

# Introdução ao desenvolvimento WEB

| <https://fastapidozero.dunossauro.com/02/>

# Objetivos dessa aula

- Criar uma base teórica sobre desenvolvimento web
- Apresentar o protocolo HTTP
- Introduzir os conceitos de APIs JSON
- Apresentar o OpenAPI
- Introduzir os schemas usando Pydantic

# A WEB

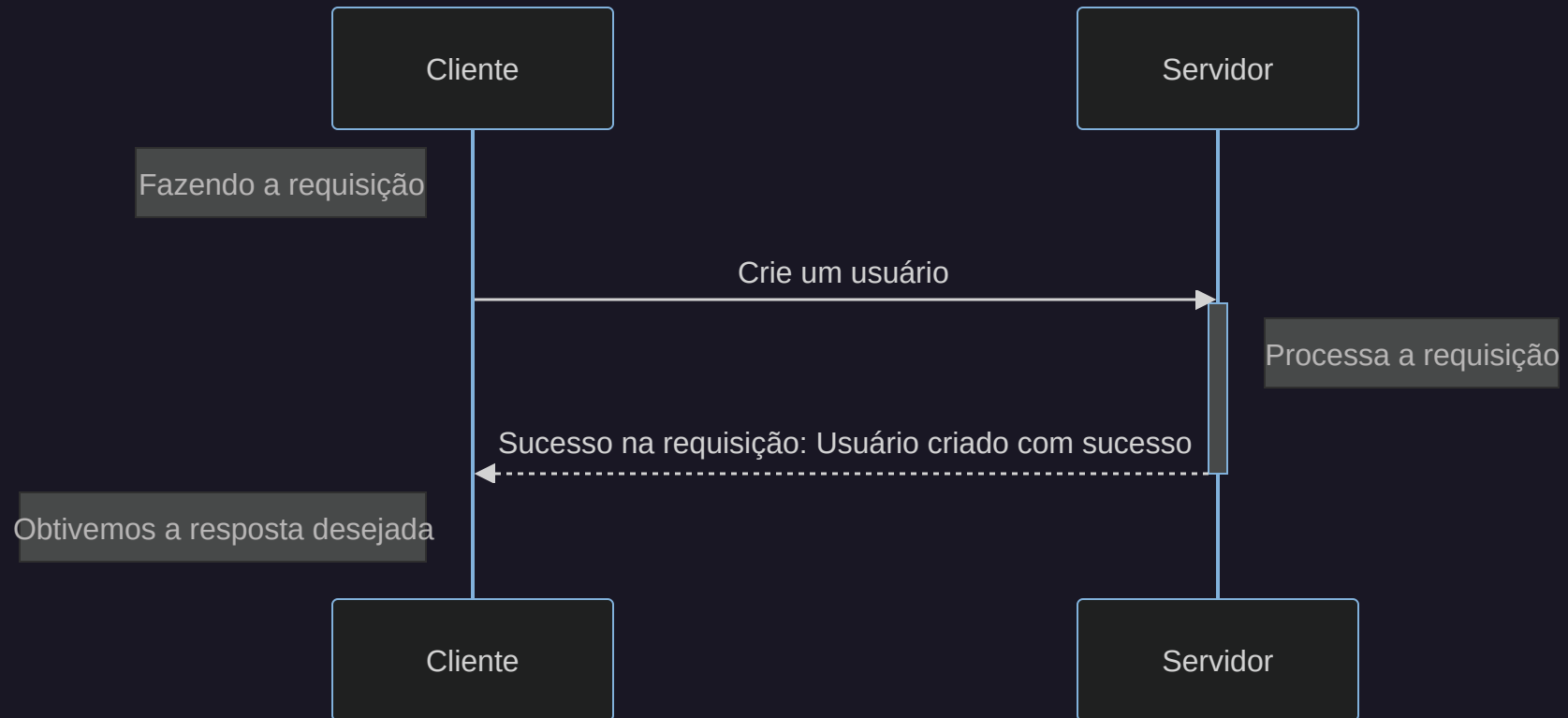
Sempre que nos referimos a aplicações web, estamos falando de aplicações que funcionam em rede.

- Dois ou mais dispositivos interconectados
- Local (**LAN**): como em sua casa ou em uma empresa
- Longa distância (**WAN**): Como diversos roteadores interconectados
- Mundial: como a própria internet

A ideia é a comunicação entre esses dispositivos.

# Cliente-Servidor

Quando falamos em comunicação, existem diversos formatos. O mais importante pra nós é o cliente-servidor.



# Cliente-servidor

Quando executamos o `fastapi` pelo comando:

```
fastapi dev fast_zero/app.py
```

Estamos iniciando um servidor web de desenvolvimento. Por isso a flag `dev`.



# O Uvicorn

Ao executar o comando `fastapi dev`, ao fim da mensagem no terminal, vemos:

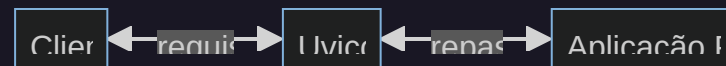
```
INFO:      Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO:      Started reloader process [893203] using WatchFiles
INFO:      Started server process [893207]
INFO:      Waiting for application startup.
INFO:      Application startup complete.
```

Embora o FastAPI seja um ótimo framework web, ele não é um "servidor de aplicação". Por baixo dos panos, ele chama o `Uvicorn`.

# Uvicorn

O uvicorn é um servidor de **aplicação**. Um servidor **ASGI**.

A responsabilidade dele é fazer a "cola" entre as chamadas de rede e repassar isso para o "código puro". Uma estrutura de alta performance para trabalhar com chamadas de rede.

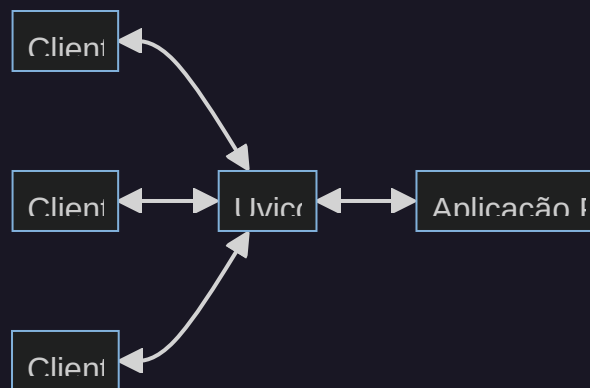


Você também poderia usar diretamente o uvicorn:

```
uvicorn fast_zero.app:app
```

# A rede local

Até esse momento, estamos usando ainda o "loopback", o nosso pc é o cliente e o servidor ao mesmo tempo. O que não é muito prático ainda, pois queremos fazer uma aplicação para diversos clientes.





# Servindo na rede local (LAN)

Saindo do loopback, podemos abrir o servidor do `uvicorn` para rede local:

```
fastapi dev fast_zero/app.py --host 0.0.0.0
```

Assim, toda a sua rede domestica (ou empresarial) já podem acessar sua aplicação se souberem o ip.

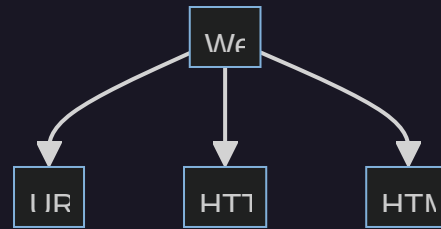
| Pode chamar alguém de casa, ou acessar por outro dispositivo `http://seu_ip:8000`

# Seu IP local com python

```
>>> import socket
>>> s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
>>> s.connect(("8.8.8.8", 80))
>>> s.getsockname()[0]
'192.168.0.100'
```

Você também pode usar comandos como `ipconfig`, `ip addr`, ...

# O modelo padrão da web



- **URL:** *Localizador Uniforme de Recursos*. Um endereço de rede pelo qual podemos nos comunicar com um computador na rede.
- **HTTP:** um protocolo que especifica como deve ocorrer a comunicação entre dispositivos.
- **HTML:** a linguagem usada para criar e estruturar páginas na web.

# URL



The diagram illustrates the components of the URL `http://127.0.0.1:8000`. It features three labels with purple curly brackets underneath them: 'Protocolo' is positioned above the 'http' part, 'Endereço' is positioned below the '127.0.0.1' part, and 'Porta' is positioned above the '8000' part. The entire URL is displayed in a large, white, monospaced font.

Protocolo

Porta

`http://127.0.0.1:8000`

Endereço

# URL

Onde está o que queremos acessar

Um filtro do recurso

/caminho/recurso?query#fragmento

A identificação do que queremos

Especifica um pedaço do recurso

# HTTP

HTTP, ou Hypertext Transfer Protocol (Protocolo de Transferência de Hipertexto), é o protocolo fundamental na web para a transferência de dados e comunicação entre clientes e servidores. Ele baseia-se no modelo de requisição-resposta: onde o cliente faz uma requisição ao servidor, que responde a essa requisição. Essas requisições e respostas são formatadas conforme as regras do protocolo HTTP.

# HTTP - Mensagens

No contexto do HTTP, tanto requisições quanto respostas são referidas como mensagens. As mensagens HTTP na versão 1 têm uma estrutura textual semelhante ao seguinte exemplo:

```
GET / HTTP/1.1
Accept: */*
Accept-Encoding: gzip, deflate
Connection: keep-alive
Host: 127.0.0.1:8000
User-Agent: HTTPie/3.2.2
```

# HTTP - Mensagem de resposta

```
HTTP/1.1 200 OK
content-length: 24
content-type: application/json
date: Fri, 19 Jan 2024 04:05:50 GMT
server: uvicorn
```

```
{
  "message": "Olá mundo"
}
```



# HTTP - Cabeçalho

O cabeçalho contém metadados sobre a requisição ou resposta:

- **Content-Type:** O tipo de mídia no corpo da mensagem. Por exemplo, `application/json` indica que o corpo da mensagem está em formato JSON. Ou `text/html`, para mensagens que contém HTML.
- **Authorization:** Usado para autenticação, como tokens ou credenciais.\*
- **Accept:** Especifica o tipo de mídia que o cliente aceita, como `application/json`.
- **Server:** Fornece informações sobre o software do servidor.

# HTTP - Verbos

Quando um cliente faz uma requisição HTTP, ele indica sua intenção ao servidor com verbos:

- **GET**: utilizado para recuperar recursos. Quando queremos solicitar um dado já existente no servidor.
- **POST**: permite criar um novo recurso. Por exemplo, enviar dados para registrar um novo usuário.
- **PUT**: Atualiza um recurso existente. Como, por exemplo, atualizar as informações de um usuário existente.
- **DELETE**: Exclui um recurso. Por exemplo, remover um usuário específico do sistema.

Na nossa aplicação FastAPI, definimos que a função `read_root` que será executada quando uma requisição GET for feita por um cliente no caminho `/`:

```
@app.get('/')  
def read_root():  
    return {'message': 'Olá Mundo!'}
```

# HTTP - Códigos de resposta

- 1xx: informativo — utilizada para enviar informações para o cliente de que sua requisição foi recebida e está sendo processada.
- 2xx: sucesso — Indica que a requisição foi bem-sucedida (por exemplo, 200 OK, 201 Created).
- 3xx: redirecionamento — Informa que mais ações são necessárias para completar a requisição (por exemplo, 301 Moved Permanently, 302 Found).

# HTTP - Códigos de resposta

- 4xx: erro no **cliente** — Significa que houve um erro na requisição feita pelo cliente (por exemplo, 400 Bad Request, 404 Not Found).
- 5xx: erro no **servidor** — Indica um erro no servidor ao processar a requisição válida do cliente (por exemplo, 500 Internal Server Error, 503 Service Unavailable).

Para mais informações a cerca do *status code* acesse a documentação do [iana](#)

# Códigos importantes para o curso

- **200 OK:** a solicitação foi bem-sucedida. O significado exato depende do método HTTP utilizado na solicitação.
- **201 Created:** a solicitação foi bem-sucedida e um novo recurso foi criado como resultado.
- **404 Not Found:** o recurso solicitado não pôde ser encontrado, sendo frequentemente usado quando o recurso é inexistente.
- **422 Unprocessable Entity:** usado quando a requisição está bem-formada, mas não pode ser seguida devido a erros semânticos. É comum em APIs ao validar dados de entrada.
- **500 Internal Server Error:** quando existe um erro na nossa aplicação

# FastAPI e códigos de resposta

Por padrão, o FastAPI já usa `200 OK` como código de resposta. Mas, podemos dizer isso explicitamente:

```
@app.get("/", status_code=HTTPStatus.OK)
def read_root():
    return {'message': 'Olá Mundo!'}
```

# HTML

O terceiro pilar fundamental da web é o HTML, sigla para Hypertext Markup Language.

Trata-se da linguagem de marcação padrão usada para criar e estruturar páginas na internet. Quando acessamos um site, o que vemos em nossos navegadores é o resultado da interpretação do HTML. Esta linguagem utiliza uma série de 'tags' – como `<html>`, `<head>`, `<body>`, `<h1>`, `<p>` e outras – para definir a estrutura e o conteúdo de uma página web.

Todo o código apresentado neste tópico é apenas um exemplo básico do uso de HTML com FastAPI e não será utilizado no curso. No entanto, é extremamente importante mencionar este tópico.



# Modelo de resposta

```
from fastapi import FastAPI
from fastapi.responses import HTMLResponse

app = FastAPI()

@app.get('/', response_class=HTMLResponse)
def read_root():
    return """
    <html>
      <head>
        <title> Nosso olá mundo!</title>
      </head>
      <body>
        <h1> Olá Mundo </h1>
      </body>
    </html>"""
```

# Um passo pra trás

Embora o HTML seja crucial para a estruturação de páginas web, nosso curso foca em uma perspectiva diferente: a transferência de dados. Enquanto o HTML é usado para apresentar dados visualmente nos navegadores, existe outra camada focada na transferência de informações entre sistemas e servidores.

# APIs

Aqui entra o conceito de APIs (Application Programming Interfaces), que ~~frequentemente~~ utilizam JSON (JavaScript Object Notation) para a troca de dados. JSON é um formato leve de troca de dados, fácil de ler e escrever para humanos, e simples de interpretar e gerar para máquinas.

As APIs originais, o termo original, se refere a HTML como a base das APIs **referência**

# JSON

Quando discutimos APIs ""modernas"", nos referimos a APIs que priorizam o tráfego de dados, deixando de lado a camada de apresentação, como o **HTML**.

O objetivo é transmitir dados de forma agnóstica para diferentes tipos de clientes.

Nesse contexto, o JSON (JavaScript Object Notation) se tornou a mídia padrão, graças à sua leveza e facilidade de leitura tanto por humanos quanto por máquinas.

# O JSON

```
{
  "livros": [
    {
      "titulo": "O apanhador no campo de centeio",
      "autor": "J.D. Salinger",
      "ano": 1945,
      "disponivel": false
    },
    {
      "titulo": "O mestre e a margarida",
      "autor": "Mikhail Bulgákov",
      "ano": 1966,
      "disponivel": true
    }
  ]
}
```

# Contratos

Quando falamos sobre o compartilhamento de JSON entre cliente e servidor, é crucial estabelecer um entendimento mútuo sobre a estrutura dos dados que serão trocados.

A este entendimento, denominamos **schema**, que atua como um contrato definindo a forma e o conteúdo dos dados trafegados.

# Pydantic

No universo de APIs e contratos de dados, especialmente ao trabalhar com Python, o Pydantic se destaca como uma ferramenta poderosa e versátil. Além de embutida no FastAPI.

A ideia dele é criar uma camada de documentação, via OpenAPI, e de fazer a validação dos modelos de entrada e saída da nossa API.

# Pydantic

Vamos criar um novo arquivo em nosso projeto chamado `fast_zero/schemas.py`:

```
from pydantic import BaseModel

class Message(BaseModel):
    message: str
```

Aqui temos a ideia de um json representada em python. Um objeto de chave `message`, com um valor do tipo `str`.



# Pydantic + FastAPI

Ao juntar o pydantic ao modelo de resposta, temos a garantia que a resposta seguirá esse formato e também documentará isso na API.

```
from http import HTTPStatus

from fastapi import FastAPI

from fast_zero.schemas import Message

app = FastAPI()

@app.get('/', status_code=HTTPStatus.OK, response_model=Message)
def read_root():
    return {'message': 'Olá Mundo!'}
```

# Documentação com schemas

Se iniciarmos o nosso servidor `task run` e entrarmos nas documentações, podemos ver o efeito:

- No swagger: <http://localhost:8000/docs>
- No redoc: <http://localhost:8000/redoc>

## Exercicio e quiz

Crie um endpoint que retorna "olá mundo" usando HTML e escreva seu teste. Dica: para capturar a resposta do HTML do cliente de testes, você pode usar `response.text`

quiz: [https://fastapidozero.dunossauro.com/quizes/aula\\_02/](https://fastapidozero.dunossauro.com/quizes/aula_02/)

# Commit

```
git add .  
git commit -m "Adicionando schema de mensagem"  
git push
```