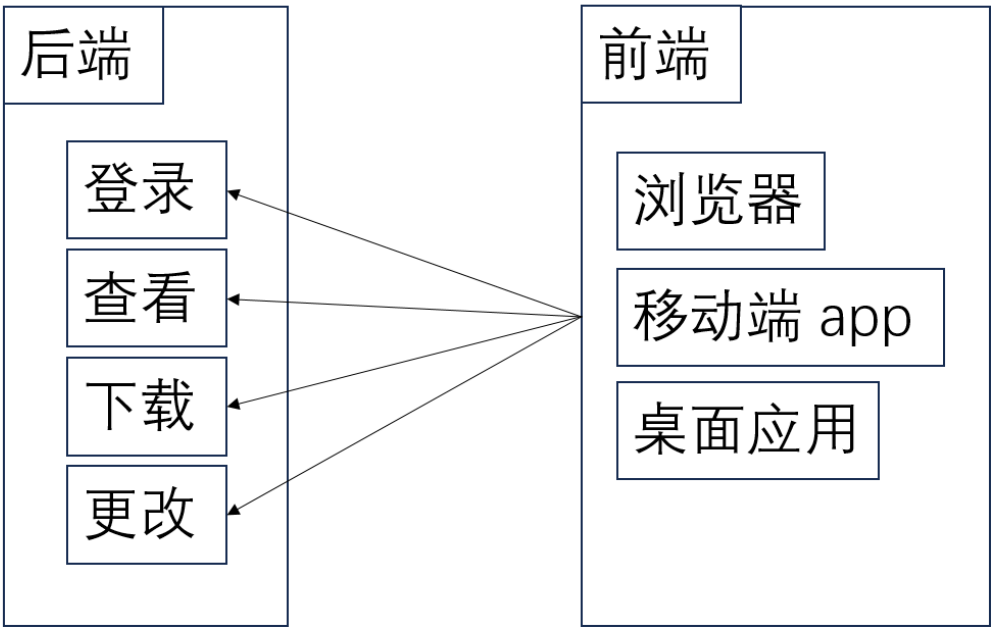
06-使用 FastAPI 编写 Web 后端

FastAPI

课程目的：

1. 了解典型的 Web 应用概念，前后端概念
2. 理解 FastAPI 做了什么
3. 了解 app 实例，了解路由
4. 进一步知道 FastAPI 路由处理对应 HTTP 报文的部分

比如某 Web 应用长这样：



前端会有很多前端，后端会有很多概念。
该应用无论有多少个前端，他们的功能都由后端统一提供。这是为了保证**同一款产品在不同前端上的行为一致。**

后端就是向前端提供功能服务的——无论是哪个前端向它发起请求，都会受到一致的响应。不同的功能被视为模块（模块化），在后端开发上，你的任务是根据核心模块，不断地拓展模块，使功能变得丰富。
因此可以说，后端框架的特征，是**模块化**、**统一接口**、**可扩展**（可拼接）的。

官网：FastAPI 是一个用于构建 API 的现代、快速（高性能）的 web 框架，使用 Python 并基于标准的 **Python 类型提示**，类型提示，建议自学。
FastAPI 的开发依赖类型标注技术，其官网有简要介绍，也可以去python 官网查看最系统的解释，请自学.

如果缺少基础概念，建议看 GitHub 上的公共知识库，或者运用搜索引擎。

FastAPI 项目

- 工具
- 项目结构
- app 与服务

工具

工具上你要准备 FastAPI 包和 uvicorn 包，以支持 FastAPI 开发。建议在官网找到 installation，并在 venv 中使用 pip 安装到虚拟环境。

```
pip install fastapi
pip install "uvicorn[standard]"
```

> uvicorn 是一个后端框架无关的，用于启动 HTTP 服务的工具。

也就是说，你需用 Uvicorn 来运行你的 FastAPI 应用。

项目结构

项目结构：
一个典型的 FastAPI 项目应该包含以下文件结构：

```
fastapi_project/
├── app/
│   ├── __init__.py
│   ├── main.py               # 主应用程序入口
│   ├── api/                  # API 路由
│   │   ├── __init__.py
│   │   └── endpoints/        # 具体的 API 端点
│   ├── models/               # 数据模型
│   ├── schemas/              # Pydantic 模式
│   ├── services/             # 业务逻辑层
│   └── database/             # 数据库连接与管理
│
├── tests/                    # 测试文件，正规开发会用到
│   ├── __init__.py
│   └── test_main.py          # 主要测试文件
│
├── requirements.txt          # 项目依赖，用 pip-install -r requirements.txt 来进行装包
└── README.md                 # 项目说明文件
```

在 FastAPI 项目中，通常以`main.py`作为启动文件——仅是习惯命名。
而核心逻辑可能集中在别的模块（如`app/api`、`app/services`等目录下）。

app 与服务

基础FastAPI应用，请查看`main.py`
`@app.get`是 python 的装饰器，用于定义路由。用装饰器修饰的函数会被**处理一次**。这里的处理指**将函数注册到 app 的路由表中。**
`async`用于异步操作，几乎所有网络相关的操作都是异步的。详情见 public 编程语言 language 和 网络 network 部分。

```
uvicorn {入口文件名}:app --reload
```

以上案例说明了，当配置了 app 最基础的部分后（大部分配置都可以通过对app进行设置实现），我们可以通过访问服务器根路由`/`来得到文本“Hello World”。这就是用 FastAPI 开发的最简单的基于 HTTP 的后端服务（server）应用。

路由

以某站为例：

```
xxx.com/                          # 首页
xxx.com/about                     # 关于页
xxx.com/about/team                # 团队页
xxx.com/login                     # 登录页
xxx.com/user/:username            # 用户页
xxx.com/video/:id                 # 视频页
xxx.com/search?keyword=value      # 搜索页
```

其中，`xxx.com`是网站的域名，后面的部分称为“路由”（`route`）。

`/about`、`/about/team`为一级、二级路由。中文常作**相关/关于页**用于展示网站、网站团队的静态图文信息。

`/login`为登录页，在输入用户名密码通过登录验证后，服务器将返回给客户端（浏览器，在用户机器上）一个 token（凭证），以便用户在后续访问时验证身份。token 常用浏览器 cookie 来存储。

> cookie：一种在用户浏览器上存储少量数据的技术。本质是临时文件。

`/user`为用户页，用于展示用户的个人信息。其中`:username`部分为动态路由，意味着`/user`后面可以跟任意的用户名（在地址栏中任何符合该格式的链接都能被匹配识别到该路由，而非其它路由上）。显然，不同用户访问某用户页时，分为**本人**和**非本人**两种情况，它们的操作逻辑也分为**查看他人信息**与**查看和修改本人信息。他人**和**本人**通过登陆后获取的 token 来区分。

`/video`为视频页，用于展示视频内容。其中`:id`部分为动态路由，意味着`/video`后面可以跟任意的视频 ID。

`/search`为搜索页，用于搜索关键字相关的内容。`/search`跟随的是http的查询参数（以`?`开头，由键值对`key=value`来提供）。

至此我们清楚了一般 Web 应用后端的需求：根据 url 链接来处理内含的不同需求，并返回给客户端。我们把处理这些需求的功能集中到一个进程，挂在在 server（服务器）上，称为 service（服务，对操作系统而言，区别于其它进程，一种进程类型）。

> url：统一资源定位符，以`http://`或`https://`开头（如 http://www.example.com）
> uri：统一资源标识符，以`{example}:`开头（如 mysql://localhost/test）
> 两者都表现为链接

FastAPI 如何工作?

FastAPI 的核心原理很简单：创建一个 FastAPI app 实例，它会监听用户发来的 HTTP 请求，然后根据请求的内容决定要返回什么。

返回 2 个类型的数据

返回 text：
[课后] `main.py`

返回 Json：
当用户访问 `/json` 时，网站会返回一个结构数据：`{"message": "Hello Json!"}`。
[课后] `main.py`

路由的处理

显然，我们的任务就是对通过 url 发送来的请求，做不同处理。可以认为处理路由就是后端业务的核心。

路径参数（动态路由）

FastAPI 可以很方便地处理这种“动态网址”，我们只需要在路由中指定一个参数（比如 `:username`）。
比如对于 `/user/:username`
当用户访问 `http://localhost:8000/user/john` 时，页面会显示“the user's name is: john”。
[课后] `main.py`

查询参数

比如在请求头的 URL 里添加 QueryParams（查询参数）。
查询字符串是键值对的集合，这些键值对位于 URL 的？之后，以 & 分隔。
比如对于 `/search?keyword=value`
当用户访问 `http://localhost:8000/search?keyword='google'` 时，页面会显示“用户搜索了关键词: google”。
[课后] `main.py`

body

上面我们一直用的是 `@app.get` 装饰器，注册 GET 请求的路由。
对于 POST 请求，我们可以获取到请求体中的数据。
比如对于 `/login`
这里突然从 `@app.get` 切换到 `@app.post` 的理由，后续马上介绍。

> 注意：token 往往是临时的，动态生成的，加密过后的。上述例子只是简化，记得用专门的 token 生成用的加密方法（函数）。

当用户发送 POST 请求到 `/login` 时，我们能分情况返回**登陆成功时的 token** 或**错误信息**。
[课后] `main.py`

路由和报文的对应关系

一个典型的 HTTP 请求报文（Request）包含如下内容：

部分	类型	举例
请求信息	字符串	类型 method、URL 链接、HTTP 版本...
请求头 headers	键值对	Content-Type、User-Agent、Accept-Language...
请求体 body	字节流	表单数据、JSON 数据、文件上传

1. HTTP method 常用的有 GET、POST（PUT、DELETE、HEAD、OPTIONS、TRACE）。`@app.get`明确了某路由下的特定类型（此处为 GET），同一路由可以有不同类型的请求。
2. GET 和 POST 的选择依据是：GET 请求的信息往往放在 URL 里（HTTP 标准设计上 GET 不包含 body），明文 URL 并不安全且信息量少（URL 长度限制）；POST 请求的信息放在 body 里，信息量更大且安全性更高。
3. 路径参数在**请求信息**的 URL 中
4. 查询参数在**请求信息**的 URL 末尾，以`?`开头，多个参数以`&`分隔
5. body 在**请求体**中，和其等价

HTTP 与 Web 标准

事实上，Web 标准自第一次制定以来，已经过去了 30 多年，在各类前端中的实现也各不一样，有相当多的坑点需要规避，虽然 FastAPI 帮我们封装了很多，但是在实际开发中，我们还是需要了解一些基本的 HTTP 和 Web 标准。

如果想要知道 Cookie、Content-Type、状态码、跨域等等标准及设计，可以在阅读完 public 后，于 MDN Web Docs 上查找详情。

必须知道的衍生

从文件分离，到路由分类集中

[课后] 看`project`文件夹

中间件

大致是讲如何在路由动作之前（或之后），统一加一个操作。这个被加在路由之前的操作，称为中间件（Middleware）。
比如在每个路由上加一个 looger，记录每个请求的日志。
具体查看文档。

数据存储：从内存到数据库

内存存储

```
my_data = []
@app.post("/add_data")
async def add_data(data: DataModel):
    my_data.append(data)
```

当数据量过大时，改为数据库存储

```
db = DataBase.connect(...)
@app.post("/add_data")
async def add_data(data: DataModel):
    db.insert(db_data)
```