کانونشن های پایتون کفته شده، استفاده از Poetry و رعایت Git Policy در این فاز الزامی است و در ارزیابي نهایی پروژه لحاظ مرشود.

روی ریپوی فاز یك باید این تغییرات و موارد خواسته شده رو پیاده سازی کنید.

ددلاین و زمانبندی تحویل پروژه

ددلاین تحویل این فاز پروژه تا ۳ اذر می باشد و با فاز قبلی در یک ارائه ارزیابی خواهد شد.