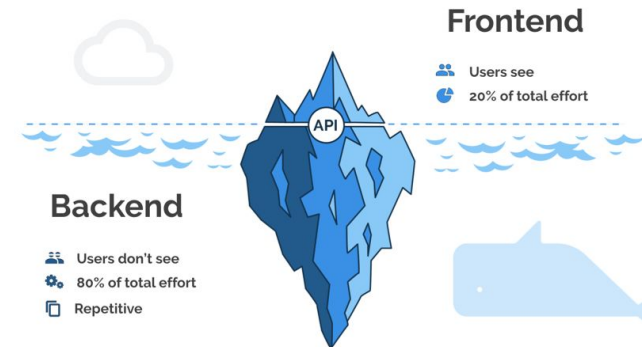


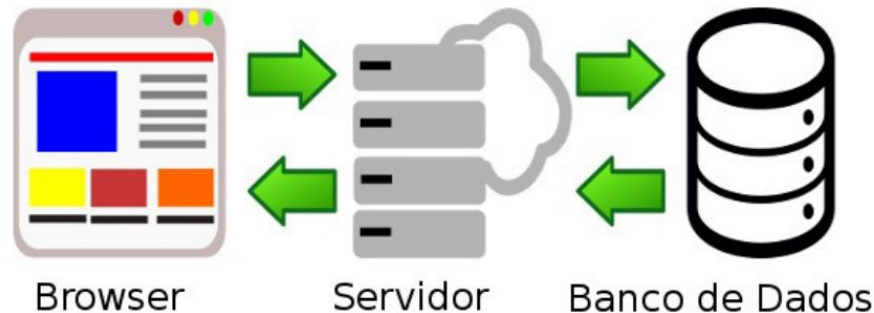
Objetivos da Aula:

- **Entender o FastAPI:** Apresentar o framework, suas vantagens (rapidez, simplicidade e tipagem com Python).
- **Criar um servidor básico:** Configurar um servidor web com FastAPI e demonstrar a criação de endpoints.
- **Definir endpoints:** Explicar o conceito de endpoint (método HTTP + rota).



O profissional fullstack

Para o desenvolvimento front-end, é comum o uso de tecnologias como HTML5, CSS e JavaScript. Já para o back-end temos o PHP, Perl, Java, C#, Python, Ruby e outras linguagens de programação. porém, quando um programador desenvolve tanto para o servidor quanto a nível de usuário, esse é chamado de desenvolvedor full stack.



Criando um Servidor Web com FastAPI (<https://fastapi.tiangolo.com/>)

Instalando o Fastapi

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

Criando um servidor fastapi simples

Crie um arquivo `main.py`:

Rodando o servidor

```
fastapi dev main.py
```

```
from typing import Union
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
def read_root():
    return {"message": "Olá, mundo com FastAPI!"}

@app.get("/items/{item_id}")
def read_item(item_id: int, q: Union[str, None] = None):
    return {"item_id": item_id, "q": q}
```

Criando um Servidor Web com FastAPI (<https://fastapi.tiangolo.com/>)

2ª forma

Rodando o servidor

python main.py

```
from typing import Union
from fastapi import FastAPI
app = FastAPI()

@app.get("/")
def read_root():
    return {"message": "Olá, mundo com FastAPI!"}

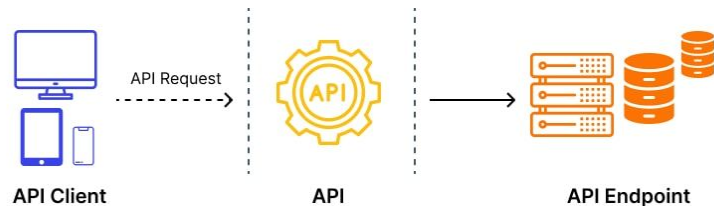
@app.get("/items/{item_id}")
def read_item(item_id: int, q: Union[str, None] = None):
    return {"item_id": item_id, "q": q}

if __name__ == "__main__":
    import uvicorn
    uvicorn.run(app, host="127.0.0.1", port=8000)
```

Conceito de Endpoint

Um endpoint é a junção de um **método** HTTP (GET, POST, PUT, DELETE) e uma **rota** (por exemplo, `/users`), onde o servidor responde a uma solicitação.

Um local centralizado onde é recebido uma **requisição**



```
from typing import Union
from fastapi import FastAPI
app = FastAPI()

@app.get("/")
def read_root():
    return {"message": "Olá, mundo com FastAPI!"}

@app.get("/items/{item_id}")
def read_item(item_id: int, q: Union[str, None] = None):
    return {"item_id": item_id, "q": q}

if __name__ == "__main__":
    import uvicorn
    uvicorn.run(app, host="127.0.0.1", port=8000)
```

Introdução à Rotas

Uma rota é uma URL associada a um método HTTP.

Exemplo: A rota `/` com o método GET responde a uma requisição feita para a raiz da aplicação.

```
from typing import Union
from fastapi import FastAPI
app = FastAPI()

@app.get("/")
def read_root():
    return {"message": "Olá, mundo com FastAPI!"}

@app.get("/items/{item_id}")
def read_item(item_id: int, q: Union[str, None] = None):
    return {"item_id": item_id, "q": q}

if __name__ == "__main__":
    import uvicorn
    uvicorn.run(app, host="127.0.0.1", port=8000)
```

Conceito de Rota

Uma rota é uma associação entre um caminho (por exemplo, “/”, “/sobre”, “/produtos”) e uma função que define o que deve ser feito quando esse caminho é solicitado.

Essa função pode processar dados, renderizar uma view, enviar uma resposta em JSON ou até mesmo redirecionar o usuário para outra página.

Geralmente, uma rota inclui:

- **Método HTTP:** (GET, POST, PUT, DELETE, etc.) – Define o tipo de ação ou operação que será realizada.
- **Caminho da URL:** A parte da URL que identifica a rota, podendo incluir parâmetros dinâmicos.
- **Handler (Controlador):** A função ou o conjunto de funções que processam a requisição e determinam a resposta.

```
EXPLORER  ...  main.py  rotas.py  .gitignore  ▶ ▼

▼ AULA_FASTAPI
  > __pycache__
  > .git
  ▼ routes
    > __pycache__
    📄 rotas.py
  > venv
  📄 .gitignore
  📄 main.py

routes > 📄 rotas.py > ...
1  from fastapi import APIRouter
2  from typing import Union
3
4  router = APIRouter()
5  |
6  @router.get("/")
7  def read_root():
8  |     return {"message": "Olá, mundo com FastAPI!"}
9
10 @router.get("/items/{item_id}")
11 def read_item(item_id: int, q: Union[str, None] = None):
12 |     return {"item_id": item_id, "q": q}
13
14 @router.post("/items")
15 async def create_item(item: dict):
16 |     return {"message": "Recurso criado", "item": item}
17
```


Principais Métodos HTTP

GET: Recupera dados do servidor.

POST: Envia dados ao servidor para criação de um recurso.

PUT/PATCH: Atualiza um recurso existente.

DELETE: Remove um recurso.

<https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Reference/Methods>

```
@app.post("/items")
async def create_item(item: dict):
    return {"message": "Recurso criado", "item": item}

@app.put("/items/{item_id}")
async def update_item(item_id: int, item: dict):
    return {"message": f"Recurso {item_id} atualizado",
            "item": item}

@app.delete("/items/{item_id}")
async def delete_item(item_id: int):
    return {"message": f"Recurso {item_id} removido"}
```

Conceitos de Resposta HTTP e Status

Principais Status de Resposta

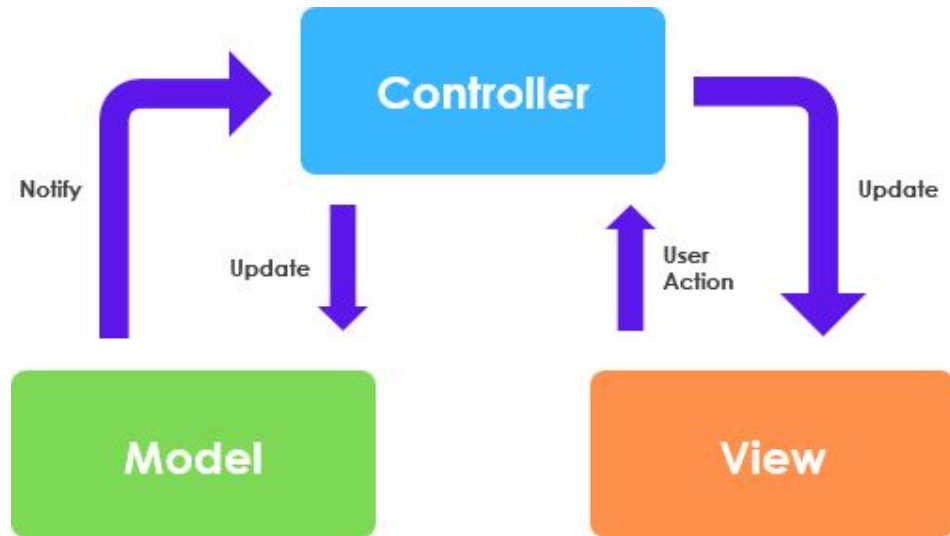
- **200 OK:** Requisição bem-sucedida.
- **201 Created:** Recurso criado (usado em requisições POST).
- **400 Bad Request:** Requisição inválida.
- **401 Unauthorized:** Não autorizado.
- **404 Not Found:** Recurso não encontrado.
- **500 Internal Server Error:** Erro interno do servidor.

<https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Reference/Status>

Introdução ao Padrão MVC

Conceito MVC

- **Model:** Responsável pela camada de dados e regras de negócio (banco de dados, validações).
- **View:** Responsável pela apresentação (templates, HTML).
- **Controller:** Responsável pela lógica que conecta o Model e a View, tratando as requisições e enviando respostas.



Introdução ao Padrão MVC

Usuário: Inicia a ação (por exemplo, ao clicar em um botão ou acessar uma URL).

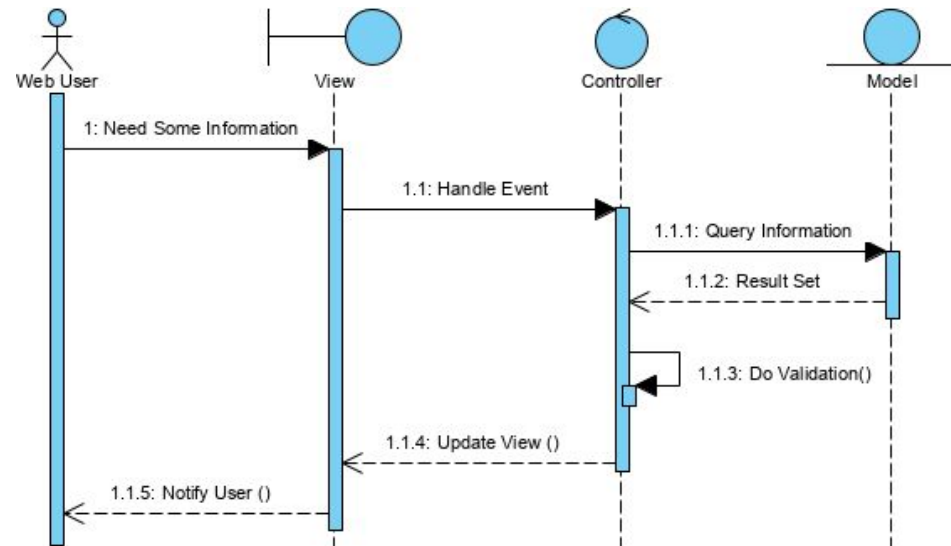
Controller: Recebe a requisição do usuário. Ele é responsável por interpretar a solicitação, validar os dados e decidir qual ação deve ser tomada.

Model: O controller delega a lógica de negócio ao model, que é responsável por acessar os dados (banco de dados, APIs, etc.) e aplicar as regras de negócio.

Controller: Recebe os dados ou resultado da operação executada pelo model.

View: O controller passa os dados para a view, que é responsável por renderizar a interface para o usuário (por exemplo, gerando HTML, JSON, etc.).

Usuário: O usuário vê a resposta renderizada, completando o ciclo da requisição.

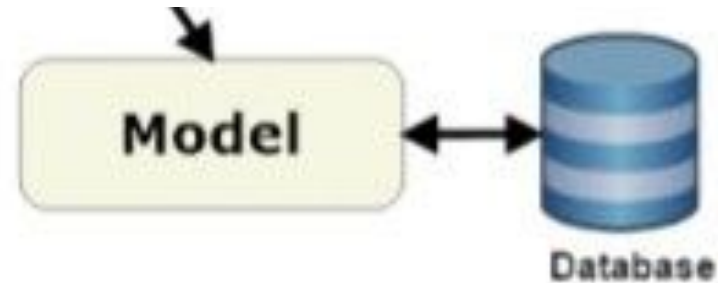


O que é M? (Modelo-Visão-Controlador)

Modelo (Model): Representa a lógica de negócios e os dados da aplicação. na web, isso geralmente significa classes que lidam com o banco de dados, como uma classe **User** que pode interagir com uma tabela de usuários, normalmente reflete a própria tabela do banco de dados.

Classe de Modelo: Singular (User)

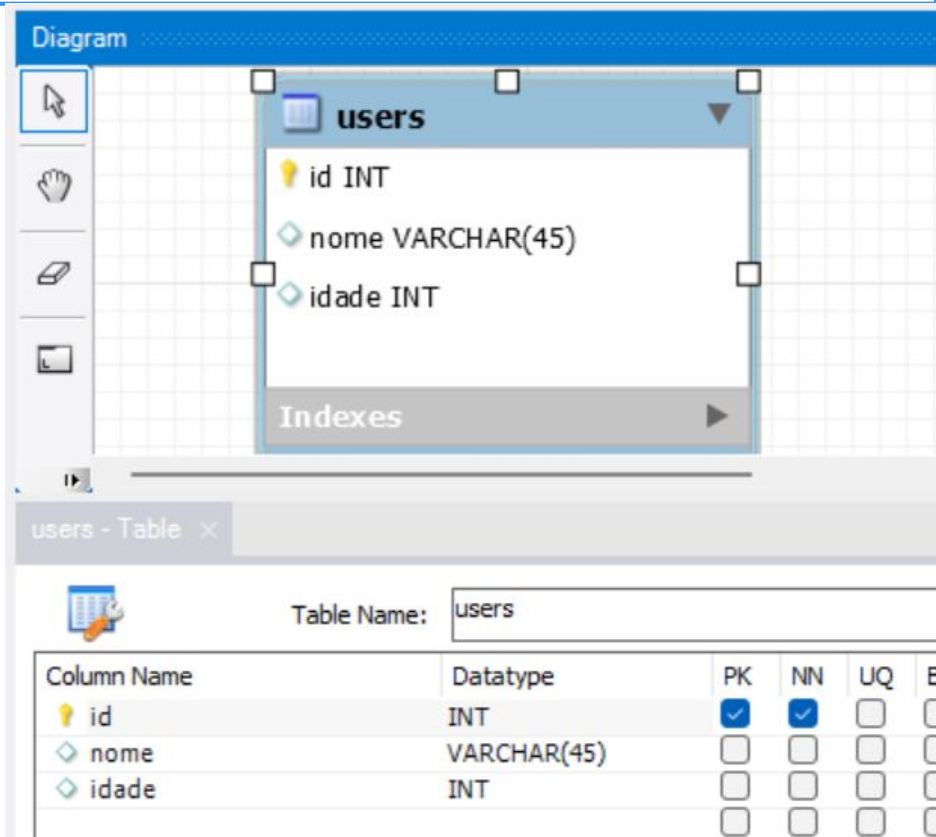
Nome da Tabela no Banco de Dados: Plural (users)



O que é M? (Modelo-Visão-Controlador)

```
from pydantic import BaseModel
from typing import Optional
from uuid import UUID
```

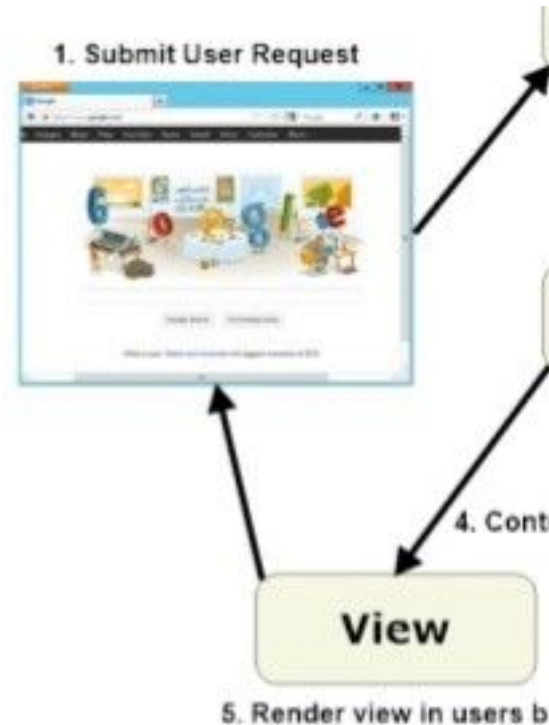
```
class Item(BaseModel):
    id: Optional[UUID] = None
    nome: str
    idade: int
```



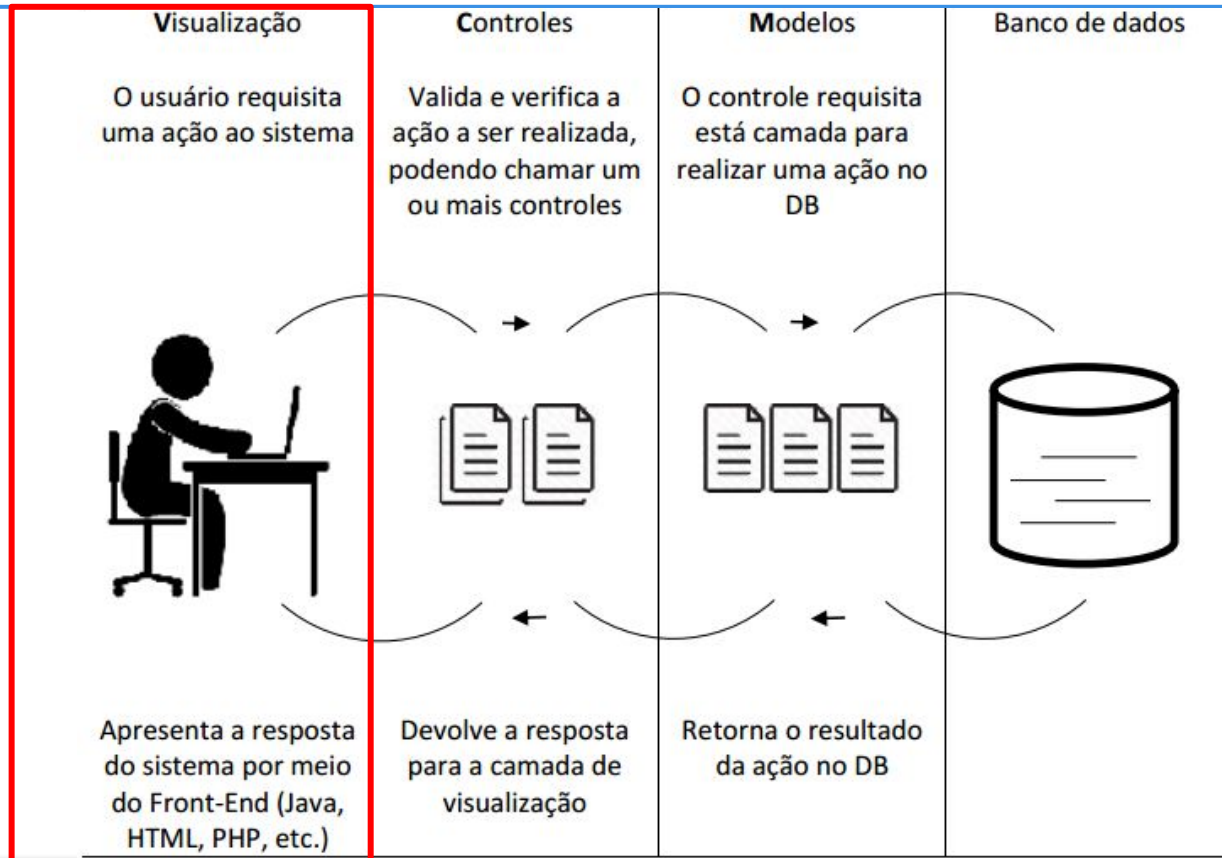
O que é MVC? (Modelo-Visão-Controlador)

Visão (View): Refere-se à interface do usuário. É o que o usuário final vê e interage. No PHP/PYTHON, geralmente são arquivos **HTML com variáveis embutidas**.

No nosso contexto no padrão Rest onde o front-end é separado do back-end, a view é o **front-end**.

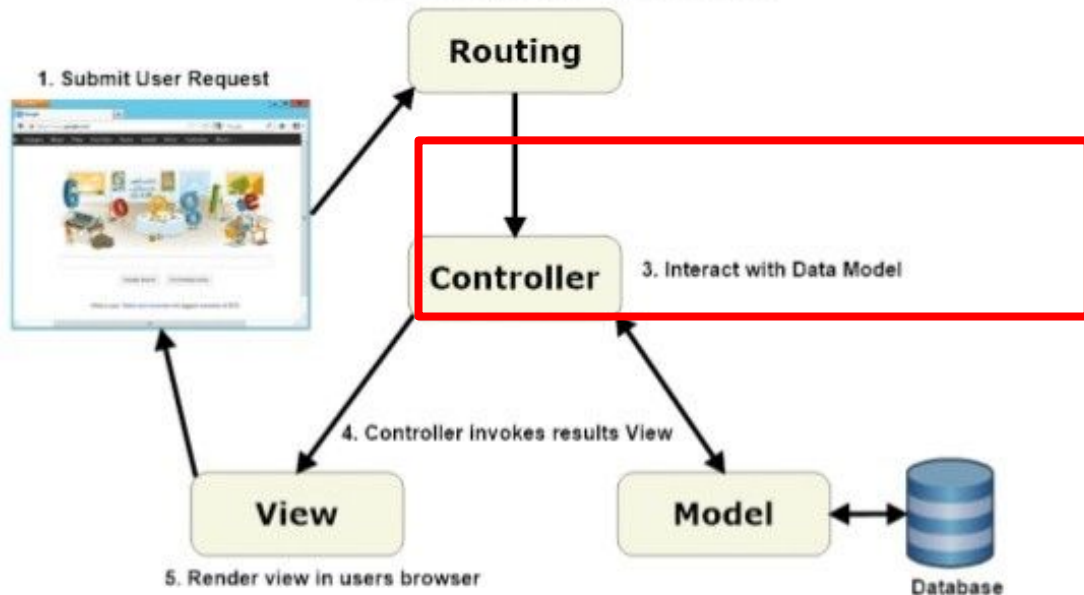


O que é V? (Modelo-Visão-Controlador)

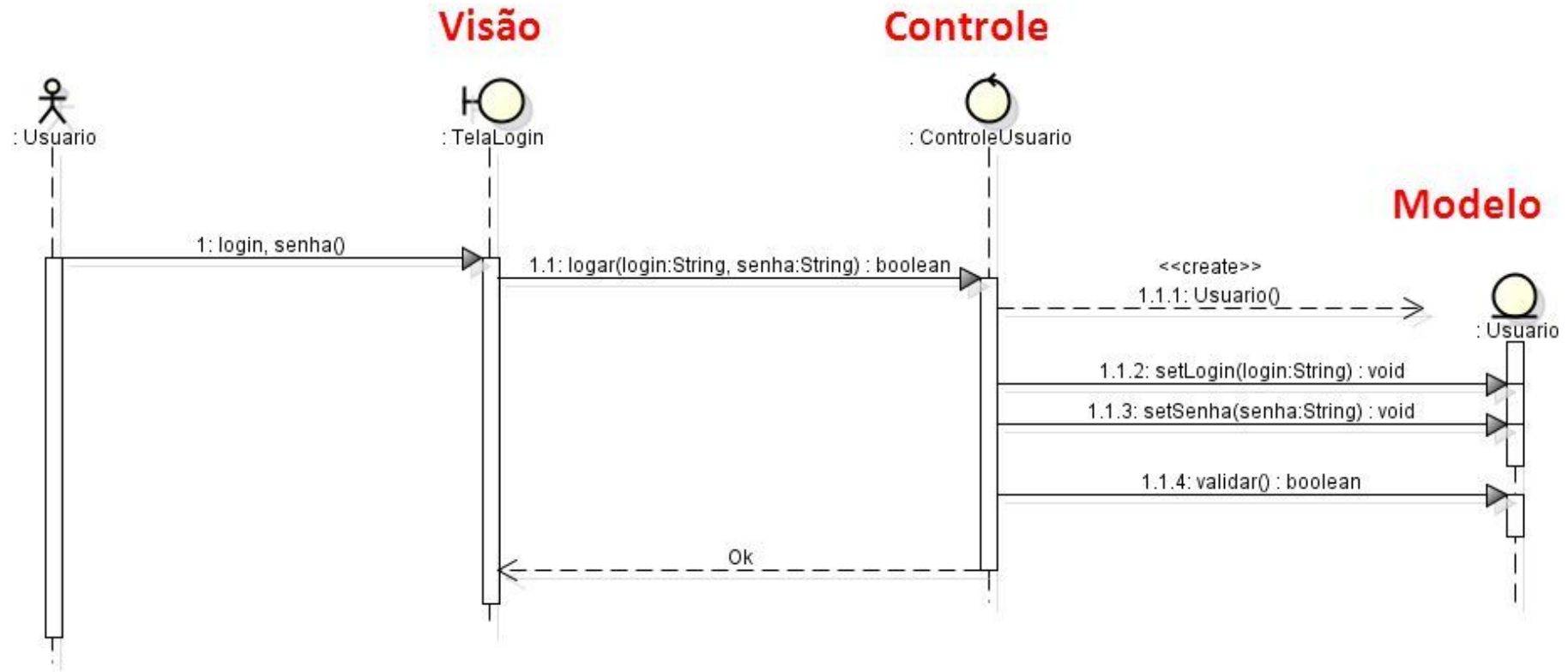


O que é MVC? (Modelo-Visão-Controlador)

Controlador (Controller): Serve como intermediário entre o Modelo e a Visão. Ele processa a entrada do usuário, interage com o Modelo, e decide qual Visão será exibida, e tem papel importante na validação dos dados.







Fluxo de Dados pelo padrão MVC



WEB II - Backend - A_6

EXPLORER


▼ AULA_F...    

> __pycache__

> .git


▼ controllers ●


> __pycache__

 ola_controller.py U

> routes ●

> venv ●

 .gitignore

 main.py

> OUTLINE

main.py

ola_controller.py U ✕

.gitignore

controllers > ola_controller.py > ola_mundo

```
1 def ola_mundo():
2     return {"message": "Olá, mundo com FastAPI!"}
3
```

rotas.py M ✕

routes > rotas.py > read_root

```
2 from typing import Union
3 from controllers.ola_controller import ola_mundo
4
5 router = APIRouter()
6
7 @router.get("/")
8 def read_root():
9     return ola_mundo()
10
11 @router.get("/items/{item_id}")
```

Entendendo Templates e Seus Usos

Vantagens dos Templates

- **Reutilização:** Você pode criar layouts comuns (por exemplo, cabeçalhos, rodapés) e reutilizá-los em várias páginas.
- **Manutenção:** Ao separar a lógica do layout, é mais fácil manter e atualizar a interface do usuário.
- **Dinamicidade:** Permite renderizar páginas com dados variáveis, como resultados de consultas ao banco de dados, sem precisar gerar HTML estático manualmente.

Motores de Templates Comuns

```
<!DOCTYPE html>

<html>

<head>

    <title>{{ title }}</title>

</head>

<body>

    <h1>{{ message }}</h1>

</body>
```

Entendendo Templates e Seus Usos

Instalar a biblioteca:

```
pip install jinja2
```

Motores de Templates Comuns

```
<!DOCTYPE html>
```

```
<html>
```

```
<head>
```

```
  <title>{{ title }}</title>
```

```
</head>
```

```
<body>
```

```
  <h1>{{ message }}</h1>
```

```
</body>
```

AULA_FASTAPI

> __pycache__

> .git

▼ controllers

> __pycache__

ola_controller.py

responde_html.py U

▼ routes

> __pycache__

rotas.py M

▼ templates

<> index.html U

> venv

.gitignore

main.py

controllers > responde_html.py > primeira_pagina

```
1 from fastapi.templating import Jinja2Templates
2 from fastapi import Request
3
4 templates = Jinja2Templates(directory="templates")
5
6 def primeira_pagina(request: Request):
7     return templates.TemplateResponse("index.html", {
8         "request": request,
9         "title": "Página Inicial",
10        "message": "Bem-vindo ao FastAPI com Templates!"
11    })
```

<> index.html U X

templates > <> index.html > html

```
3 <head>
4     <meta charset="UTF-8">
5     <meta name="viewport" content="width=device-width, initial-scale=1.0">
6     <title>{{ title }}</title>
7 </head>
8 <body>
9     <h1>{{ message }}</h1>
10 </body>
11 </html>
```

> OUTLINE

> TIMELINE

> SONARQUBE ISSUE LOCATI

EXPLORER

main.py ola_controller.py **responde_html.py U** rotas.py M index.html U

AULA_FASTAPI

controllers > responde_html.py > primeira_pagina

```
1 from fastapi.templating import Jinja2Templates
2 from fastapi import Request
3
4 templates = Jinja2Templates(directory="templates")
5
6 def primeira_pagina(request: Request):
7     return templates.TemplateResponse("index.html", {
8         "request": request,
9         "title": "Página Inicial",
10        "message": "Bem-vindo ao FastAPI com Templates!"
11    })
```

index.html U

templates > index.html > html

```
3 <head>
4     <meta charset="UTF-8">
5     <meta name="viewport" content="width=device-width, initial-scale=1.0">
6     <title>{{ title }}</title>
7 </head>
8 <body>
9     <h1>{{ message }}</h1>
10 </body>
11 </html>
```

The image is a screenshot of a Visual Studio Code editor window. The Explorer sidebar on the left shows a project structure for 'AULA_FASTAPI'. It includes folders like '__pycache__', '.git', 'controllers', 'routes', 'templates', and 'venv'. The 'controllers' folder is expanded, showing 'ola_controller.py' and 'responde_html.py'. The 'responde_html.py' file is selected and highlighted with a red box. A red arrow points from this box to the 'primeira_pagina' function in the main editor. The 'templates' folder is also expanded, showing 'index.html', which is also highlighted with a red box. A red arrow points from this box to the 'index.html' file in the main editor. The main editor displays the code for 'primeira_pagina' in 'responde_html.py' and the HTML template in 'index.html'. The code uses Jinja2 for templating, passing 'request', 'title', and 'message' to the 'TemplateResponse'. The HTML template uses these variables to set the title and message in the body. A red arrow points from the 'request' parameter in the function signature to the 'request' key in the dictionary passed to 'TemplateResponse'. Another red arrow points from the 'message' key in the dictionary to the 'message' variable in the Jinja2 template.

Entendendo Templates e Seus Usos

alterações

incluir o html para template
criar no controller a forma de
injetar as respostas no template
e no arquivo de rotas

```
from fastapi import APIRouter, Request
from typing import Union
from controllers.ola_controller import ola_mundo
from controllers.responde_html import primeira_pagina

router = APIRouter()

@router.get("/view", response_model=None)
async def chama_template(request: Request):
    return primeira_pagina(request)
```


Referências

BEAZLEY D.; JONES, B.K. Python Cookbook: Receitas para dominar Python. 3 ed. São Paulo: Novatec, 2019.

CRUZ, FJ. Python: escreva seus primeiro programas. 1.ed. São Paulo: Casa do Código, 2021.

PEREIRA, E, DOUGLAS MICHAEL. Trilhas Python: Programação multiparadigma e desenvolvimento web com Flask. 1.ed. São Paulo: Casa do Código, 2020.

